29/10/24.

DATE: PAGE:

A* algorithm for 8 queens.

1) import heap queue.
2) Create class node, define a function init with a parameter state, g, h.
3) state: current configuration, g: cost from the start node to current node (number of moves made), h: heuristic estimate of the cost to reach the goal (no of attacks), f: total estimated cost (g+h)
4) define a function less than operator for priority.

```
def __lt__(self, other):
    return self.f < other.f
```

5) define a function heuristic with parameter 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
```

6) define a function A_star_8_queens

```
initial_state = tuple([-1]*8)
open_set = [] push a openset, initial_set, 0,
heuristic to the queue.
visited = set()
```

it will pop the current node from the priority queue. if node visited it will do the further processing it creates through all colums for each new state calculate g and h

7) Define a function display board.

```
for i in range(8):
    line = ""
    for col in
```

```
if state[row] == col:
    line += "Q"
else line += ".".
print(line)
solution = a_star_8_queens()
```

print the display_board and it will print the solution if it found @ else it will print solution not found.

Hill climbing for 8 queens:

1) Import random.
2) Define a function calculate attacks

```
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-1:
            attacks += 1
```

3) Define a function hill_climbing

```
State = [random.randint(0,7)
    for _ in range(8)]
current_attacks = calculate_attacks(state)
for _ in range(10):
    neighbors = [] take a range for rows
and columns.
    if state[row] != col:
        neighbor = state[:]
        neighbor[row] = col
        neighbors.append(neighbor)
```

calculate the new state & take a min of the neighbors.

if next_attacks >= current_attacks
    breals
State = next_state.
Current_attacks = new_attacks
Define a function display_board.
for i in range(8):
    line = " "
    for j in range(8):
        if state[row] == col
            line += "Q"
        else line += "."
    print(line).
for in range(attempts)
solution = hill_climbing
It will check for the best solution and it will print the best solution.

Proceed

→ A* for 8 queens.

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_8_queens():
    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_set, Node(new_state, g, h))
    return None
def display_board(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_board(solution)
else:
    print("No solution found.")
```

output:

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

Hill climbing

```
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
```

DATE.

```
def hill_climbing_8_queens():
    state = [random.randint(0,7) for _ in range(8)]
    current_attacks = calculate_attacks(state)
    for _ in range(10):
        neighbours = []
        for row in range(8):
            for col in range(8):
                if state[row] != col:
                    neighbor = state[:]
                    neighbor[row] = col
                    neighbours.append(neighbor)
        next_state = min(neighbours, key=calculate_attacks)
        next_attacks = calculate_attacks(next_state)
        if next_attacks >= current_attacks:
            break
        state = next_state
        current_attacks = next_attacks
    return state, current_attacks

def display_board(state):
    for row in range(8):
        line = ""
        for col in range(8):
            if state[row] == col:
                line += "Q"
            else:
                line += "."
        print(line)
    print()

best_solution = None
best_attacks = float('inf')
attempts = 10
```

```
for _ in range(attempts):
    solution, attacks = hill_climbing_8_queens()
    if attacks < best_attacks:
        best_solution = solution
        best_attacks = attacks
    if best_attacks == 0:
        break
if best_solution:
    print(f"Best solution found (with {best_attacks}
        attacking pairs):")
    display_board(best_solution)
else:
    print("No solution found")
```

output

Best solution found (with 0 attacking pairs):