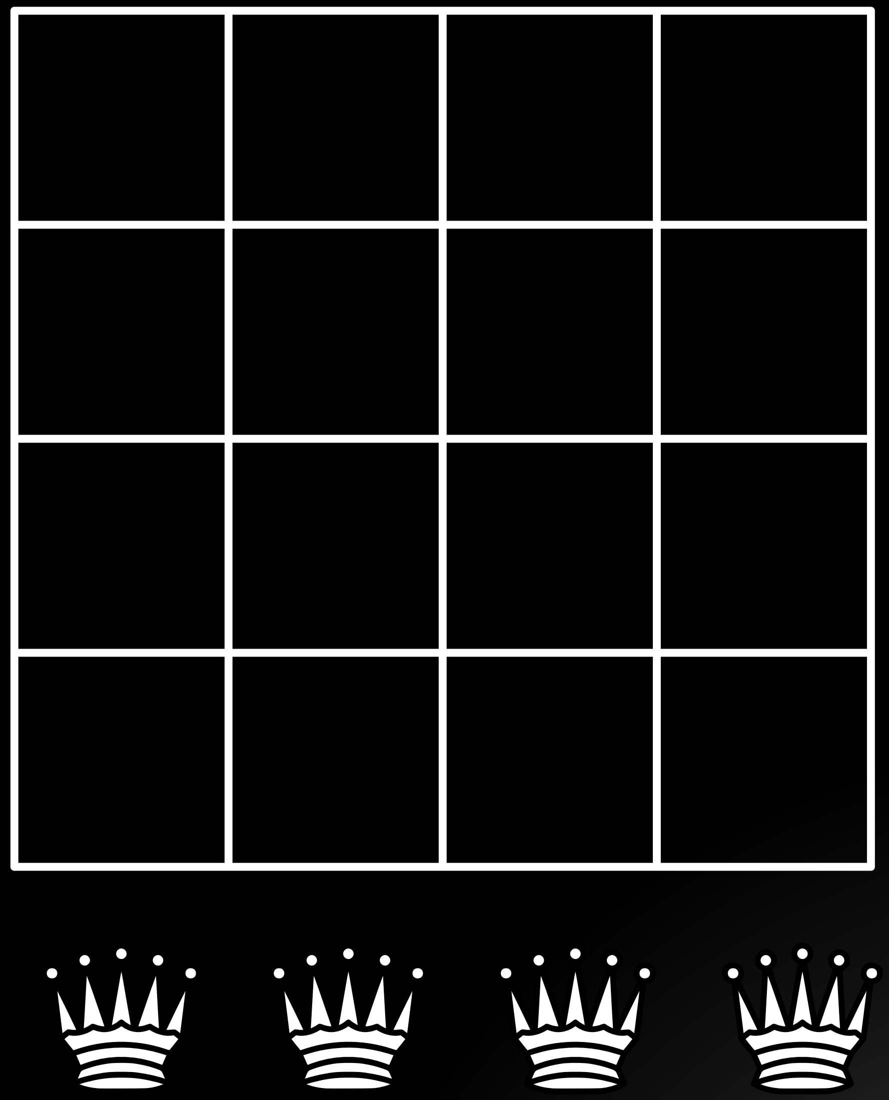
N QUEEN VISUALIZER

PRESENTATION

PROBLEM STATEMENT

