#A\* N-queens

import heapq

def Calculate\_attacks(board\_position):

attacks = 0 ## to calculate the number of attacks of queens

n = len(board\_position) ## to find the number of queens based on the size of the list

for i in range(n): ## i is the row number of 1st queen

for j in range(i + 1, n): ## j is the row number of the next queen

if board\_position[i] == board\_position[j] or abs(board\_position[i] - board\_position[j]) == j - i: ## checks the diagonal attacks of the queen

attacks += 1

return attacks

## checks if the current board position is the solution of the algo

def is\_Checkmate(board\_position):

return Calculate\_attacks(board\_position) == 0

## For each queen on the board, try moving her to every other column in her row, one at a time.Each time you do that, save the new board setup.At the end, return the list of all those new board positions.

def generate\_legal\_moves(board\_position):

legal\_moves = []

n = len(board\_position)

for i in range(n): ## row

for j in range(n): ## every column in the row

if j != board\_position[i]:

new\_position = board\_position[:]

new\_position[i] = j

legal\_moves.append(new\_position)

return legal\_moves

##Implements the A\* search algorithm.

##start\_position: all queens at column 0 (e.g., [0, 0, 0, 0])

##f = g + h:

##g = steps taken (depth of search)

##h = number of attacks (heuristic)

##open\_list: priority queue of possible boards (sorted by lowest f)

##At each step:

##Take the best board (lowest f).

##Check if it's a solution.

##If not, generate and add all new legal boards.

def a\_star\_search(n):

global count

count = 0

start\_position = [0] \* n

open\_list = []

heapq.heappush(open\_list, (Calculate\_attacks(start\_position), 0, start\_position))

closed\_list = set()

while open\_list:

count += 1

f, g, board\_position = heapq.heappop(open\_list)

print("Current State: ")

print\_chessboard(board\_position)

if is\_Checkmate(board\_position):

return board\_position

closed\_list.add(tuple(board\_position))

for new\_position in generate\_legal\_moves(board\_position):

if tuple(new\_position) not in closed\_list:

heapq.heappush(open\_list, (g + 1 + Calculate\_attacks(new\_position), g + 1, new\_position))

return None

def print\_chessboard(board\_position):

n = len(board\_position)

for i in range(n):

row = ['Q' if col == board\_position[i] else '\_' for col in range(n)]

print(' '.join(row))

print()

if \_\_name\_\_ == "\_\_main\_\_":

n = 8

solution = a\_star\_search(n)

if solution:

print(f"Goal State Found in {count} steps:")

print\_chessboard(solution)

else:

print("No solution Found")