

Date 22/10/24

Program Title: 4 Queens using Hill climbing

Algorithm

The image shows a handwritten page from a notebook with a 'classmate' header. The header includes a 'Date' field and a 'Page' field. The page contains a handwritten algorithm for the 4 Queens problem using Hill climbing. The algorithm is written in a mix of uppercase and lowercase letters. It starts with 'Algorithm:' followed by a description of the 'HILL-CLIMBING (problem)' function. The function returns a state that is a local maximum. The algorithm then initializes 'current' to 'MAKE-NOISE (problem.INITIAL STATE)'. It enters a 'loop do' which finds a 'neighbor' that is a 'highest-valued successor of current' and 'neighbor.VALUE > current.VALUE'. If found, it 'return current.STATE' and 'current <- neighbor'. Below the loop, there are several bullet points: 'State: n queens on n board. One queen per column', 'Variables: x_0, x_1, x_2, x_3 x_i 's row position of queen in column i (1 queen per column)', 'Domain for each variable: $x_i \in \{0, 1, 2, 3, 4\}$ ', 'Initial state: random state', 'Goal state: n queens on board, none attacking each other', 'Neighbour relation: swap the row positions of two queens', 'The number of pairs of queens attacking each other directly or indirectly.' The word 'OUTPUT' is written at the bottom of the page.

classmate
Date _____
Page _____

Algorithm:
function HILL-CLIMBING (problem) returns a state that is
a local maximum
current \leftarrow MAKE-NOISE (problem.INITIAL STATE)
loop do
neighbor \leftarrow a highest-valued successor of current
if neighbor.VALUE > current.VALUE then
return current.STATE
current \leftarrow neighbor

- State: n queens on n board. One queen per column
- Variables: x_0, x_1, x_2, x_3 x_i 's row position of queen in
- column i (1 queen per column)
- Domain for each variable: $x_i \in \{0, 1, 2, 3, 4\}$
- Initial state: random state
- Goal state: n queens on board, none attacking each other
- Neighbour relation:
swap the row positions of two queens
- The number of pairs of queens attacking each
other directly or indirectly.

OUTPUT

Code:

```

import random
def calculate_cost(state):
    attacking_pairs = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacking_pairs += 1
    return attacking_pairs
def get_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors
def hill_climbing(initial_state):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    state_tree = {tuple(current_state): current_cost} # Dictionary to store state and cost
    step = 0
    while True:
        print(f"\nStep {step}:")
        neighbors = get_neighbors(current_state)
        print("Neighbors:")
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            print(f" {neighbor}: Cost = {cost}")
            state_tree[tuple(neighbor)] = cost # Store neighbor state and cost
        best_neighbor = None
        best_cost = current_cost
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor
        if best_cost >= current_cost:
            print(f"\nNo better neighbor found. Final state reached.")
            return current_state, current_cost, state_tree
        current_state = best_neighbor
        current_cost = best_cost
        step += 1
    initial_state = [3, 1, 2, 0]
    final_state, final_cost, state_space_tree = hill_climbing(initial_state)
    print("\nInitial state:", initial_state)
    print("Final state:", final_state)
    print("Final cost (attacking pairs):", final_cost)

```

```

import random
def calculate_cost(state):
    attacking_pairs = 0
    n = len(state)

```

```

for i in range(n):
    for j in range(i + 1, n):
        if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
            attacking_pairs += 1
    return attacking_pairs
def get_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors
def hill_climbing(initial_state):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    state_tree = {tuple(current_state): current_cost} # Dictionary to store state and cost
    step = 0
    while True:
        print(f"\nStep {step}:")
        neighbors = get_neighbors(current_state)
        print("Neighbors:")
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            print(f" {neighbor}: Cost = {cost}")
            state_tree[tuple(neighbor)] = cost # Store neighbor state and cost
        best_neighbor = None
        best_cost = current_cost
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor
        if best_cost >= current_cost:
            print(f"\nNo better neighbor found. Final state reached.")
            return current_state, current_cost, state_tree
        current_state = best_neighbor
        current_cost = best_cost
        step += 1
initial_state = [3, 1, 2, 0]
final_state, final_cost, state_space_tree = hill_climbing(initial_state)
print("\nInitial state:", initial_state)
print("Final state:", final_state)

```

```
print("Final cost (attacking pairs):", final_cost)
```

Snapshot of the output:

```
Step 0:
Neighbors:
  [1, 3, 2, 0]: Cost = 1
  [2, 1, 3, 0]: Cost = 1
  [0, 1, 2, 3]: Cost = 6
  [3, 2, 1, 0]: Cost = 6
  [3, 0, 2, 1]: Cost = 1
  [3, 1, 0, 2]: Cost = 1

Step 1:
Neighbors:
  [3, 1, 2, 0]: Cost = 2
  [2, 3, 1, 0]: Cost = 2
  [0, 3, 2, 1]: Cost = 4
  [1, 2, 3, 0]: Cost = 4
  [1, 0, 2, 3]: Cost = 2
  [1, 3, 0, 2]: Cost = 0

Step 2:
Neighbors:
  [3, 1, 0, 2]: Cost = 1
  [0, 3, 1, 2]: Cost = 1
  [2, 3, 0, 1]: Cost = 4
  [1, 0, 3, 2]: Cost = 4
  [1, 2, 0, 3]: Cost = 1
  [1, 3, 2, 0]: Cost = 1

No better neighbor found. Final state reached.

Initial state: [3, 1, 2, 0]
Final state: [1, 3, 0, 2]
Final cost (attacking pairs): 0
```

