Lab-6 8queens using A* and Hill Climbing Algorithm

1) 8-queens using A*:

Taking f(x) as No. of empty spaces left
g(x) as No. of queens left to place
Find state has h(x) = 0

~~Calculating functional value~~: Defining ~~function~~ Initial

```
1) def emptySpaces(board):
       for i in range(8)
           for j in range(8):
```

Initial Setup:

Assuming an 8x8 array of integers representing the chessboard. 1 indicates queen placed on square, 0 indicates not placed

```
board = [[0 for _ in range(8)] for _ in range(8)]
```

Heuristic functional values:

```
1) def emptySpaces(board):  espaces = 0
       for i in range(8)
           for j in range(8):
               row_check = set(board[i])
               col_check = set([board[k][j] for
                           k in range(8)])
               diag_indexes
               diag_check = [(k, k+(j-i)) for k in
                           range(8)]
               for i in diag_indexes:
                   if min(i) < 0 or max(i) > 7:
                       dig_indexes.remove(i)
               diag_check = [board[i[0]][i[1]]
                           for i in diag_indexes]
               diag_check = set(diag_check)
```

if len(row_check) == 1 and len(col_check) == 1 and len(diag_check) == 1:
    spaces += 1
return spaces

2) def no of Queens left(board):
    count = 0
    for row in board:
        if 1 in row:
            count += 1
    return 8 - count

General A* Algorithm:

Step 1: Initialize priority queue

Step 2: Push empty board state into queue

Step 3:

```
def astar(board):
    queue = []
    queue.append((board, 0))
    while queue:
        front = queue.popleft()
        if front[1] == 8: return front
        for i in range(8):
            new_board = front[0][:]
            new_board[front[1]][i] = 1
            queue.append((new_board, front[1]+1))
        queue.sort(key = (emptySpaces(x), noOfQueensLeft(x)))
    return queue[0]
        if (isSafe(board, (front[1], i))):
            queue.append((new_board, front[1]+1))
    return
```

Proceed

→ Hill Climbing Algorithm:

Initial Setup:

Consider a mathematical expression which we have to attempt to maximise

```
def cost(x):
    return math.cos(x)

def hill-climbing(initial-sol = 0, step = 0.01,
                  max_steps = 1000):
    current = initial-sol
    best = current
    for _ in range(max_steps):
        if neighbours = [current + step,
                         current - step]
        neighbours.sort(key = lambda x:
                        cost(x))
        current = neighbour[-1]
        if ~~best~~ cost(best) > cost(current):
            best = current
            ~~current = best~~
        else:
            break
    return best
```

→ Initialization Code:

```
initial-sol = random uniform(-10, 10)
steps = random.choice(0.01, 0.001)
print(hill-climbing(initial-sol, steps))
```

Cropped

A* Output:

Possible solution: (only col no.s)

(0, 4, 7, 5, 2, 6, 1, 3)

Hill Climbing output:

cost function: math.sin(x)

final solution: -4.71634

29/10/24