8 Queens problem

Introduction:https://colab.research.google.com/drive/1eNlJBu1UDFIjLPwV1oufSx-y7ezuwuA1?usp=drive_link

- N-Queens problem using a **Breadth-First Search (BFS)** approach. The N-Queens problem is a classic combinatorial problem that requires placing n queens on an **n×n** chessboard so that no two queens threaten each other. The program finds and displays a valid solution, if one exists.

A. Function: is_safe(board, row, col)

- This function checks if it is safe to place a queen at the position (row, col) on the chessboard. It ensures that there are no conflicts with other queens placed so far

1. **Input Parameters**:
   - **board:** The current state of the chessboard.
   - **row:** The row index where the queen is being considered.
   - **col:** The column index where the queen is being considered.
2. **Key Checks**:
   - **Column Check**: Verifies that no queen exists in the same column above the current row.
   - **Upper Left Diagonal Check**: Ensures no queen exists diagonally above to the left.
   - **Upper Right Diagonal Check**: Ensures no queen exists diagonally above to the right.
3. **Output**:
   - Returns True if it is safe to place the queen; otherwise, False.

B. Function: solve_n_queens(n)

- This function attempts to solve the N-Queens problem using a BFS strategy.

1. **Input Parameter:**
   - n: The size of the chessboard (n×n) and the number of queens to place.
2. **Initialization**:
   - Creates an empty chessboard represented as a 2D list with all entries set to 0 (no queens placed initially).
   - Initializes a queue (deque) for BFS with the starting state (row = 0, board = initial_empty_board).
3. **Breadth-First Search (BFS)**:
   - While the queue is not empty, the algorithm processes the current state (current row and board configuration).
   - If queens have been successfully placed in all rows, the function returns the current board as the solution.
   - For the current row, it tries placing a queen in every column. If a position is deemed safe (**is_safe**), a new board state is created, and the next row and board state are added to the queue.
4. **Return Value**:
   - **If a solution is found**, the function returns the chessboard configuration.
   - **If no solution** exists after exploring all possibilities, it returns None.

8 Queens problem

C. Function: print_board(board)

- This function formats and displays the chessboard

1. **Input Parameter**:
   - board: The n×n chessboard containing 1 (queen) and 0 (empty spaces).
2. **Functionality**:
   - Converts each row into a string where:
   - 'Q' represents a queen (1).
   - '.' represents an empty space (0).
   - Prints the formatted board row by row.

1. Main Execution (if __name__ == "__main__":)

- This block serves as the entry point for the program.
- **Steps**:
  - Sets n=8n = 8n=8 for the 8-Queens problem.
  - Calls solve_n_queens(n) to solve the problem.
  - If a solution is found, it prints "Solution:" followed by the chessboard using print_board.
  - If no solution exists, it prints "No solution found.

Queens Positions:

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	-	-	-	-	-	-	-	Q
(1,0)	-	-	-	Q	-	-	-	-
(2,0)	Q	-	-	-	-	-	-	-
(3,0)	-	-	Q	-	-	-	-	-
(4,0)	-	-	-	-	Q	-	-	-
(5,0)	-	Q	-	-	-	-	-	-
(6,0)	-	-	-	-	-	-	Q	-
(7,0)	-	-	-	-	-	Q	-	-

- Q1 = (0,7)
- Q2 = (1,3)
- Q3 = (2,0)
- Q4 = (3,2)
- Q5 = (4,5)
- Q6 = (5,1)
- Q7 = (6,6)
- Q8 = (7,4)

8 Queens problem

How It Works

1. **Board Representation**:
   - The board is represented as a 2D list, where `0` indicates an empty cell and `1` indicates a queen's position.
2. **Conflict Checks**:
   - Using `is_safe`, the program ensures queens are not placed in the same column, row, or diagonals.
3. **Search Strategy**:
   - The BFS approach systematically explores all possible queen placements row by row. Unlike Depth-First Search (DFS), BFS ensures all options at the current level (row) are explored before moving deeper.
4. **Termination**:
   - If the queue is empty and no solution is found, the function returns `None`.
   - If a valid board configuration is reached (all rows have queens), the function returns the board.

