LAB-6 - Implementing A* and Hill Climbing Algorithm on 8 Queens.

Observation book:

Date 29/10/2024
Page

A* Search Algorithm :

Algorithm :

* Initialize an open set with the starting node and a closed set as empty.
* ~~#~~ Define a function to calculate the heuristic cost ~~for~~ from the start state to the goal state with starting and ending nodes as parameters.
* While the open set is not empty, remove the node with the lowest total cost (including heuristic).
* If goal state is reached, return the path taken and the cost.
* Else, generate neighbours of the current node using heuristics and iterate through each neighbour.
- This neighbour should be within bounds and not belong to the closed set.
* Calculate the cost from the starting node to the neighbour and update the minimum cost if needed.
- Add this neighbour to the closed set and proceed to step 4.
* Once all the neighbours are processed, if the goal state has not been reached and the open set is empty, then no path exists from the given start state to the goal state.

~~Hill climbing Search Algorithm : [For 8-queens]~~

A* Search Algorithm for 8-queens Problem :

* Initialize the open set with initial state and its cost.
* While the open set is not empty :

* Select the state with the lowest-total cost
  - $f(n) = g(n) + h(n)$
  - $g(n) \rightarrow$ No. of queens placed so far
  - $h(n) \rightarrow$ No. of queens remaining to be placed.
* If this state is the goal state $[h(n) = 0]$, then return the solution.
* Generate the next possibilities by placing a queen in the next row.
  - Compute $f(n)$ at each step.
* If a deadend is reached, backtrack.

Hill Climbing Search Algorithm [8-queens]:

* Place 8 queens randomly on the chess board
* Calculate the number of attacking pairs (conflicts)
* If there are 0 attacking pairs, then the goal state has been reached.
* ~~Gen~~ Generate neighbouring states by moving one queen to a different column in the same row.
* Select the neighbour with the fewest conflicts.
* If ~~a~~ ~~neighbour~~ the best neighbour has fewer conflicts than the current state, move to that neighbour.
* Go back to step 2 if the goal state is not reached.

Output A*:

<P.T.O>

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

<u>Output Hill Climbing Search :</u>

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

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

.
.
.

<P.T.O>

Date
Page

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

19/11

A* algorithm:

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):
    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

# Get user input for the initial state
initial_state = []
for i in range(8):
    while True:
        try:
            row = int(input(f"Enter row for queen {i+1} (0 to 7): "))
            if 0 <= row < 8 and row not in initial_state:
```

```
                initial_state.append(row)
                break
            else:
                print("Invalid row. Enter a number between 0 and 7, and each row must be unique.")
        except ValueError:
            print("Invalid input. Please enter an integer between 0 and 7.")


# Execute the A* algorithm
solution = a_star_8_queens(initial_state)
if solution:
    print("A* solution:", solution)
else:
    print("A* solution: No solution found.")
print(“Nikhilesh 1bm22cs181”)
```

output:

```
A* solution: [0, 2, 4, 6, 1, 3, 5, 7]
1BM22CS181
>>>
```

Hill climbing:

Code:

```
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(initial_state):
    state = initial_state  # Start with user-provided 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

# Get user input for the initial state
initial_state = []
for i in range(8):
    while True:
        try:
            row = int(input(f"Enter row for queen {i+1} (0 to 7): "))
            if 0 <= row < 8 and row not in initial_state:
                initial_state.append(row)
                break
            else:
                print("Invalid row. Enter a number between 0 and 7, and each row must be
unique.")
        except ValueError:
            print("Invalid input. Please enter an integer between 0 and 7.")

# Execute the Hill Climbing algorithm
solution = hill_climbing_8_queens(initial_state)
if solution:
    print("Hill Climbing solution:", solution)
else:
    print("Hill Climbing solution: No solution found.")
print("1BM22CS181")
```

Output:

```
Hill Climbing solution: [0, 2, 4, 6, 1, 3, 5, 7]
1BM22CS181
>>>
```