N-Queens Backtracking and Branch-and-Bound Algorithms

def print_board(board): for row in board: print(" ".join("Q" if col else "." for col in row)) print("\n")

def is_safe(board, row, col, n): # Check the current column for i in range(row): if board[i][col]: return False for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j]: return False for i, j in zip(range(row, -1, -1), range(col, n)): if board[i][j]: return False return True

def solve_n_queens_backtracking(board, row, n): if row == n: # If all queens are placed, print the solution print_board(board) return True

```
found_solution = False
for col in range(n):
    if is_safe(board, row, col, n):
        board[row][col] = True  # Place the queen

        # Recur to place the rest of the queens
        found_solution = solve_n_queens_backtracking(board,
row + 1, n) or found_solution

        # If placing the queen at (row, col) didn't lead to a
solution, backtrack
        board[row][col] = False
return found_solution
```

def n_queens_backtracking(n): board = [[False] * n for _ in range(n)] # Initialize n x n board with False (no queens placed) if not solve_n_queens_backtracking(board, 0, n): print("No solution exists.")

def is_safe_branch_and_bound(row, col, cols, diags1, diags2, n): return not (cols[col] or diags1[row + col] or diags2[row - col + (n - 1)]) def solve_n_queens_branch_and_bound(row, n, cols, diags1, diags2, board): if row == n: # If all queens are placed, print the solution print_board(board) return True found_solution = False

```
for col in range(n):
    if is_safe_branch_and_bound(row, col, cols, diags1,
diags2, n):
        # Place the queen
        board[row][col] = True
        cols[col] = True
        diags1[row + col] = True
        diags2[row - col + (n - 1)] = True

        # Recur to place the rest of the queens
        found_solution = solve_n_queens_branch_and_bound(row
+ 1, n, cols, diags1, diags2, board) or found_solution
        board[row][col] = False
        cols[col] = False
        diags1[row + col] = False
        diags2[row - col + (n - 1)] = False
return found_solution
```

def n_queens_branch_and_bound(n): board = [[False] * n for _ in range(n)] # Initialize n x n board with False (no queens placed) cols = [False] * n diags1 = [False] * (2 * n - 1) # For "/" diagonals diags2 = [False] * (2 * n - 1) # For "" diagonals if not solve_n_queens_branch_and_bound(0, n, cols, diags1, diags2, board): print("No solution exists.")

n_queens_backtracking(4)

n_queens_branch_and_bound(4)

explain the above code line by line with complete details why , what , how , etc and also give complete theory about important topics in this, also provide visual explanations

I'll explain the N-Queens problem code in detail, covering both the backtracking and branch-and-bound solutions, and provide the theory behind these approaches.

N-Queens Problem Explanation

The N-Queens problem is a classic chess puzzle: place N queens on an N×N chessboard so that no two queens threaten each other. In chess, a queen can attack any piece in the same row, column, or diagonal.

Code Breakdown

Function: `print_board(board)`

```
def print_board(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")
```

This function visualizes the chessboard:

- It iterates through each row of the board
- For each cell, it prints "Q" if there's a queen (True value) or "." if empty (False value)
- The `" ".join()` connects these characters with spaces to create a row
- After printing all rows, it adds two newlines for spacing between solutions

Backtracking Solution Functions

Function: `is_safe(board, row, col, n)`

```python
def is_safe(board, row, col, n):
    # Check the current column
    for i in range(row):
        if board[i][col]:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j]:
            return False
    return True
```

This function checks if placing a queen at position (row, col) is safe:

- First loop: Checks if there's any queen in the same column above the current position
- Second loop: Checks the upper-left diagonal (decreasing both row and column indices)
- Third loop: Checks the upper-right diagonal (decreasing row while increasing column)
- Returns True only if no queen threatens this position

We don't need to check the row because the algorithm places only one queen per row.

Function: `` `solve_n_queens_backtracking(board, row, n)` ``

```python
def solve_n_queens_backtracking(board, row, n):
    if row == n:
        # If all queens are placed, print the solution
        print_board(board)
        return True
```

```
    found_solution = False
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = True  # Place the queen

            # Recur to place the rest of the queens
            found_solution = solve_n_queens_backtracking(board, row + 1, n) or found_solution

            # If placing the queen at (row, col) didn't lead to a solution, backtrack
            board[row][col] = False
    return found_solution
```

This is the core recursive function:

- Base case: If row == n, we've successfully placed all n queens, so print the solution and return True
- For each column in the current row, it:
    1. Checks if placing a queen is safe
    2. If safe, places the queen by setting board[row][col] = True
    3. Recursively tries to place queens in subsequent rows
    4. Uses the "or found_solution" to track if any solution was found
    5. Backtracks by removing the queen (setting False) if this path doesn't work
- Returns whether any solution was found

Function: `` `n_queens_backtracking(n)` ``

```
def n_queens_backtracking(n):
    board = [[False] * n for _ in range(n)]  # Initialize n x n board with False (no queens placed)
```

```python
    if not solve_n_queens_backtracking(board, 0, n):
        print("No solution exists.")
```

This function:

- Creates an empty n×n board (initialized with False values)
- Calls the recursive function starting from row 0
- Prints a message if no solution exists