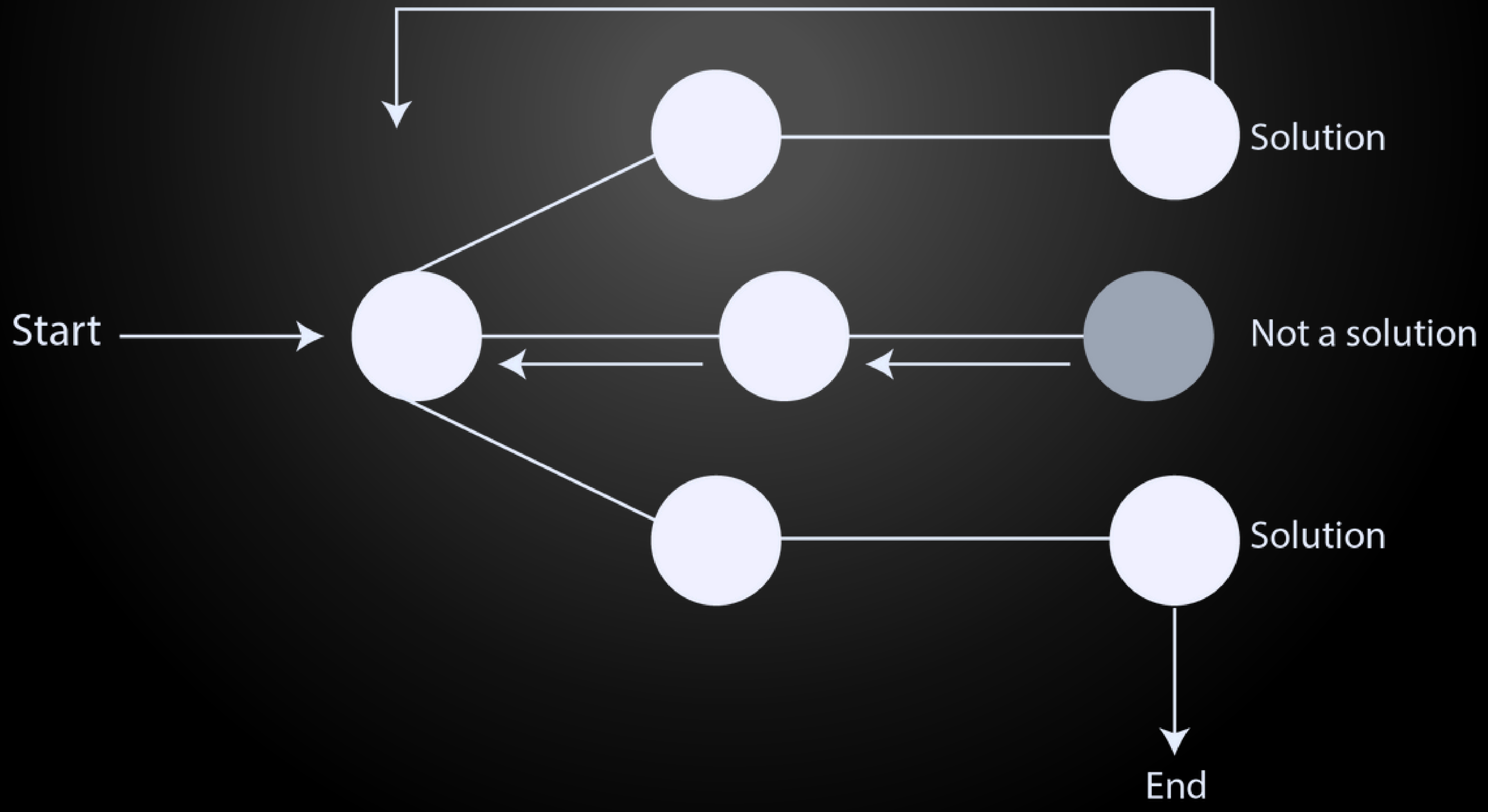
- Place N queens on an N×N chessboard.
- No two queens threaten each other.

