from collections import deque

# Function to check if placing queen is safe

def is\_safe(board, row, col):

for i in range(row):

if board[i] == col or abs(board[i] - col) == abs(i - row):

return False

return True

# BFS implementation

def bfs\_nqueen(n):

board\_queue = deque([(0, [])]) # (row, current board config)

while board\_queue:

row, board = board\_queue.popleft()

if row == n: # If solution is found

print(board) # Prints the board configuration (queen positions)

continue

for col in range(n):

if is\_safe(board, row, col):

new\_board = board + [col] # Add new queen position

board\_queue.append((row + 1, new\_board))

# Example usage:

n = 4 # For 4-Queens problem

bfs\_nqueen(n)

**Explanation**:

* This is a BFS-based approach where we explore all possible column placements for each row.
* is\_safe() ensures that no queens are attacking each other.
* board\_queue stores the current row and board configuration. When a valid configuration is found, it's printed.

**DFS (Depth-First Search) Solution for N-Queen:**

python

Copy code

# DFS implementation

def dfs\_nqueen(n):

def dfs(row, board):

if row == n: # If all queens are placed, solution is found

print(board) # Prints the board configuration

return

for col in range(n):

if is\_safe(board, row, col):

dfs(row + 1, board + [col]) # Recur for next row

dfs(0, []) # Start DFS from the first row

# Example usage:

n = 4 # For 4-Queens problem

dfs\_nqueen(n)

**Explanation**:

* DFS explores one path fully before backtracking and exploring other options.
* Recursive function dfs() is used to place queens row by row.
* is\_safe() checks if the queen placement is valid for the current board configuration.

**4. UCS (Uniform Cost Search) Solution for N-Queen:**

Since in UCS, cost is the priority, here we assign each queen placement a cost based on its row and column, and search for the configuration with minimum cost (even though N-Queen doesn't typically involve a cost function, we'll simulate it for educational purposes).

python

Copy code

# UCS implementation

def ucs\_nqueen(n):

queue = [(0, 0, [])] # (cost, row, board configuration)

while queue:

queue.sort() # Sort by cost (UCS characteristic)

cost, row, board = queue.pop(0)

if row == n: # Solution found

print(f"Board: {board}, Cost: {cost}")

return

for col in range(n):

if is\_safe(board, row, col):

new\_board = board + [col]

new\_cost = cost + col # Adding column as cost

queue.append((new\_cost, row + 1, new\_board))

# Example usage:

n = 4 # For 4-Queens problem

ucs\_nqueen(n)

**Explanation**:

* UCS prioritizes board configurations based on their "cost", which we artificially set to be the column index of the queens.
* It explores the configurations with the least cost first.

**5. GBFS (Greedy Best-First Search) Solution for N-Queen:**

In GBFS, we'll create a heuristic that helps guide the search. For simplicity, we can use the number of conflicts between queens as a heuristic.

python

Copy code

# Heuristic function: number of conflicts

def heuristic(board, row):

conflicts = 0

for i in range(row):

if board[i] == board[row] or abs(board[i] - board[row]) == abs(i - row):

conflicts += 1

return conflicts

# GBFS implementation

def gbfs\_nqueen(n):

queue = [(0, 0, [])] # (heuristic, row, board configuration)

while queue:

queue.sort() # Sort by heuristic (conflicts)

h\_value, row, board = queue.pop(0)

if row == n: # Solution found

print(f"Board: {board}, Heuristic: {h\_value}")

return

for col in range(n):

if is\_safe(board, row, col):

new\_board = board + [col]

h\_value = heuristic(new\_board, row) # Calculate heuristic

queue.append((h\_value, row + 1, new\_board))

# Example usage:

n = 4 # For 4-Queens problem

gbfs\_nqueen(n)

**Explanation**:

* GBFS uses a heuristic to guide the search. Here, we use a simple heuristic that counts the number of queen conflicts.
* The board configurations are sorted based on this heuristic, and the configuration with the least conflicts is explored first.

**Summary:**

* **BFS** explores the board configurations level by level (row by row), trying all possibilities.
* **DFS** explores one full path, trying one queen placement at a time, and backtracks if no solution is found.
* **UCS** simulates cost-based exploration, although the cost is artificially assigned here.
* **GBFS** uses a heuristic to guide the search towards less conflicting queen placements.

These algorithms help understand how different search techniques can be applied to the same problem.