Branch and Bound Solution Functions

Function: `is_safe_branch_and_bound(row, col, cols, diags1, diags2, n)`

```python
def is_safe_branch_and_bound(row, col, cols, diags1, diags2, n):
    return not (cols[col] or diags1[row + col] or diags2[row -
col + (n - 1)])
```

This improved safety check function:

- Uses pre-computed arrays to track occupied columns and diagonals
- `cols[col]`: True if column is occupied
- `diags1[row + col]`: True if "/" diagonal is occupied (sum of row+col is constant)
- `diags2[row - col + (n-1)]`: True if "" diagonal is occupied (difference of row-col is constant)
- The offset `(n-1)` ensures indices are non-negative for the second diagonal

Function: `solve_n_queens_branch_and_bound(row, n, cols, diags1, diags2, board)`

```python
def solve_n_queens_branch_and_bound(row, n, cols, diags1, diags2,
board):
    if row == n:
        # If all queens are placed, print the solution
```

```
        print_board(board)
        return True
    found_solution = False

    for col in range(n):
        if is_safe_branch_and_bound(row, col, cols, diags1,
diags2, n):
            # Place the queen
            board[row][col] = True
            cols[col] = True
            diags1[row + col] = True
            diags2[row - col + (n - 1)] = True

            # Recur to place the rest of the queens
            found_solution = solve_n_queens_branch_and_bound(row
+ 1, n, cols, diags1, diags2, board) or found_solution
            board[row][col] = False
            cols[col] = False
            diags1[row + col] = False
            diags2[row - col + (n - 1)] = False
    return found_solution
```

Similar to the backtracking function, but with optimizations:

- When placing a queen, it marks the column and diagonals as occupied
- When backtracking, it unmarks these constraints
- This allows for much faster safety checks

Function: `` `n_queens_branch_and_bound(n)` ``

```
def n_queens_branch_and_bound(n):
    board = [[False] * n for _ in range(n)]  # Initialize n x n
board with False (no queens placed)
    cols = [False] * n
```

```
    diags1 = [False] * (2 * n - 1)  # For "/" diagonals
    diags2 = [False] * (2 * n - 1)  # For "\" diagonals
    if not solve_n_queens_branch_and_bound(0, n, cols, diags1, 
diags2, board):
        print("No solution exists.")
```

This function:

- Creates an empty board
- Initializes tracking arrays for columns and both diagonal types
- There are 2n-1 possible diagonals in each direction on an n×n board
- Calls the recursive function starting from row 0

Theory Behind The Algorithms

Backtracking

Backtracking is a general algorithmic technique that incrementally builds candidates for solutions and abandons (backtracks) when it determines a candidate cannot lead to a valid solution.

Key concepts:

1. **Decision Space**: Each row must contain exactly one queen, so we make decisions row by row
2. **Constraints**: Queens cannot share rows, columns, or diagonals
3. **Incremental Building**: We place queens one at a time, checking constraints
4. **Backtracking**: When we can't place a queen in the current row, we go back to the previous row and try a different column

Time Complexity: O(N!), as in the worst case, we might need to explore all possible queen placements (although constraints reduce this significantly)

Branch and Bound

Branch and bound improves upon backtracking by:

1. **Efficient Constraint Checking**: Using arrays to track columns and diagonals
2. **Pruning**: By maintaining these constraints, we avoid exploring paths that would violate them
3. **Memory-Time Tradeoff**: Uses more memory (three tracking arrays) but reduces computation time

The diagonal tracking insight:

- "/" diagonals have the property that row+column is constant
- "" diagonals have the property that row-column is constant (plus offset to avoid negative indices)

Time Complexity: Still O(N!) in worst case, but practically much faster than pure backtracking

Visual Understanding

Board Representation

The board is represented as a 2D array of boolean values:

- `False` represents an empty cell
- `True` represents a cell with a queen

Solution Visualization

For a 4×4 board, a solution looks like:

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

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

Diagonal Checking

The diagonals are tracked in two arrays:

1. **"/" Diagonals**: Cells where row+column is constant
   - For example, (0,2), (1,1), and (2,0) are on the same diagonal where row+col=2
2. **"" Diagonals**: Cells where row-column is constant (with offset)
   - For example, (0,0), (1,1), (2,2), (3,3) are on the same diagonal where row-col=0

Branch and Bound Improvement

The branch and bound approach avoids repeatedly scanning the board to check constraints by tracking:

- Which columns are occupied (cols array)
- Which "/" diagonals are occupied (diags1 array)
- Which "" diagonals are occupied (diags2 array)

This means checking if a position is safe becomes a simple O(1) operation instead of O(N) for each check.

Key Takeaways

1. **Backtracking** is essential for combinatorial problems like N-Queens
2. **Branch and Bound** optimizes backtracking with additional data structures
3. **Pruning** the search space is vital for efficiency

4. **Time-Space Tradeoffs** can dramatically improve algorithm performance
5. **Recursive Solutions** elegantly solve problems with a natural hierarchical structure

The N-Queens problem is a classic example used to teach constraint satisfaction and search algorithms because it clearly demonstrates how to reduce a seemingly exponential search space through clever constraints and pruning techniques.