OUTPUT

step: 0

neighbours

[1, 3, 2, 0] : cost = 1

[2, 1, 3, 0] : cost = 1

[0, 1, 2, 3] : cost = 6

[3, 2, 1, 0] : cost = 6

$[3, 1, 0, 2] : \text{cost} = 1$

Step 1

Neighbors:

$[3, 1, 2, 0] : \text{cost} = 2$

$[2, 3, 1, 0] : \text{cost} = 2$

$[0, 3, 2, 1] : \text{cost} = 4$

$[1, 2, 3, 0] : \text{cost} = 4$

$[1, 0, 2, 3] : \text{cost} = 2$

$[1, 3, 0, 2] : \text{cost} = 0$

Step 2

Neighbors:

$[3, 1, 0, 2] : \text{cost} = 1$

$[0, 3, 1, 2] : \text{cost} = 1$

$[2, 3, 0, 1] : \text{cost} = 4$

$[1, 0, 3, 2] : \text{cost} = 4$

$[1, 2, 0, 3] : \text{cost} = 1$

$[1, 3, 2, 0] : \text{cost} = 1$

Initial State: $[3, 1, 2, 0]$

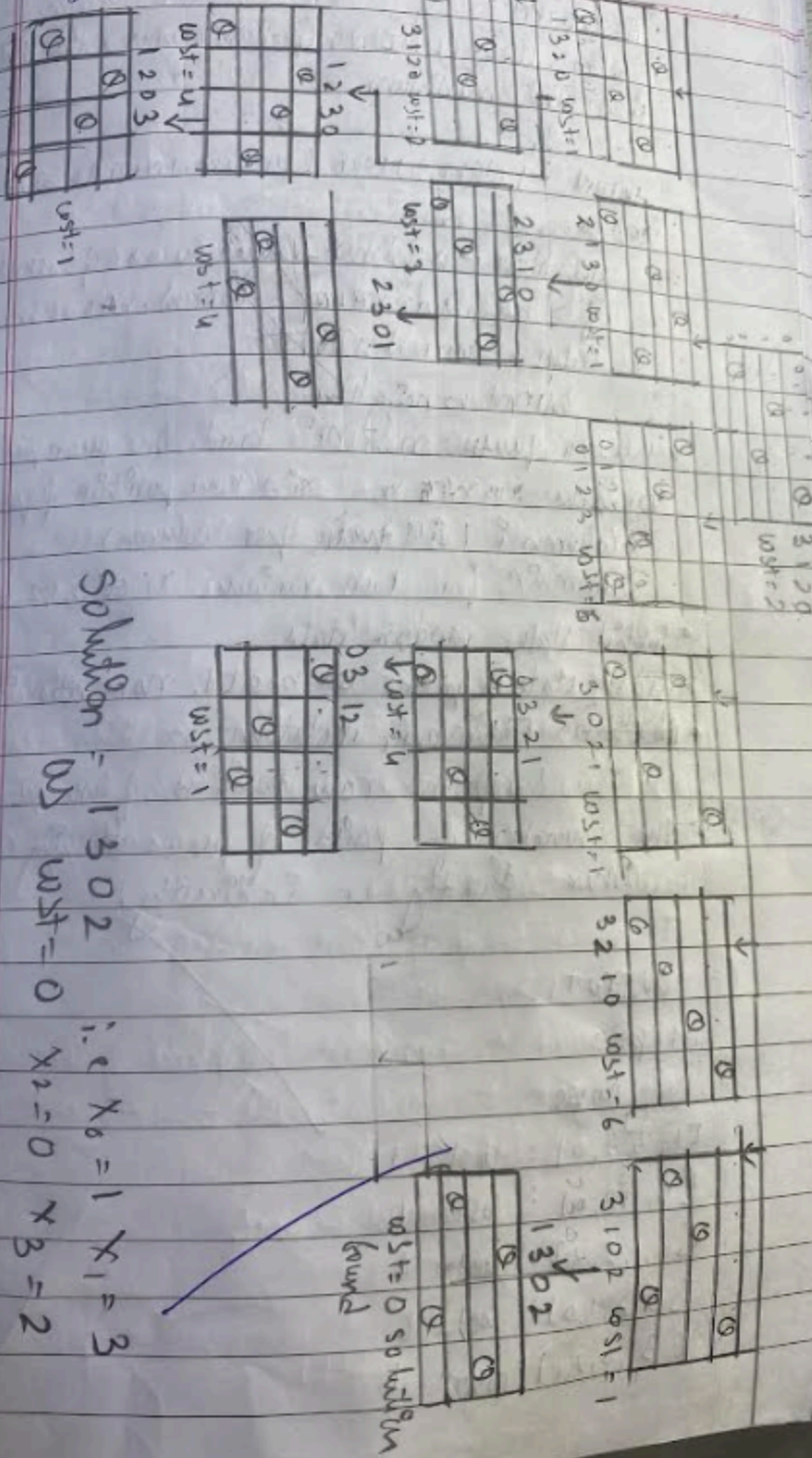
Final state: $[1, 3, 0, 2]$

For Final cost attaching pairs: 0

29/10/24

State Space tree-

N queens using npr climbing Algorithm
state space tree



Solution = 1 3 0 2
 $x_0 = 1$ $x_1 = 3$
 $x_2 = 0$ $x_3 = 2$