**ASSIGNMENT HELP**

**MANUAL**



SUBMITTED

TO

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY, PUNE

FOR THE SKILL AND COMPETENCY EVALUATION OF

ARTIFICIAL INTELLIGENCE [CAUA31201]

IN

**CSE AI DEPARTMENT**

BY

**Vedant Rakesh Mukhekar**

**Class: T.Y. BTech Division: A Batch: A2**

**Batch Teacher**

**Dr. ANURADHA YENKIKAR.**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **SR. NO.** | **CONTENTS** | **PAGE NO.** |
| **1** | **PROBLEM STATEMENT** | **4-5** |
| **2** | **LIBRARY USED** | **5-6** |
| **3** | **THEORY** | **6-7** |
| **4** | **METHODOLOGY** | **7-14** |
| **5** | **ADVANTAGES & DISADVANTAGES** | **14-15** |
| **6** | **WORKING** | **16-17** |
| **7** | **DIAGRAM** | **17-18** |
| **8** | **CONCLUSION** | **18-19** |

### Problem Statement

The **8-Queens Problem** is a classic combinatorial problem in which the objective is to place eight queens on a chessboard in such a way that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The challenge is to find all possible arrangements of the eight queens on the board.

The problem can be approached using basic search strategies such as backtracking, which systematically explores all potential configurations until a valid solution is found or all configurations are exhausted. The aim is to implement a solution that finds all valid placements of the queens on the board.

### Libraries Used

* **Python Standard Libraries**:
  + No external libraries are required for the basic implementation.

### Theory

The **8-Queens Problem** is a classic example of a constraint satisfaction problem (CSP) in combinatorial optimization. It challenges the solver to place eight queens on an 8x8 chessboard in such a way that no two queens threaten each other. This problem is notable in the fields of artificial intelligence, algorithms, and combinatorial optimization for several reasons.

#### Problem Representation

In the context of the 8-Queens Problem:

* **State**: A specific arrangement of queens on the chessboard.
* **Initial State**: An empty chessboard with no queens placed.
* **Actions**: Placing a queen in a specific row and column.
* **Goal State**: A configuration where all eight queens are placed without any two queens attacking each other.

#### Constraints

The constraints for this problem can be categorized as:

1. **Row Constraint**: Each queen must occupy a unique row. This is inherently satisfied by placing one queen per row, as queens are placed sequentially from row 0 to row 7.
2. **Column Constraint**: Each queen must occupy a unique column. No two queens can be placed in the same column.
3. **Diagonal Constraint**: Queens placed on the same diagonal can attack each other. Two queens are on the same diagonal if the absolute difference between their row indices equals the absolute difference between their column indices. This can be expressed mathematically as:
   * For queens at positions (r1,c1)(r1, c1)(r1,c1) and (r2,c2)(r2, c2)(r2,c2), they are on the same diagonal if:
     + ∣r1−r2∣=∣c1−c2∣|r1 - r2| = |c1 - c2|∣r1−r2∣=∣c1−c2∣

These constraints create a search space where each potential configuration can either be valid (no queens attacking each other) or invalid (at least one queen threatens another).

#### Search Strategies

The problem can be approached using different search strategies, with backtracking being one of the most effective.

1. **Backtracking**:
   * This algorithm is a depth-first search method that explores all potential placements of queens recursively.
   * When a queen is placed, the algorithm checks if the placement is valid by applying the constraints.
   * If a valid placement is found, the algorithm recursively attempts to place queens in the next row.
   * If a placement leads to a conflict, the algorithm backtracks by removing the last placed queen and trying the next column in the previous row.
2. **Performance**:
   * The naive approach would be to try every possible configuration, which would be computationally expensive (on the order of O(n!)O(n!)O(n!) for nnn queens).
   * Backtracking significantly reduces the search space by eliminating invalid configurations early and only exploring valid placements.

### Methodology

1. **Define the Chessboard**:
   * Use an 8x8 matrix (list of lists) to represent the chessboard. Each cell can either be empty or occupied by a queen.
2. **Implement the Backtracking Algorithm**:
   * Start placing queens row by row.
   * For each row, attempt to place a queen in each column and check if the placement is valid.
   * If valid, recursively attempt to place queens in the next row.
   * If all queens are placed successfully, record the solution.
   * If a placement leads to a conflict, backtrack and try the next column.
3. **Record and Display Solutions**:
   * Store each valid arrangement of the queens and display them in a readable format.

### Advantages & Disadvantages

* **Advantages**:
  + Backtracking is a straightforward and systematic approach to solving the problem.
  + It can efficiently find all solutions without exploring unnecessary configurations.
* **Disadvantages**:
  + The performance may degrade for larger board sizes due to the exponential growth of possibilities.
  + It can be less efficient compared to optimized algorithms, like constraint propagation or using heuristics.

