N-Queens Problem

Name -: Aman Yadav

Roll no. -: 202401100400028

Course -: BTech

Branch -: Cse(Ai&Ml)

Course Code -: AI101B

INTRODUCTION

N-Queens Problem

The N-Queens problem is a constraint satisfaction problem in which we must place N queens on an N × N chessboard so that no two queens attack each other.

Constraints:

A valid placement ensures that:

1. **No two queens share the same row.**
2. **No two queens share the same column.**
3. **No two queens share the same diagonal (both main and anti-diagonal).**

Example:

For **N = 4**, one possible valid solution is:

```
css

. Q . .
. . . Q
Q . . .
. . Q .
```

METHODOLOGY

Simple Explanation of How to Solve N-Queens

The goal is to place N queens on an N × N chessboard so that no two queens attack each other.

Approaches to Solve It

1. Backtracking (Try & Fix) – Best for Small N

- **Place queens one by one in rows.**
- **If a queen attacks another, move it to a different column.**
- **If no valid position, go back (undo last move) and try again.**
- **Repeat until all queens are placed correctly.**

Example:

👑 ✅ 👑 ❌ → Move the ❌ queen somewhere else!

✅ Works well for small boards (N ≤ 15), but slow for large boards.

CODE

```python
import random

def random_board(n):
    """Generate a random board where each queen is in a unique row and column."""
    return random.sample(range(n), n)

def count_conflicts(board):
    """Count the number of diagonal conflicts."""
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i+1, n):
            if abs(board[i] - board[j]) == abs(i - j):  # Check diagonals
                conflicts += 1
    return conflicts

def hill_climbing(n, max_iterations=1000):
    """Solve the N-Queens problem using Hill Climbing."""
    current_board = random_board(n)
    current_conflicts = count_conflicts(current_board)

    for _ in range(max_iterations):
        if current_conflicts == 0:
            return current_board  # Solution found
```

```python
        # Generate all possible neighbor states by swapping two queens
        best_neighbor = current_board
        best_conflicts = current_conflicts
        for i in range(n):
            for j in range(i+1, n):
                neighbor = list(current_board)
                neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap two queens
                conflicts = count_conflicts(neighbor)
                if conflicts < best_conflicts:
                    best_neighbor = neighbor
                    best_conflicts = conflicts

        # If no better neighbor, return the current board (local minimum)
        if best_conflicts >= current_conflicts:
            return current_board

        # Move to the best neighbor
        current_board = best_neighbor
        current_conflicts = best_conflicts

    return current_board  # Return the best solution found
```

```python
def print_board(board):
    """Print the board in a readable format."""
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

# Main
n = int(input("Enter the size of the board (N ≥ 4): "))
if n < 4:
    print("N must be at least 4.")
else:
    solution = hill_climbing(n)
    print("Solution:")
    print_board(solution)
    print("Conflicts:", count_conflicts(solution))
```

OUTPUT

```
Execution output from 20 Feb 2025 14:24
0KB
  Stream
    Enter the size of the board (N ≥ 4): 4
    Solution:
    . . Q .
    Q . . .
    . . . Q
    . Q . .

    Conflicts: 0
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
print(df.head())
```

	BoardID	N	Board Placement	Validity	Difficulty Level
0	1	4	[0, 1, 2, 3]	1	Easy
1	2	4	[0, 1, 3, 2]	1	Easy
2	3	4	[0, 2, 1, 3]	1	Easy
3	4	4	[0, 2, 3, 1]	1	Easy
4	5	4	[0, 3, 1, 2]	1	Easy

```
print(df.shape)
```