DLS

Introduction:

https://colab.research.google.com/drive/1WPeEVWl2lugzDBxBCJQPazK_MIdEf2Uj?usp=drive_link

- The goal of the N-Queens problem is to place NNN queens on an N×NN \times NN×N chessboard while ensuring:
  - No two queens are in the same row.
  - No two queens are in the same column.
  - No two queens are on the same diagonal.
  - This implementation employs a DLS approach to systematically explore potential solutions up to a specified depth.

A. . Function: is_safe(board, row, col)

- This function checks whether placing a queen at position (row, col) is safe.
- How It Works:

1. **Column Check**:
   - Iterates through all rows above the current row to ensure no queen exists in the same column.
2. **Upper-Left Diagonal Check**:
   - Checks the diagonal above and to the left of (row, col) using synchronized iteration with zip.
3. **Upper-Right Diagonal Check**:
   - Checks the diagonal above and to the right of (row, col) in a similar fashion.

8 Queens problem

Output:

- Returns True if the position is safe for a queen.
- Returns False if placing a queen at (row, col) would result in a conflict

B. Function: solve_n_queens_dls(board, row, col, limit)

- This is the core function that solves the N-Queens problem using Depth-Limited Search.
- **Parameters:**
  - **board**: The $N \times N$ chessboard, represented as a 2D list.
  - **row:** The current row to place a queen.
  - **col:** The current column to consider for placement.
  - **limit:** The maximum depth the search can explore.

Algorithm:

- Base Case - All Queens Placed:
- If row == len(board), all queens have been successfully placed, and the function returns True.
  Base Case - Exceed Column or Depth Limit:
- If col >= len(board) or limit == 0, no valid placement is found in the current branch, and the function returns False.

Recursive Search:

- **Safe Placement:**
  - If is_safe(board, row, col) evaluates to True, place a queen at (row, col) (set board[row][col] = 1).
  - Recursively call solve_n_queens_dls for the next row (row + 1) with the depth limit decremented (limit - 1).
  - If successful, return True. Otherwise, backtrack by removing the queen (set board[row][col] = 0).
- **Try Next Column:**
- If placing a queen at (row, col) is not safe or fails, recursively attempt placement in the next column (col + 1).

Output:

- Returns True if a valid configuration is found.
- Returns False if no valid configuration exists within the depth limit.

.

8 Queens problem

C. Function: print_board(board)

- This function formats and prints the chessboard.
- **Input:**
  - **board:** A 2D list where 1 represents a queen and 0 represents an empty space.
- **Output:**
  - Displays each row of the chessboard as a string: **'Q' for queens**. **'.' for empty spaces.**

Function: main()

- The main driver function initializes the board and invokes the solver.
- Steps:
  - **Define N**=8 for the standard 8-Queens problem.
  - **Create an N×N** chessboard initialized with 0s (empty spaces).
  - Set the initial depth limit to N (equal to the number of queens to be placed).
  - Call solve_n_queens_dls with the initial parameters:
  - Start at row = 0, col = 0, and the specified depth limit.
  - **If a solution is found**:
    - Print "Solution:" and display the board using print_board.
  - **If no solution is found within the depth limit:**
    - Print "No solution found

Queens Positions:

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	Q	-	-	-	-	-	-	-
(1,0)	-	-	-	-	Q	-	-	-
(2,0)	-	-	-	-	-	-	-	Q
(3,0)	-	-	-	-	-	Q	-	-
(4,0)	-	-	Q	-	-	-	-	-
(5,0)	-	-	-	-	-	-	Q	-
(6,0)	-	Q	-	-	-	-	-	-
(7,0)	-	-	-	Q	-	-	-	-

- Q1 = (0,0)
- Q2 = (1,4)
- Q3 = (2,7)
- Q4 = (3,5)
- Q5 = (4,2)
- Q6 = (5,6)
- Q7 = (6,1)
- Q8 = (7,3)

8 Queens problem

D. How Depth-Limited Search (DLS) Works

- DLS is a variation of depth-first search (DFS) that restricts exploration to a predefined depth. In this program:
  - Each level of the search tree corresponds to placing a queen in a specific row.
  - The depth limit ensures that the search does not exceed the required number of queens (N)
- By limiting the depth, the algorithm avoids unnecessary exploration, making it more focused and efficient for this problem.

Iterative Deepening Search (IDS)

Introduction:https://colab.research.google.com/drive/1aJh0omFUO2ADBTiUn7Ol9GpVkpgN48h3?usp=drive_link

- The N-Queens problem is a classic combinatorial problem.
- The objective is to place N queens on an N×N chessboard such that:
  - No two queens threaten each other in the same row.
  - No two queens threaten each other in the same column.
  - No two queens threaten each other along the same diagonal.

