

Date: 12/11/24

Program Title: 8 Queens using Simulated Annealing

Algorithm:

8 queens using Simulated Annealing Algorithm

Algorithm:

1. Initialize the current state
2. Set initial temperature (and cooling rate)
3. Compute initial no. of conflicts for each state. I no. of attacking queens
4. While $T > \text{minTemp}$ and conflicts > 0 :
 - a) Generate a neighbor state by swapping two queens in different rows
 - b) Calculate conflicts for neighbor state
 - c) Calculate ΔE between current and neighbor
 - d) If $\Delta E < 0$ take it as new state
 - e) or accept neighbour w.p.m $p = e^{-\Delta E/T}$
5. $T = T \cdot \alpha$
6. Stop when $T < T_{\text{min}}$ or conflicts = 0
7. Return the final state and no. of conflicts

current = initial state
 $T \leftarrow$ a large positive value
while $T > 0$ do
 next = random neighbor of current
 $p \leftarrow \text{current.cost} - \text{next.cost}$
 if $\Delta E > 0$ then
 current = next
 else
 current = next with probability $p = e^{-\Delta E/T}$

end of
do-while
end while
return current

Code:

```

import random
import math
import matplotlib.pyplot as plt
def generate_initial_state():
    return [4, 1, 6, 2, 5, 3, 8, 7]
def count_conflicts(state):
    conflicts = 0
    for i in range(8):
        for j in range(i + 1, 8):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_neighbor(state):
    neighbor = state[:]
    row1, row2 = random.sample(range(8), 2)
    neighbor[row1], neighbor[row2] = neighbor[row2], neighbor[row1]
    return neighbor
def simulated_annealing():
    current_state = generate_initial_state()
    current_conflicts = count_conflicts(current_state)
    temperature = 10000
    cooling_rate = 0.995
    min_temperature = 0.1
    while temperature > min_temperature and current_conflicts > 0:
        neighbor_state = get_neighbor(current_state)
        neighbor_conflicts = count_conflicts(neighbor_state)
        delta_e = neighbor_conflicts - current_conflicts
        if delta_e < 0 or random.random() < math.exp(-delta_e / temperature):
            current_state = neighbor_state
            current_conflicts = neighbor_conflicts
            temperature *= cooling_rate
    return current_state, current_conflicts
final_state, final_conflicts = simulated_annealing()
print(f"Final State: {final_state}")

```

```

print(f"Final State: {final_state}")
print(f"Final Conflicts: {final_conflicts}")

```

```

import random
import math
import matplotlib.pyplot as plt
def generate_initial_state():
    return [4, 1, 6, 2, 5, 3, 8, 7]
def count_conflicts(state):
    conflicts = 0
    for i in range(8):
        for j in range(i + 1, 8):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_neighbor(state):
    neighbor = state[:]
    row1, row2 = random.sample(range(8), 2)

```

```

neighbor[row1], neighbor[row2] = neighbor[row2], neighbor[row1]
return neighbor
def simulated_annealing():
    current_state = generate_initial_state()
    current_conflicts = count_conflicts(current_state)
    temperature = 10000
    cooling_rate = 0.995
    min_temperature = 0.1
    while temperature > min_temperature and current_conflicts > 0:
        neighbor_state = get_neighbor(current_state)
        neighbor_conflicts = count_conflicts(neighbor_state)
        delta_e = neighbor_conflicts - current_conflicts
        if delta_e < 0 or random.random() < math.exp(-delta_e / temperature):
            current_state = neighbor_state
            current_conflicts = neighbor_conflicts
        temperature *= cooling_rate
    return current_state, current_conflicts
final_state, final_conflicts = simulated_annealing()
print(f"Final State: {final_state}")
print(f"Final Conflicts: {final_conflicts}")

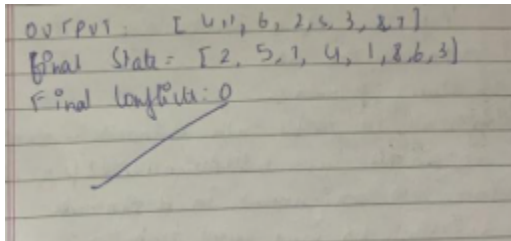
```

Snapshot of the output:

```

→ Final State: [8, 2, 5, 3, 1, 7, 4, 6]
Final Conflicts: 0

```



ORPOT: [4, 1, 6, 2, 5, 3, 8, 7]
 Final State: [2, 5, 3, 4, 1, 8, 6, 7]
 Final Conflicts: 0