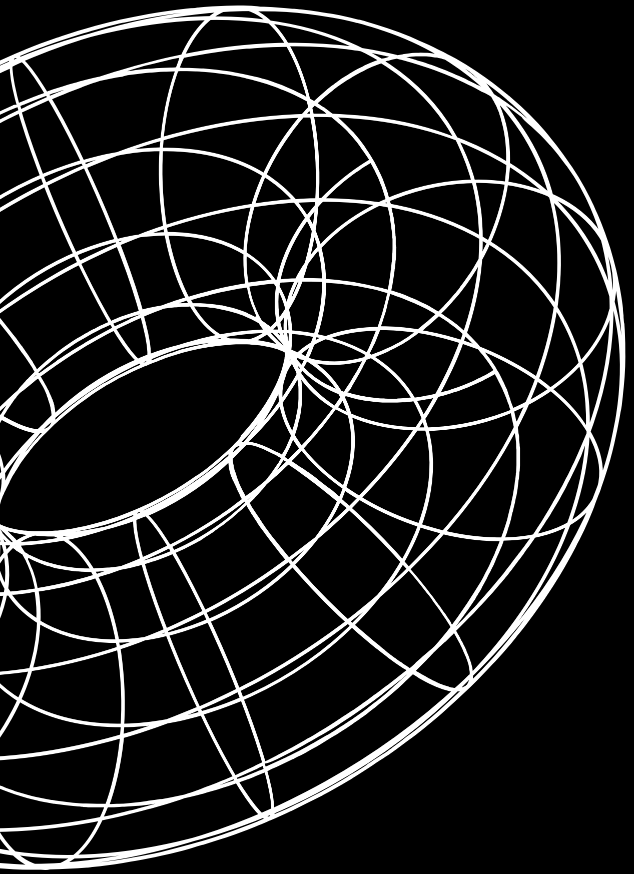
Ensuring no two queens share the same row, column, or diagonal.



ALGORITHM USED

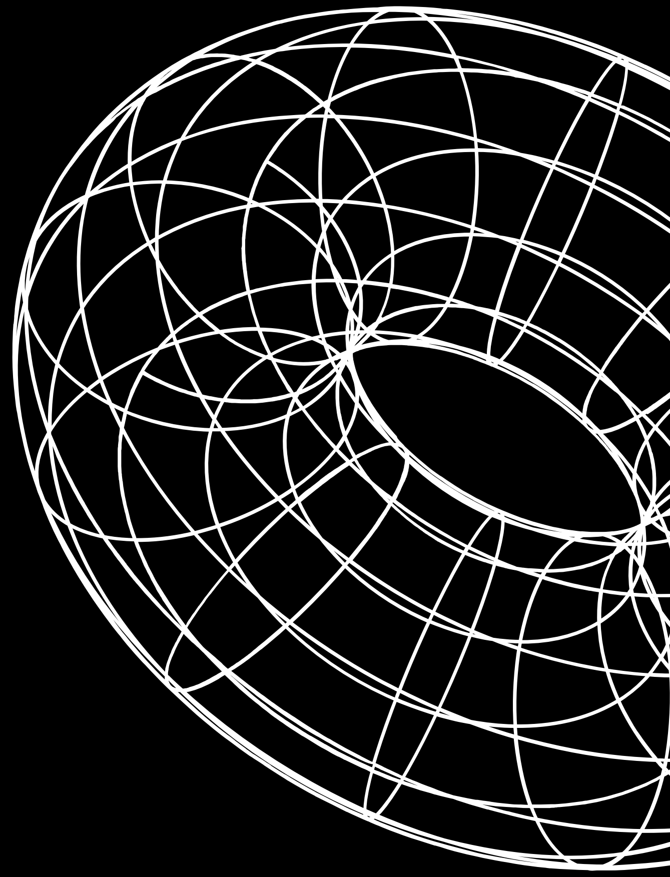
BACKTRACKING





EXPLANATION

- Place a queen in the first row.
- Move to the next row and try placing a queen in a safe column.
- Continue until all queens are placed or no safe columns remain.
- Backtrack if no safe column is found.



FUNCTIONS USED

```
void printBoard(const vector<int>& board, int N) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      if (board[i] == j) {
        cout << "Q ";
      } else {
        cout << ". ";
      }
    }
    cout << endl;
  }
  cout << endl;
  usleep(500000); // Sleep for 0.5 seconds to
visualize the steps
}
```

Purpose: This function prints the current state of the chessboard.