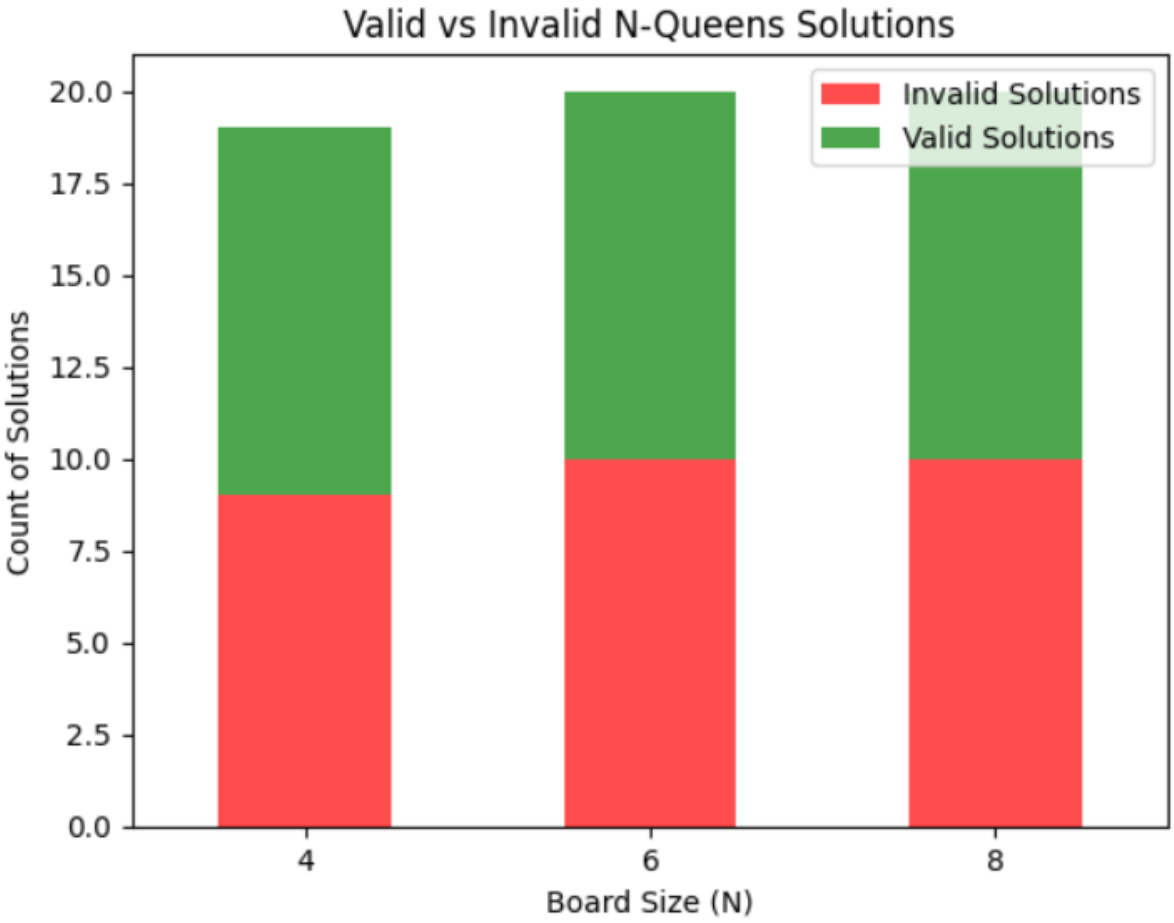
```
(59, 5)
```

```
print(df.describe())
```