This implementation leverages IDS to systematically explore possible configurations up to increasing depth limits until solutions are found.

A. Function: is_safe(board, row, col)

- **Purpose:**
  - Determines if placing a queen at position (row, col) is safe.
- **Input:**
  - **board:** A list representing the current state of the board, where board[i] is the column index of the queen in row i, or -1 if no queen is placed.
  - **row:** The row where a queen is to be placed.
  - **col:** The column where a queen is to be placed.
- **Algorithm:**
  - Vertical Attack Check:
    - Iterates through previous rows to check if a queen exists in the same column.
  - Diagonal Attack Check:
    - Calculates the absolute difference between column indices (|board[i] - col|) and row indices (|i - row|).
    - If these values are equal, a diagonal conflict exists.

Output:

- Returns True if the position (row, col) is safe.
- Returns False if it results in a conflict.

B. Function: ids_find_solutions(n)

- **Purpose:**
  - Finds all possible solutions to the N-Queens problem using Iterative Deepening Search (IDS).
- **Input:**
  - **n:** The size of the board and the number of queens to place.
- **Key Components:**
  - **Nested Function:** dfs(board, row, depth_limit)
    - Implements depth-limited DFS to explore queen placements.
- **Parameters:**
  - **board:** Current board state represented as a list.
  - **row:** The current row to process.
  - **depth_limit:** Maximum depth to explore.
- **Algorithm:**
  - If row == n, all queens are placed, and the current board state is appended to the solutions list.
  - If row >= depth_limit, the function stops exploration (depth limit reached).

For each column in the current row:

- If is_safe(board, row, col) is True, place a queen (board[row] = col) and recursively call dfs for the next row.
  - After recursion, backtrack by removing the queen (board[row] = -1).

Iterative Deepening:

- **Output:**
  - Returns a list of solutions,
    - where each solution is a list of column indices representing queen placements.

C. Displaying a Solution

- After finding all solutions, the program allows displaying a specific solution (e.g., the second or third one). It iterates through each row of the solution and prints:
  - "Q" for the column where a queen is placed.
  - "." for empty columns.

.

8 Queens problem

(IDDFS)

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	-	-	-	-	-	-	-	Q
(1,0)	-	Q	-	-	-	-	-	-
(2,0)	-	-	-	Q	-	-	-	-
(3,0)	Q	-	-	-	-	-	-	-
(4,0)	-	-	-	-	-	-	Q	-
(5,0)	-	-	-	-	Q	-	-	-
(6,0)	-	-	Q	-	-	-	-	-
(7,0)	-	-	-	-	-	Q	-	-

Queens Positions:

- Q1 = (0,7)
- Q2 = (1,1)
- Q3 = (2,3)
- Q4 = (3,0)
- Q5 = (4,6)
- Q6 = (5,4)
- Q7 = (6,2)
- Q8 = (7,5)

D. Steps in This Implementation:

- Start with a depth limit of 0.
- Run a depth-limited DFS to explore queen placements up to the depth limit.
- Increment the depth limit and repeat until at least one solution is found

Bidirectional Backtracking

Introduction:https://colab.research.google.com/drive/1B_BI1LiMcED3dQJI22tdL6gC2faxQ8nU?usp=drive_link

- The N-Queens problem is a classic combinatorial problem.
- The objective is to place N queens on an N×N chessboard such that:
  - No two queens threaten each other in the same row.
  - No two queens threaten each other in the same column.

his implementation uses bidirectional backtracking to divide the problem into two halves, reducing the search space by simultaneously placing queens from the top and bottom of the board

8 Queens problem

A. Function: is_safe(board, row, col)

- **Purpose:**
  - Checks if placing a queen at position (row, col) is safe, considering previously placed queens.
- **Input:**
  - **board:** A list representing the current board state. Each index represents a row, and the value at each index represents the column where the queen is placed. A value of -1 indicates no queen is placed in that row.
  - **row:** The row being evaluated.
  - **col:** The column being evaluated.
- **Algorithm:**
  - Iterate through all rows up to the current row.
  - For each previous queen:
  - **Column Check:** Ensure no queen exists in the same column.
  - **Left Diagonal Check**: Ensure no queen exists on the same left diagonal (board[r] - r == col - row).
  - **Right Diagonal Check**: Ensure no queen exists on the same right diagonal (board[r] + r == col + row).
- **Output:**
  - **Returns True if** (row, col) is safe.
  - **Returns False if** conflicts exist.