How it works:

- It iterates over each row and column of the board.
- It prints 'Q' if there's a queen in that column for the current row; otherwise, it prints '.'.
- It pauses the execution for 0.5 seconds to allow visualization of each step.

FUNCTIONS USED

```
bool isSafe(const vector<int>& board, int row, int col,
int N) {
  for (int i = 0; i < row; ++i) {
    if (board[i] == col || board[i] - i == col - row ||
board[i] + i == col + row) {
      return false;
    }
  }
  return true;
}
```

Purpose: This function checks if placing a queen at a given row and column is safe, meaning no other queen can attack it.

How it works:

- It checks all previously placed queens (up to the current row) to see if any of them are in the same column, or on the same major or minor diagonal.
- Returns true if the position is safe, otherwise returns false.

FUNCTIONS USED

```
void solveNQueensUtil(vector<int>& board, int row, int N,
vector<vector<int>>& solutions) {
    if (row == N) {
        solutions.push_back(board);
        printBoard(board, N); // Print each found solution
        return;
    }

    for (int col = 0; col < N; ++col) {
        if (isSafe(board, row, col, N)) {
            board[row] = col;
            printBoard(board, N); // Print board at each step
            solveNQueensUtil(board, row + 1, N, solutions);
            board[row] = -1; // Backtrack
            printBoard(board, N); // Print board after backtracking
        }
    }
}
```

Purpose: This is a recursive function that tries to place queens on the board and backtracks if a conflict is found. It is the core of the backtracking algorithm.

How it works:

- If row equals N, a complete solution is found, and it is added to the solutions vector.
- Otherwise, it attempts to place a queen in each column of the current row.
- For each column, it checks if it's safe to place a queen.
- If safe, it places the queen, recursively calls itself for the next row, and then backtracks (removes the queen) to try the next column.

FUNCTIONS USED

```
void solveNQueens(int N) {
    vector<int> board(N, -1);
    vector<vector<int>> solutions;
    solveNQueensUtil(board, 0, N, solutions);

    if (solutions.empty()) {
        cout << "No solution exists for " << N << " queens." << endl;
    } else {
        cout << "Final Solutions:\n";
        for (const auto& sol : solutions) {
            printBoard(sol, N);
        }
    }
}
```

Purpose: This function initializes the board and starts the recursive backtracking process to solve the N-Queens problem.

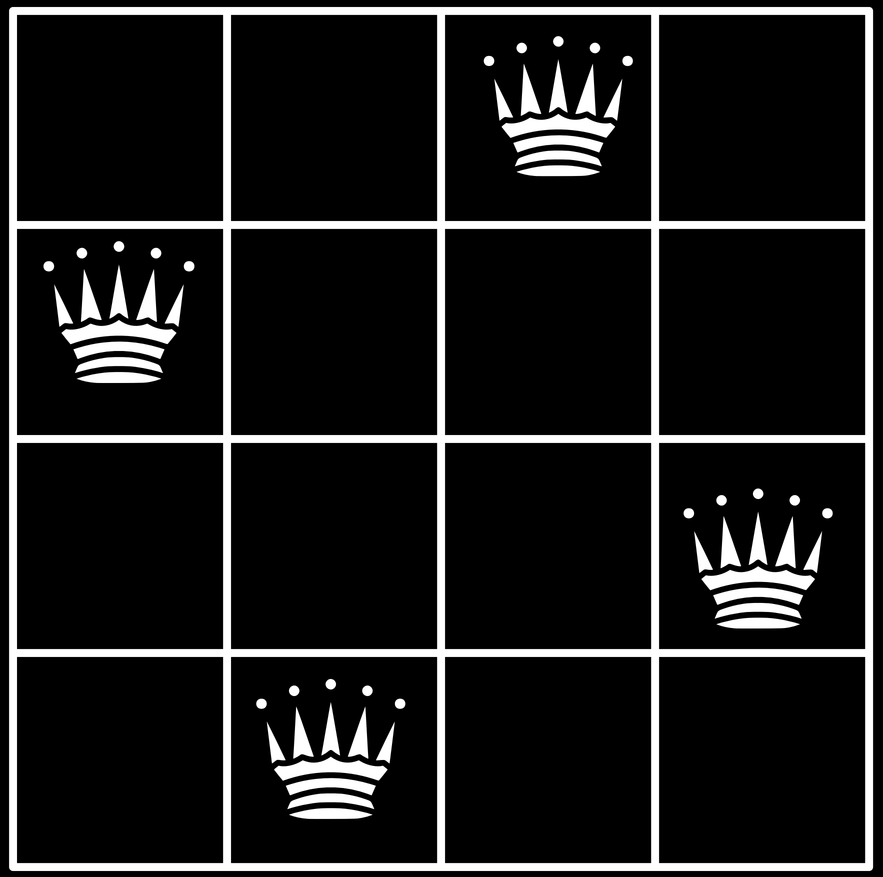
How it works:

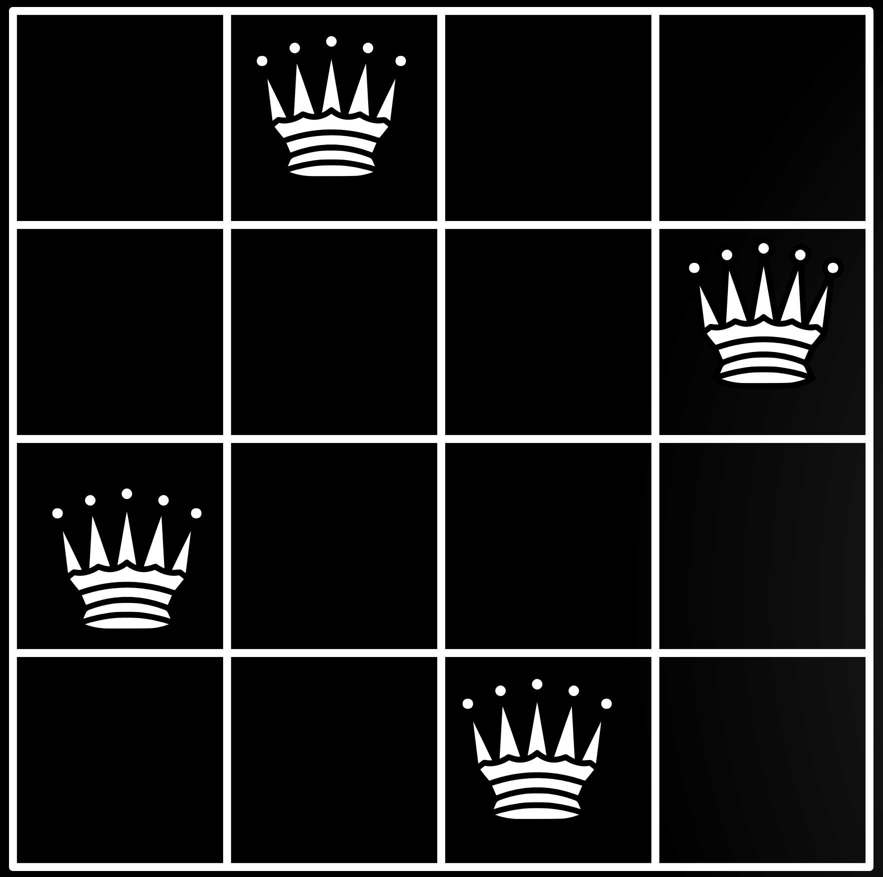
- It initializes the board with -1 (indicating no queens placed).
- It initializes a solutions vector to store all found solutions.
- It calls solveNQueensUtil to start the backtracking algorithm.
- After the algorithm finishes, it checks if any solutions were found and prints them.