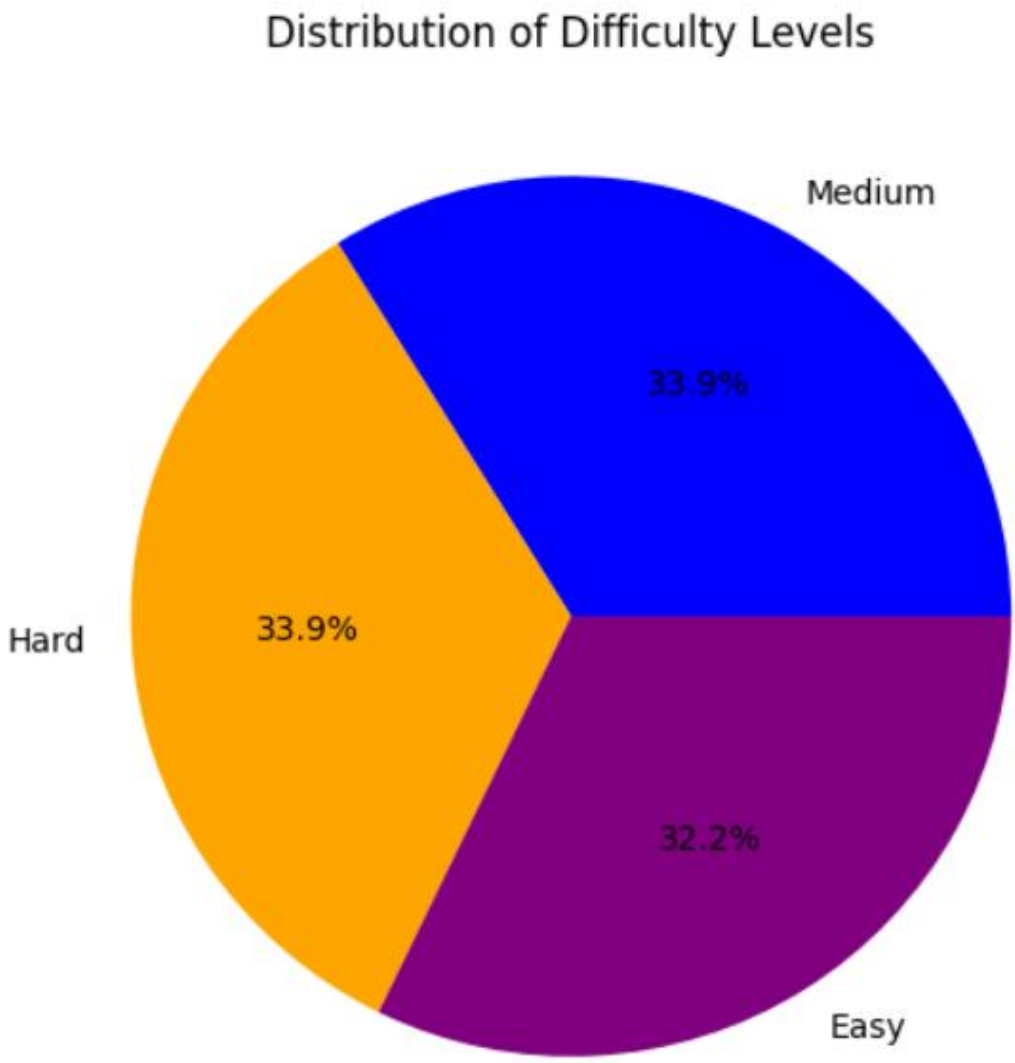
	BoardID	N	Validity
count	59.000000	59.000000	59.000000
mean	30.000000	6.033898	0.508475
std	17.175564	1.639660	0.504219
min	1.000000	4.000000	0.000000
25%	15.500000	4.000000	0.000000
50%	30.000000	6.000000	1.000000
75%	44.500000	8.000000	1.000000
max	59.000000	8.000000	1.000000

```
# Convert 'Board Placement' from string to list (if necessary)
df["Board Placement"] = df["Board Placement"].apply(lambda x: eval(x) if isinstance(x, str) else x)


# Group by board size (N) and count valid/invalid solutions
```

```
validity_counts = df.groupby(["N",
"Validity"]).size().unstack().fillna(0)

validity_counts.columns = ["Invalid Solutions", "Valid Solutions"]


# 📊 **Bar Chart: Valid vs Invalid Solutions**

plt.figure(figsize=(8, 5))

validity_counts.plot(kind="bar", stacked=True, color=["red",
"green"], alpha=0.7)

plt.xlabel("Board Size (N)")

plt.ylabel("Count of Solutions")

plt.title("Valid vs Invalid N-Queens Solutions")

plt.legend(["Invalid Solutions", "Valid Solutions"])

plt.xticks(rotation=0)

plt.show()


# 📊 **Pie Chart: Distribution of Difficulty Levels**

plt.figure(figsize=(6, 6))

df["Difficulty Level"].value_counts().plot.pie(autopct="%1.1f%%",
colors=["blue", "orange", "purple"])

plt.title("Distribution of Difficulty Levels")

plt.ylabel("")

plt.show()
```



Distribution of Difficulty Levels



REFERENCES/CREDITS

Backtracking Approach:

- **D. Bitner & J. G. Reingold, *Backtracking: A General Algorithmic Technique*, 1975.**
- **Link: https://doi.org/10.1145/321906.321909**