B. Function: solve_8_queen_bidirectional()

- **Purpose:**
  - Attempts to solve the 8-Queens problem using bidirectional backtracking.

- **Initialization:**
  - Set n = 8 for an 8x8 chessboard.
  - Create a board list initialized with -1 to represent an empty board.
- **Nested Function: backtrack(row_top, row_bottom):**
  - Recursively places queens starting from the top and bottom rows, moving toward the center.
- **Parameters:**
  - **row_top**: Current row being processed from the top half of the board.
  - **row_bottom**: Current row being processed from the bottom half of the board.
  - Algorithm:
  - **Base Case: If** row_top > row_bottom, all queens have been successfully placed.
  - Return True to indicate success.
  - For each column in the top row:
    - Check if placing a queen at (row_top, col_top) is safe.

- Temporarily place the queen by updating board[row_top] = col_top.
- For each column in the bottom row:
- Check if placing a queen at (row_bottom, col_bottom) is safe.
- Temporarily place the queen by updating board[row_bottom] = col_bottom.
- Recursively call backtrack(row_top + 1, row_bottom - 1) to process the next rows.
- If the recursive call returns True, a solution is found.
- Backtrack by resetting board[row_bottom] = -1.
- Backtrack by resetting board[row_top] = -1.

Return False if no valid placement is found.

- **Output:**

➢ **If a solution is found**, call print_solution(board) to display it.
Otherwise, print "No solution found."

C. Function: print_solution(board)

- **Purpose:**

➢ Displays the board in a readable format where:

- **"Q"** represents a queen.
- **"."** represents an empty space.

- **Input**:

➢ **board:** The final configuration of the board where queens are successfully placed.

- **Algorithm**:

➢ Iterate through each row.
➢ For each column in the row**, print "Q"** if a queen is present and "." otherwise.

8 Queens problem

Bidirectional Backtracking

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	Q	-	-	-	-	-	-	-
(1,0)	-	-	-	-	Q	-	-	-
(2,0)	-	-	-	-	-	-	-	Q
(3,0)	-	-	-	-	-	Q	-	-
(4,0)	-	-	Q	-	-	-	-	-
(5,0)	-	-	-	-	-	-	Q	-
(6,0)	-	Q	-	-	-		-	-
(7,0)	-	-	-	Q	-	-	-	-

Queens Positions:

- Q1 = (0,0)
- Q2 = (1,4)
- Q3 = (2,7)
- Q4 = (3,5)
- Q5 = (4,2)
- Q6 = (5,6)
- Q7 = (6,1)
- Q8 = (7,3)

Alpha- Beta pruning

Introduction:https://colab.research.google.com/drive/1rJqzzroBF4aLPEDgWBaeXztkV_bUW1WR?usp=drive_link

- The N-Queens problem is a classic combinatorial problem.
- The objective is to place N queens on an N×N chessboard such that:
  - No two queens threaten each other in the same row.

No two queens threaten each other in the same column

A. Method: is_safe(self, row, col)

- **Purpose:**
  - Checks if placing a queen at position (row, col) is safe, based on the queens already placed.

8 Queens problem

- **Input:**
  - **row:** The row to evaluate.
  - **col**: The column to evaluate.
- **Algorithm:**
  - Iterate through all rows up to the current row.
  - For each previously placed queen, verify:
    - **Column Check**: Ensure no queen is in the same column (self.board[i] == col).
    - **Left Diagonal Check**: Ensure no queen is on the left diagonal (self.board[i] - i == col - row).
    - **Right Diagonal Check**: Ensure no queen is on the right diagonal (self.board[i] + i == col + row).
- **Output:**
  - **Returns True** if (row, col) is safe.
  - **Returns False** if conflicts exist.

