**Shadedra Williams and Ahmad Sibai**

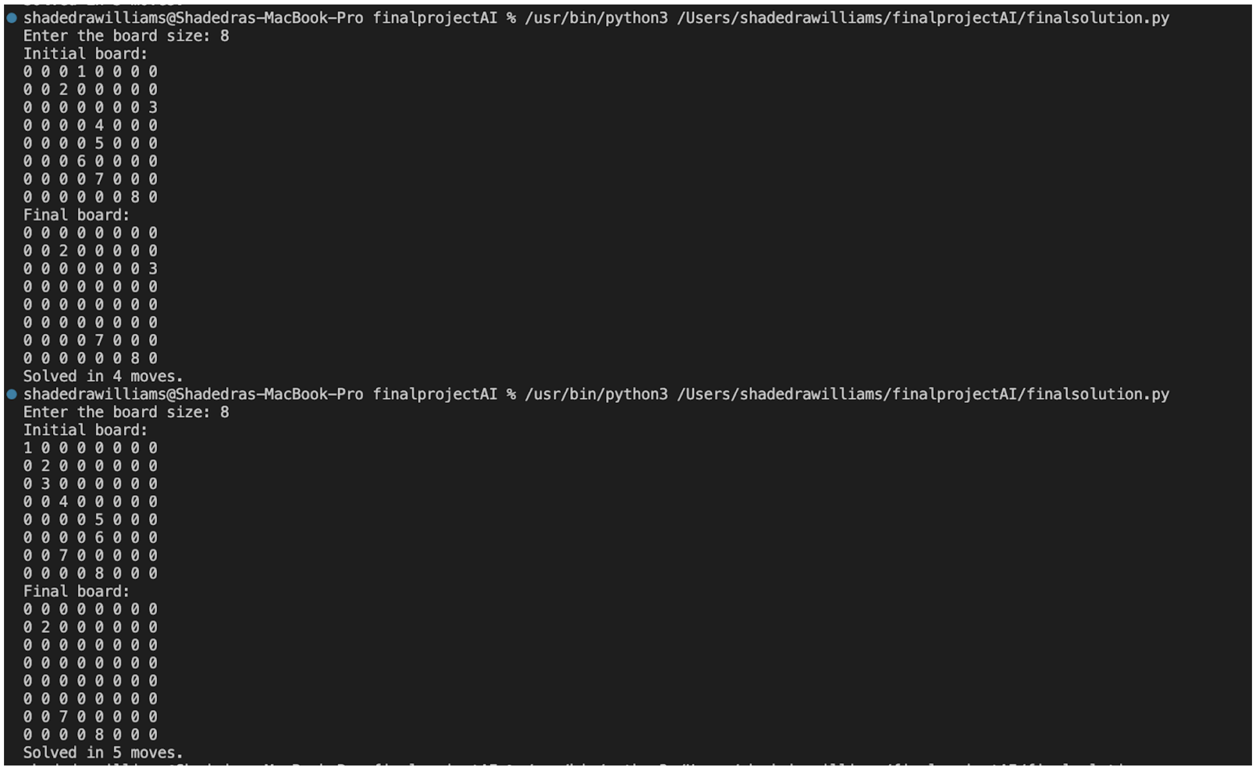
# N Queens Problem using Constraint Satisfaction Problem

The N Queens problem is one in which we are given n queens to place on a N * N chess board in such a manner that no two queens should be able to attack each other. Consequently, neither one of the n queens being placed on the board can be in the same row, column or diagonal. This problem can be solved using methods such as backtracking and constraint satisfaction problems (CSP). The CSP method is more efficient than the backtracking method. Our system utilizes code that models the problem as a Constraint Satisfaction Problem solved using a minimum conflict heuristic. It produces solutions for all natural numbers excluding 2 and 3.

