* LAB-06 :-

* Implementation of A* search algorithm for 8 queens

→ * Algorithm :-

1. create an empty board where no queens are placed which start state.
2. Heuristic (h) : h is the number of attacking queens pairs. where lower values are better.

3. priority $f^n$ : $f(n) = g(n) + h(n)$
   where $g(n)$ is the number of queens placed so far
   where as f prioritizes states with fewer attacking queens.
4. Add start state to the priority queue:
   * Add the initial empty board with $f = 0$
   * while queue Not empty
     - Select states with lowest attacking queens
     - Now check for the $sol^n$ :- if all the 8 queens are placed without any difficuties then return the board.
   - place a queen in the next row in valid columns & add each state to the queue.
5. Return solution ~~... queue is empty~~.

DATE: PAGE: 25

* Implementation of Hill climbing algorithm for 8 queens.

→ initialize :-
- start with a random configuration of 8 queens on a board.
where one queen per column is placed.

→ Evaluate Heuristic :-
Now calculate the number of pairs of queens that are attacking each other (lower values are better h).

→ For each queen, move it within it's column to each row & calculate new heuristic for each board-ing state.

→ select the neighboring state with the lowest heuristic value h',
if h' is lower than the current h, update the current state to this neighbor

→ Repeat :- continue evaluating neighbors until h=0

→ If stuck in any state, restart with new state or random state, repeat same

Proceed

DATE: 26 PAGE:

* code:- Pseudocode:

```
function A* (start)
    open_set = priority queen()
    open_set. enqueue (start, priority = heuristic
                               (start))

while open_set is not empty
    current = open_set. dequeue()
    if heuristic (current) == 0:
        return current
    for neighor in neighbors (current):
        cost = heuristic (neighbor)
        open_set. enqueue (neighbor, priority = cost)

function heuristic (state):
    return no of conflicts queen pairs in
        state.
```

* output:

A* solution ; [0, 4, 7, 5, 2, 6, 1, 3]

DATE: 27 PAGE:

* Hill climbing Pseudo code :-

```
function Hill (start)
    current = start
    while true :
        next = best_neighbor (current)
    if heuristic (next) >= heuristic (current)
return current if heuristic (current) = =0
    else failure
function best_neighbor (state)
    return neighbour of state with the lowest
        heuristic
funtion heuristic (state)
    return no of conflicting queen pairs in
        state.
```

* output :

Hill climbing solution : [1, 3, 0, 6, 4, 2, 5, 7]

other possible outputs:

soln : [0, 2, 4, 6, 1, 3, 5, 7]

soln : [3, 0, 6, 2, 5, 1, 7, 4]

29/10/24

**LAB 06**

A* Algorithm  code for 8 Queens:

Code :

```
import numpy as np

import heapq


class Node:

    def __init__(self, state, g, h):

        self.state = state  # current state of the board

        self.g = g          # cost to reach this state

        self.h = h          # heuristic cost to reach goal

        self.f = g + h      # total cost

    def __lt__(self, other):

        return self.f < other.f


def heuristic(state):

    # Count pairs of queens that can attack each other

    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 = [-1] * 8  # -1 means no queen placed

    open_list = []

    closed_set = set()
```

```
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
print("A* solution:", solution)
```

**qoutput:**

```
A* solution: [7, 0, 6, 3, 1, -1, 4, 2]
```

Hill Climbing for 8 queens

```
import random

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 hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]  # Random initial state
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state

        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
```

```
                    new_state[col] = row
                    h = heuristic(new_state)
                    if h < next_h:
                        next_h = h
                        next_state = new_state

        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum
        state = next_state

solution = hill_climbing_8_queens()
print("Hill Climbing solution:", solution)
```

**Output:**

```
Hill Climbing solution: [4, 2, 7, 3, 6, 0, 5, 1]
```

**LAB 06**

## A* Algorithm code for 8 Queens:

## Code :

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g          # cost to reach this state
        self.h = h          # heuristic cost to reach goal
        self.f = g + h      # total cost
    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    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 = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()
```

```
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
print("A* solution:", solution)
```

**qoutput:**

```
A* solution: [7, 0, 6, 3, 1, -1, 4, 2]
```

Hill Climbing for 8 queens

```
import random

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 hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]  # Random initial state
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state

        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
```

```
                    new_state[col] = row
                    h = heuristic(new_state)
                    if h < next_h:
                        next_h = h
                        next_state = new_state

        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum
        state = next_state

solution = hill_climbing_8_queens()
print("Hill Climbing solution:", solution)
```

**Output:**

```
Hill Climbing solution: [4, 2, 7, 3, 6, 0, 5, 1]
```