B. Method: place_queen(self, row)

- **Purpose:**
  - Recursively places queens on the board using backtracking.
- **Input:**
  - **row:** The current row to process.
- **Algorithm:**
  - **Base Case**: If `row == 8` (all queens are placed), return `True`.
  - Iterate through all columns in the current row.
  - For each column:
    - Use `is_safe(row, col)` to check if placing a queen at `(row, col)` is valid.
    - If valid, place the queen by setting `self.board[row] = col`.
    - Recursively call `place_queen(row + 1)` to process the next row.
    - If the recursive call returns `True`, a valid solution is found.
    - Otherwise, backtrack by resetting `self.board[row] = -1`.
  - If no valid column is found for the current row, **return `False`**

8 Queens problem

C. Method: print_board(self)

- **Purpose:**

  - Prints the chessboard in a readable format.

- **Algorithm:**

  - Iterate through each row.
  - For each column in the row, print:
    - "Q" if the queen is present.
    - "." for an empty space.

D. Function: solve_eight_queens()

- **Purpose:**

  - Creates an instance of the EightQueens class, solves the problem, and prints the result.

- **Algorithm:**

  - Instantiate the EightQueens class.
  - Call place_queen(0) to start placing queens from the first row.
  - If a solution is found, call print_board() to display the solution.

  - If no solution is found, print "No solution found.".

- Q1 = (0,0)
- Q2 = (1,4)
- Q3 = (2,7)
- Q4 = (3,5)
- Q5 = (4,2)
- Q6 = (5,6)
- Q7 = (6,1)
- Q8 = (7,3)

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	Q	-	-	-	-	-	-	-
(1,0)	-	-	-	-	Q	-	-	-
(2,0)	-	-	-	-	-	-	-	Q
(3,0)	-	-	-	-	-	Q	-	-
(4,0)	-	-	Q	-	-	-	-	-
(5,0)	-	-	-	-	-	-	Q	-
(6,0)	-	Q	-	-	-	-	-	-
(7,0)	-	-	-	Q	-	-	-	-

8 Queens problem

Min and Max Solutions

Introduction:

https://colab.research.google.com/drive/1hUEWCGSZemlcGxkfNMq5eIUmlPn4k3gH?usp=drive_link

- The N-Queens problem is a classic combinatorial problem.
- The objective is to place N queens on an N×N chessboard such that:
  - No two queens threaten each other in the same row.
  - No two queens threaten each other in the same column.

A. Function: is_safe(board, row, col, n)

- **Purpose:**
  - This function checks if placing a queen at (row, col) is safe based on the current board configuration.
- **Logic**:
  - **Column Check**: Ensures no queen is placed in the same column (board[i] == col).
  - **Diagonal Check**: Ensures no queen is placed on the same diagonal (board[i] - i == col - row or board[i] + i == col + row).
- **Returns**: True if it's safe to place the queen; False otherwise.

B. Function: solve_eight_queens_min(n, row=0, board=None)

- **Purpose**:

This function finds the **first valid solution** (min solution) by using backtracking. It stops after the first solution is found.

- **Input**:
  - **n:** The size of the board (in this case, 8).
  - **row:** The current row being processed.
  - **board**: A list of size n where each element represents the column of the queen placed in the corresponding row. A value of -1 indicates no queen placed in that row.
- **Logic**:
  - **Base Case**: If all queens are placed (i.e., row == n), print the board and return True.
  - **Column Check**: For each column in the current row, check if it's safe to place a queen using is_safe().
  - **Recursion**: If placing a queen is safe, place the queen and recursively call solve_eight_queens_min() for the next row.
  - **Backtracking**: If no valid placement is found, backtrack by resetting the queen position.

- **Returns**:
  - The function **returns True** to stop further searching after the first solution is found.

C. Function: solve_eight_queens_max(n, row=0, board=None)

- **Purpose**: This function finds **all possible solutions** (max solutions) by continuing the search after finding each valid solution.
- **Input**: Similar to solve_eight_queens_min(), but it continues searching for more solutions instead of stopping after the first one.
- **Logic**:
  - **Base Case**: When all queens are placed, print the board and continue to find additional solutions.
  - **Column Check**: Similar to solve_eight_queens_min(), it iterates over all columns to find safe placements.
  - **Backtracking**: Resets the queen position when no valid solution is found in a row.
- **Returns**: The function returns False after printing each solution to continue searching for more solutions.

D. function: print_board(board, n)

- **Purpose**: This function prints the chessboard configuration in a readable format.
- **Logic**:
  - For each row, it prints "Q" for a queen and "." for an empty space. It constructs the output for each row by checking the queen's column for that row.