Our primary function for resolving the N-Queens problem is the **solve_n_queens** function. It takes the board size as input, initializes the board, prints the initial board state, and then proceeds to move the queens to less conflicting positions until the board is solved or a maximum number of moves is reached. The program makes use of multiple auxiliary functions to assist this function in solving the problem.  One such function is the **count_conflicts** function which takes the current board state and a row and column position as parameters and returns the number of conflicts that the queen at that position has with other queens on the board. Every conflict for each queen on the board is located by the **find_conflicts** function, which returns the results as a dictionary. The board's empty positions are all located by the **find_empty_positions** function, which returns them as a list of tuples. Using the current board state, a row and column position, and the **move_queen function**, the queen at the specified row, column position is moved to a less conflicting location. The function then returns the new row and column position of the queen.  The initial board with randomly and potentially conflicting queen placements is generated by the **initial_board** function which takes a board size N as input and initializes an NxN board represented by a 2D array. The **print_board** function prints the current board state to the console.

The solve_n_queens function starts by prompting the user to enter the board size. If the board size is equal to 2 or 3, the function prints an error message and prompts the user to enter a board size that is not equal to 2 or 3. The function then utilizes the **intial_board** and **print_board** functions to initialize the board and print the initial board state. The variable representing the number of moves to solve the problem is then set to 0. The function then enters a loop where it uses the **find_conflicts** function to find the conflicts for every queen on the board, and if all the conflicts are zero, the board is solved, and the function exits the loop. If not, the function picks the queen with the greatest number of conflicts, which is determined by the **count_conflicts** function, and moves it to a less conflicting position. If the number of moves exceeds 250, the function prints a message indicating that the maximum number of moves has been reached and outputs the current board state. This restriction on the number of moves was implemented as we found that the number of moves required to solve the problem increases as the board size increases, and the time required to solve the problem increases exponentially as the board size increases. Finally, the function prints the final board state and the number of moves it took to solve the board.

To test we executed the code with different board sizes and measured the time and number of moves required to solve the problem. We also measured the percentage of solved boards for different board sizes. For board sizes larger than 20, the code takes a long time to solve the problem and frequently gets stuck in local minima. We also realized that depending on the initial state of the board, the number of moves to solve the problem will vary. Thus, for example, when 8 is entered it will solve the problem in 4 moves then once re-entered, the initial solution will require a different number of moves (see figure 1 below).



```
shadedrawilliams@Shadedras-MacBook-Pro finalprojectAI % /usr/bin/python3 /Users/shadedrawilliams/finalprojectAI/finalsolution.py
Enter the board size: 8
Initial board:
0 0 0 1 0 0 0 0
0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 3
0 0 0 0 4 0 0 0
0 0 0 0 5 0 0 0
0 0 0 6 0 0 0 0
0 0 0 0 7 0 0 0
0 0 0 0 0 0 8 0
Final board:
0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 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 7 0 0 0
0 0 0 0 0 0 8 0
Solved in 4 moves.
shadedrawilliams@Shadedras-MacBook-Pro finalprojectAI % /usr/bin/python3 /Users/shadedrawilliams/finalprojectAI/finalsolution.py
Enter the board size: 8
Initial board:
1 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0
0 3 0 0 0 0 0 0
0 0 4 0 0 0 0 0
0 0 0 0 5 0 0 0
0 0 0 0 6 0 0 0
0 0 7 0 0 0 0 0
0 0 0 0 8 0 0 0
Final board:
0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 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 0 0 0 0 0
0 0 0 0 8 0 0 0
Solved in 5 moves.
```

Figure 1: Image showing varying results when number of queens = 8

The N Queens problem is a puzzle where n queens need to be placed on a chess board of N * N size such that no two queens can attack each other. The problem can be solved using backtracking and Constraint Satisfaction Problem (CSP) techniques, with CSP being the better approach. The system presented uses a minimum conflict heuristic to solve the problem for all natural numbers except 2 and 3. The solve_n_queens function is the primary function, which takes the board size as input and initializes the board, prints the initial board state, and moves the queens to less conflicting positions until the board is solved or a maximum number of moves is reached. Auxiliary functions like count_conflicts, find_conflicts, find_empty_positions, and move_queen are used to assist the primary function. The code was tested with different board sizes to measure the time, number of moves, and percentage of solved boards. The number of moves to solve the problem varies depending on the initial state of the board. For larger board sizes, the code takes a long time to solve the problem and can get stuck in local minima.