EXAMPLE

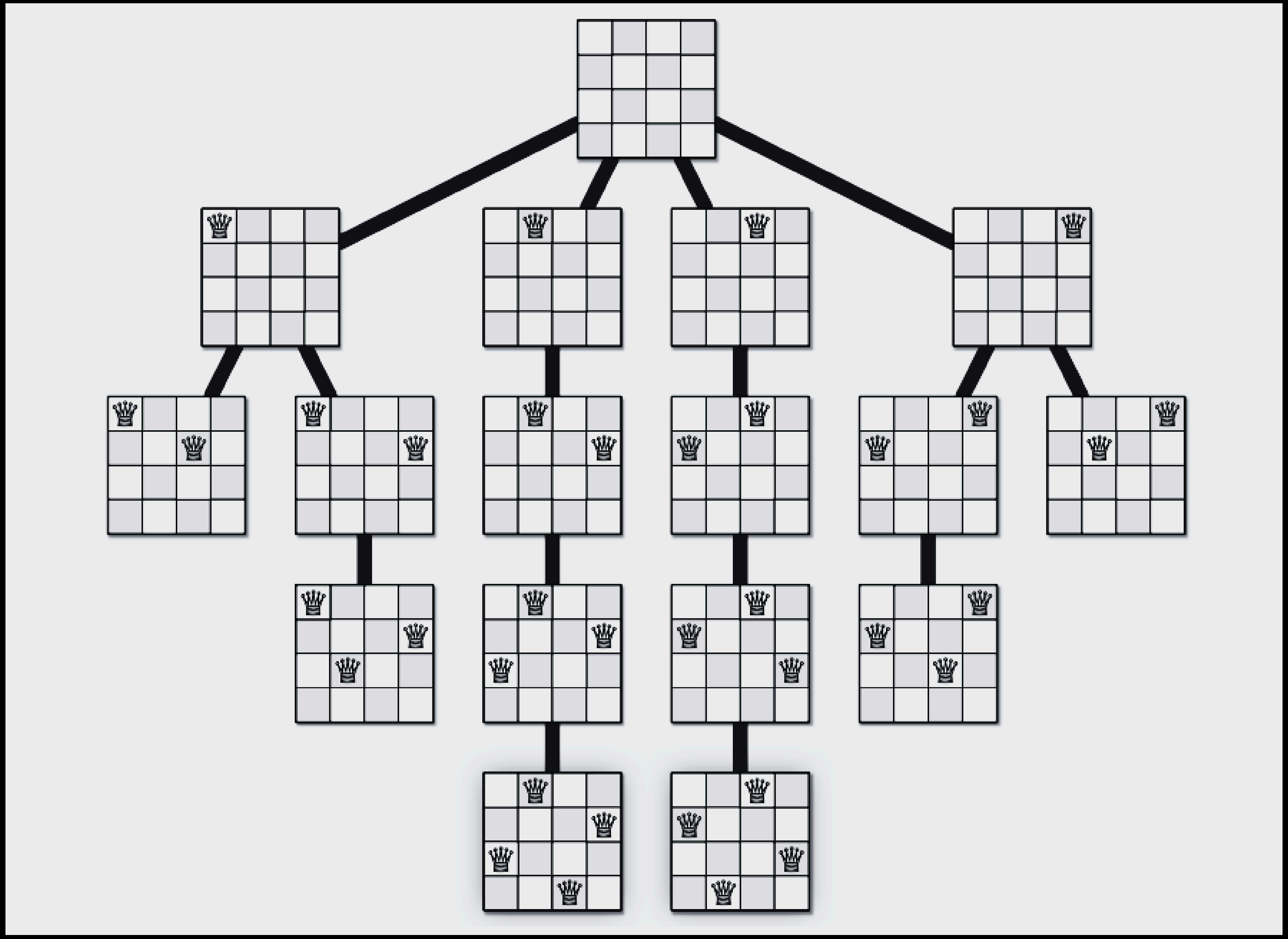
FOR N = 4

2 solutions





VISUALIZATION



GITHUB LINK HERE

N_Queen_viz

THANK YOU

TECH COMPANY