Algorithm Walkthrough

For solve_eight_queens_min() (First Solution):

1. Start at row = 0 and try placing a queen in each column (0 to 7).
2. For each column, check if placing a queen is safe using is_safe().
3. If safe, place the queen and move to the next row recursively (row + 1).
4. If a solution is found (all queens placed), print the board and return True to stop.
5. If no valid placement is found, backtrack by resetting the queen's position.

For solve_eight_queens_max() (All Solutions):

1. Similar to solve_eight_queens_min(), but it continues searching for all solutions.
2. After finding a solution, print the board and return False to continue finding other solutions.
3. The search continues until all possible configurations have been explored

8 Queens problem

- Q1 = (0,0)
- Q2 = (1,4)
- Q3 = (2,7)
- Q4 = (3,5)
- Q5 = (4,2)
- Q6 = (5,6)
- Q7 = (6,1)
- Q8 = (7,3)

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	Q	-	-	-	-	-	-	-
(1,0)	-	-	-	-	Q	-	-	-
(2,0)	-	-	-	-	-	-	-	Q
(3,0)	-	-	-	-	-	Q	-	-
(4,0)	-	-	Q	-	-	-	-	-
(5,0)	-	-	-	-	-	-	Q	-
(6,0)	-	Q	-	-	-	-	-	-
(7,0)	-	-	-	Q	-	-	-	-

Hill climbing

Introduction: https://colab.research.google.com/drive/1ca_KvfLVCnfE-qy_XYZPIqB1fYEt_gtJ?usp=drive_link

- The N-Queens problem is a classic combinatorial problem.
- The objective is to place N queens on an N×N chessboard such that:
  - No two queens threaten each other in the same row.
  - No two queens threaten each other in the same column.

A. Function: print_board(board)

- **Purpose**: This function prints the current state of the chessboard in a human-readable format.
- **Logic**:
  - For each row, it prints "Q" where a queen is placed and "." for an empty space.
  - After printing all rows, it prints an additional newline for readability.

B. Function: is_safe(board, row, col)

- **Purpose**: This function checks whether placing a queen at (row, col) is safe.
- **Logic**:
  - **Column Check**: Ensures no other queen is placed in the same column by checking the upper part of the current column.
  - **Left Diagonal Check**: Ensures no other queen is placed on the same diagonal by checking all positions to the upper left.
  - **Right Diagonal Check**: Ensures no other queen is placed on the same diagonal by checking all positions to the upper right.
- **Returns**: True if the position is safe for placing a queen; False otherwise.

C. Function: solve_n_queens_util(board, row)

- **Purpose**: This is the recursive backtracking function that attempts to place queens row by row.
- **Logic**:
  - **Base Case**: If row is greater than or equal to the board size, all queens have been placed, so the current board configuration is printed.
  - **For Each Column**: The function iterates over all columns in the current row and tries to place a queen. If placing a queen is safe, it proceeds recursively to place the queen in the next row.
  - **Backtracking**: If no valid position is found in the current row, the function backtracks by removing the queen and trying the next position.

D. function: solve_n_queens(n)

- **Purpose**: This is the main function that sets up the chessboard and calls the recursive function to solve the problem.
- **Logic**:
  - Initializes an empty n x n chessboard with all positions set to '.' (empty).
  - Calls the solve_n_queens_util() function to begin placing queens from row 0.

Algorithm Walkthrough

1. **Initialization**: The board is initialized as an n x n grid filled with ".", representing empty spaces.
2. **Recursive Backtracking**:
   - Starting from row 0, the algorithm attempts to place a queen in each column of the current row.
   - For each potential column, it checks if placing a queen is safe using the is_safe() function.
   - If a safe position is found, the queen is placed, and the function proceeds to place a queen in the next row recursively.
   - If the queen cannot be placed in any column of the current row, the function backtracks by removing the queen and trying the next column in the previous row.
3. **Solution Found**: Once all queens are successfully placed (i.e., the function reaches row n), the current board configuration is printed.
4. **Multiple Solutions**: The backtracking algorithm explores all possible configurations. If one solution is found, the function continues exploring other possible placements.