### Working Example (Python Code)

python

Copy code

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell else "." for cell in row))

print()

def is\_safe(board, row, col):

# Check the column

for i in range(row):

if board[i][col]:

return False

# Check the upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if j < 0:

break

if board[i][j]:

return False

# Check the upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, len(board))):

if j >= len(board):

break

if board[i][j]:

return False

return True

def solve\_n\_queens(board, row, solutions):

if row == len(board):

solutions.append([r[:] for r in board]) # Append a copy of the solution

return

for col in range(len(board)):

if is\_safe(board, row, col):

board[row][col] = True # Place the queen

solve\_n\_queens(board, row + 1, solutions)

board[row][col] = False # Remove the queen (backtrack)

def eight\_queens():

n = 8

board = [[False for \_ in range(n)] for \_ in range(n)]

solutions = []

solve\_n\_queens(board, 0, solutions)

print(f"Total Solutions: {len(solutions)}\n")

for i, solution in enumerate(solutions):

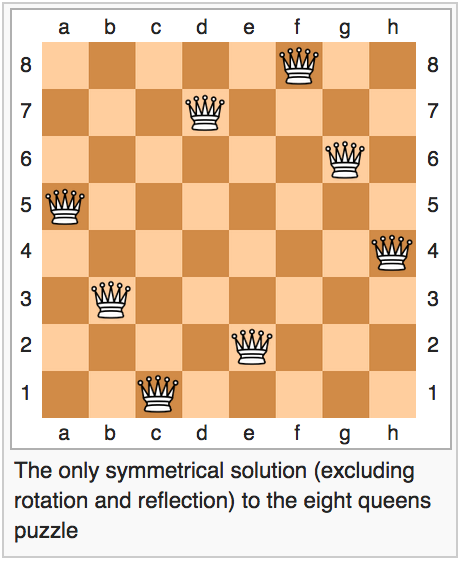
print(f"Solution {i + 1}:")

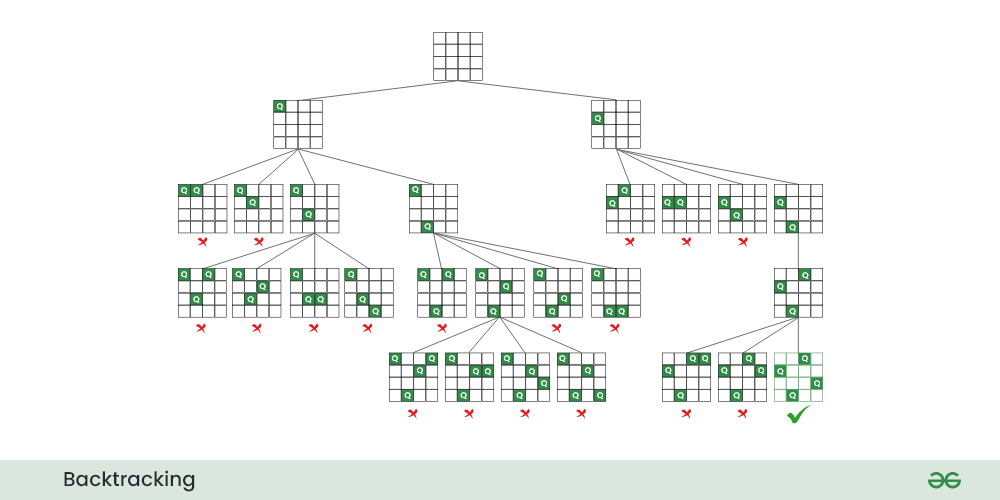
print\_solution(solution)

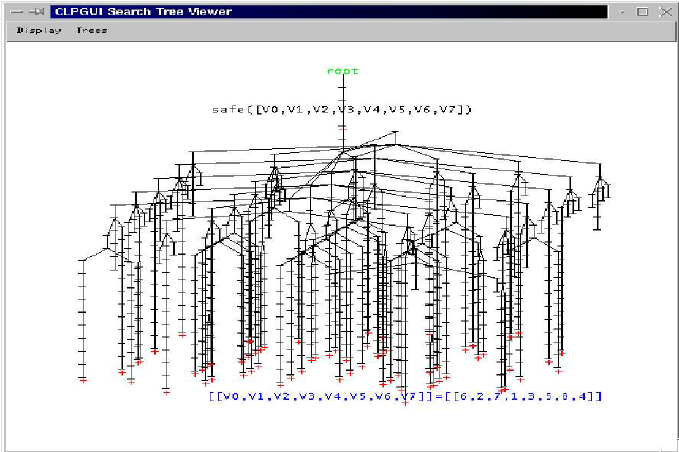
# Run the eight queens problem solver

eight\_queens()

### Diagram







### Conclusion

The **8-Queens Problem** serves as an excellent example of the application of backtracking algorithms in solving combinatorial problems. This implementation effectively finds all valid arrangements of queens on a chessboard, demonstrating the utility of basic search strategies in artificial intelligence and computational problem-solving. The backtracking approach not only provides all possible solutions but also illustrates the systematic exploration of the problem space.

Top of Form

Bottom of Form