A star algorithm

Step 1 (a) create a 8x8 chessboard to initialize state
(b) Set up an open set to explore different configurations
(c) set up a visited state to display all different configurations

step 2: calculate the number of attacking pairs that the queens should not be in the same row, same column or same diagonal

if state[i] == state[j] or abs(state[i]-state[j]) == j-i || diagonal
same row/column
attacks += 1

then we increment the variable attacks to determine attacking pair

open_set = []

step 3: Assign initial state to open set for the first iteration. If the node is not visited then first put it to the open set, the open set will push the node to heapq (priority queue)
heapq.heappush(open_set, Node(new_state, g, h))

step 4: we calculate total estimated cost, f = g+h where g is the cost to reach the current state, h is the no. of attacks to reach goal state

step 5: In the main loop remove the node with the lowest cost

step 6: we have to determine the next row to place the queen

step 7: this happens in the main loop after doing this we generate new state

update g and calculate h

Hill climbing algorithm

Step 1: Place 8 queens randomly

Step 2: Find the attacking pairs such that no queens are placed in the same row, same column or same diagonal

Step 3: Place 8 queens randomly on the chessboard

state = [random. randint (0,7) for in range (8)]

current_attack = calculate_attack (state)

Step 4: In the previous state calculate attacks choose one with few attacking pairs

Step 5: if next_attack >= current_attack
break

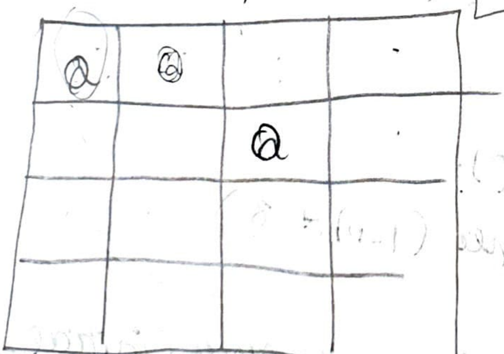
update the current state

state = next_state

current_attack = new_attack

Step 6: display board

$\boxed{8 = 9th}$



Passed

A star algorithm

```
import heapq
class Node:
    def _init_(self, state, g, h):
        self.state = state
        self.g = g
        self.h = h
        self.f = g+h

    def _lt_(self, other):
        return self.f < other.f

def heuristic(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i+1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j])
            == j - i:
                attacks += 1
    return attacks

def a_star_8queens():
    initial_state = tuple([-1] * 8)
    open_set = []
    heapq.heappush(open_set, Node(initial_state, 0,
    heuristic(initial_state)))
    visited = set()
    while open_set:
        current_node = heapq.heappop(open_set)
        current_state = current_node.state

if current_node.h == 0 and -1 not in current_state:
return current.state
```

```
if current_state in visited:
    continue
visited.add(current_state)
next_row = current_state.index(-1) if -1 in
current_state else len(current_state)
if next_row < 8:
    for col in range(8):
        new_state = list(current_state)
        new_state[next_row] = col
        new_state = tuple(new_state)
        if new_state not in visited:
            g = current_node.g + 1
            h = heuristic(new_state)
            heapq.heappush(open_list, Node(new_state, g, h))

return None

def display(state):
    for row in range(8):
    line = " "
        for col in range(8):
            if state[row] == col:
                line += "Q"
            else:
                line += "."
        print(line)
    print()
solution = a_star_8_queens()
if solution:
    print("A* solution:")
    display(solution)
else:
    print("No solution found")
```

Output

A* solution

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

Hill climbing algorithm

```
import random
def calculate-attacks (state):
    attacks = 0
    for i in range (len (state)):
        for j in range (i+1, len( state)):
            if state[i] == state (j) or abs (state [i] - state[j] == j - i:
                attacks += 1
    return attacks

def hill ():
    state = [random. randint (0, 7) for _ in range (8)]
    ca = calculate- attack (state)
    for _ in range (100):
        n = []
        for r in range (8):
            for c in range (8):
                if state [row] != col:
                    n = state [:]
                    n[r] = c
                    n. append (n)
    next- state = min (n, key = calculate_attacks)
    next- attacks = calculate - attacks (next. state)
    if next_attacks >= current-attacks :
        break

    state = next _state
    current-attacks = next_attack
    return state, current-attack

def display (state):
    for r in range (8):
    line = " "
    for c in range (8):
        if state [r] == c :
            line += "Q"
        else:
```

```
line += "."
print(line)
print()

best_solution = None
best_attacks = float("inf")
attempts = 100
for _ in range(attempts):
    s, a = hill();
    if attacks < best_attacks:
        best_solution = solution
        best_attacks = attacks
    if best_attacks == 0:
        break

if best_solution:
    print({best_attacks})
    display(best_solution)
else:
    print("No solution found")
```

Output:

Best solution found

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

29/10/24