8 Queens problem

- Q1 = (0,0)
- Q2 = (1,4)
- Q3 = (2,7)
- Q4 = (3,5)
- Q5 = (4,2)
- Q6 = (5,6)
- Q7 = (6,1)
- Q8 = (7,3)

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,0)	Q	-	-	-	-	-	-	-
(1,0)	-	-	-	-	Q	-	-	-
(2,0)	-	-	-	-	-	-	-	Q
(3,0)	-	-	-	-	-	Q	-	-
(4,0)	-	-	Q	-	-	-	-	-
(5,0)	-	-	-	-	-	-	Q	-
(6,0)	-	Q	-	-	-	-	-	-
(7,0)	-	-	-	Q	-	-	-	-

Genetic Algorithms

Introduction:https://colab.research.google.com/drive/1lQ0MGVDnzkWZN8Lgp1Hj5qrjclMQftkx?usp=drive_link

A. Algorithm Overview

- The genetic algorithm operates as follows:

1. **Population Initialization**: A population of random solutions is generated, where each solution represents a valid arrangement of queens on the chessboard.
2. **Fitness Evaluation**: Each individual in the population is evaluated based on its "fitness", which is determined by the number of conflicts (penalties) between queens.
3. **Selection**: Individuals with lower penalties (better fitness) are selected probabilistically to act as parents for the next generation.
4. **Crossover**: Parents are crossed over to generate offspring, combining traits from both parents to create new solutions.
5. **Mutation**: Random mutations are introduced in offspring to maintain genetic diversity and potentially escape local minima.
6. **Termination**: The algorithm terminates when a solution with no conflicts is found, or after a specified number of generations.

B. Init_pop(pop_size)

- **Purpose**: Initializes the population with random solutions.
- **Logic**:
  - A solution is represented as an array of 8 integers, each representing the column position of a queen in the respective row.
  - The population is initialized using `np.random.randint(8, size=(pop_size, 8))`, which generates `pop_size` random solutions where each queen is placed at a random column in each row.

8 Queens problem

C. calc_fitness(population)

- **Purpose**: Calculates the fitness of each individual in the population.
- **Logic**:
  - The fitness is determined by counting the number of conflicts (penalties) between queens. Conflicts can arise if two queens are in the same column or share a diagonal.
  - For each queen, the algorithm checks whether any other queen in the solution shares the same column or diagonal. The fitness score is the negative of the penalty, with lower penalties indicating better solutions.

D. selection(population, fitness_vals)

- **Purpose**: Selects individuals for the next generation based on their fitness.
- **Logic**:
  - The selection process uses a probabilistic method where individuals with better fitness (lower penalties) have a higher chance of being selected.
  - The probabilities are normalized and used to randomly select individuals to form a new population.

E. crossover(parent1, parent2, pc)

- **Purpose**: Performs crossover between two parent solutions to create two offspring.
- **Logic**:
  - If a random number is less than the crossover probability (pc), the algorithm selects a random crossover point and combines portions of both parents to form two offspring.
  - If no crossover occurs, the children are just copies of the parents

F. mutation(child, pm)

- **Purpose**: Introduces random mutations in the offspring.
- **Logic**:
  - The mutation involves changing the column position of a randomly selected queen in the solution with a probability (pm). This introduces genetic diversity into the population and helps the algorithm explore the solution space.

G. crossover_mutation(population, pc, pm)

- **Purpose**: Applies both crossover and mutation to the entire population.
- **Logic**:
  - This function iterates over pairs of individuals, applies crossover to generate children, and then applies mutation to each child.
  - The new population generated by this function replaces the old one, and the evolutionary process continues.

H. is_solution_found(population, fitness_vals)

- **Purpose**: Checks whether any solution in the population has zero penalties, indicating a valid solution.
- **Logic**:
  - If any individual has a fitness of zero, the solution is considered found, and that individual is returned.

I. Genetic_algorithm(pop_size, pc, pm, max_generations)

- **Purpose**: The main genetic algorithm loop.
- **Logic**:
  - Initializes the population, calculates fitness, and repeatedly evolves the population through selection, crossover, and mutation.
  - If a valid solution (fitness = 0) is found, the algorithm terminates and returns the solution.
  - If no solution is found after a specified number of generations (max_generations), the algorithm terminates with no solution.

Algorithm Parameters

The algorithm has the following configurable parameters:

- pop_size (**Population size**): The number of individuals in each generation.
- pc (**Crossover probability**): The probability that crossover occurs between two parents.
- pm (**Mutation probability**): The probability that mutation occurs in the offspring.
- **max_generations:** The maximum number of generations before the algorithm terminates.