Name – Indranil Bain
DEPT. – Computer Science & Technology
Enrolment No. – 2020CSB039, Assignment - 04
Subject – Artificial Intelligence LAB

1. Develop an advanced program to explore and generate solutions for the placement of eight queens on an 8x8 chessboard. The objective is to devise a strategy ensuring that no queen can attack another, presenting a challenge that demands the implementation of a sophisticated algorithm for systematic exploration of feasible queen configurations, introducing heightened complexity in managing the constraints associated with preventing mutual attacks among the queens.

Answer:

```
class EightQueens:
    def __init__(self, size):
        self.size = size
        self.board = [[0 for _ in range(size)] for _ in range(size)]
        self.solutions = []

    def is_safe(self, row, col):
        # Check if there is a queen in the same row
        if any(self.board[row]):
            return False

        # Check if there is a queen in the same column
        if any(self.board[i][col] for i in range(self.size)):
            return False

        # Check if there is a queen in the upper-left to lower-right diagonal
        if any(self.board[i][j] for i, j in zip(range(row, -1, -1), range(col, -1, -1))):
            return False

        # Check if there is a queen in the upper-right to lower-left diagonal
        if any(self.board[i][j] for i, j in zip(range(row, -1, -1), range(col, self.size))):
            return False

        return True

    def solve(self, row):
        if row == self.size:
            # All queens are placed successfully
            self.solutions.append([row[:] for row in self.board])
            return
```

```
        for col in range(self.size):
            if self.is_safe(row, col):
                # Place queen and recursively solve for the next row
                self.board[row][col] = 1
                self.solve(row + 1)
                # Backtrack
                self.board[row][col] = 0

    def find_solutions(self):
        self.solve(0)

    def display_solutions(self):
        for solution in self.solutions:
            print("Solution:")
            for row in solution:
                print(" ".join("Q" if cell else "." for cell in row))
            print()


# Create an instance of EightQueens and find solutions
eight_queens = EightQueens(8)
eight_queens.find_solutions()

# Display all solutions
eight_queens.display_solutions()
```

2. In the context of a room with a door, a monkey finds itself hungry and positioned at the room's entrance. In the center of the room, a banana dangles tantalizingly from the ceiling. However, the monkey's current stature prevents it from reaching the banana directly from the floor. Adjacent to a window, there exists a box that the monkey can potentially employ to its advantage. The monkey is equipped with a repertoire of actions: walking on the floor, climbing the box, pushing the box (if already at it), and grasping the banana if standing on the box directly beneath it.

a. Investigate the feasibility of the monkey successfully obtaining the banana, factoring in its initial position, physical constraints, and the spectrum of actions available within the room.

b. Devise a meticulous and intricate strategy outlining the specific sequence of actions the monkey should execute to secure the banana. This plan should intricately incorporate floor navigation, box utilization, and precise movements to align the monkey underneath the hanging banana. The complexity of this problem-solving task is heightened due to the intricacies involved in orchestrating a successful sequence of actions.

Answer:

```
class Monkey:
    def __init__(self):
        self.position = 'entrance'
        self.box_position = 'window'
        self.banana_position = 'center'
        self.on_box = False

    def walk_to_box(self):
        print("Monkey walks towards the box.")
        self.position = self.box_position

    def climb_box(self):
        print("Monkey climbs the box.")
        self.on_box = True

    def push_box_to_center(self):
        print("Monkey pushes the box towards the center.")
        self.box_position = 'center'

    def align_with_banana(self):
        print("Monkey aligns itself with the banana.")

    def grasp_banana(self):
        print("Monkey grasps the banana.")

    def obtain_banana(self):
        if self.position == 'entrance':
            self.walk_to_box()

        self.climb_box()

        if self.position != 'center':
            self.push_box_to_center()

        self.align_with_banana()
        self.grasp_banana()

# Create a Monkey instance
monkey = Monkey()

# Execute the strategy to obtain the banana
monkey.obtain_banana()
```

3. Develop a program to generate a step-by-step solution for the problem of transporting three missionaries and three cannibals across a river using a boat with a maximum capacity of two individuals. The challenge is subject to the constraint that on both banks, the number of missionaries must not be outnumbered by cannibals. Violating this constraint would result in cannibals consuming the missionaries. Furthermore, the boat cannot traverse the river alone without any passengers. This programming task necessitates the creation of an algorithm capable of systematically determining the sequence of movements that adhere to the specified constraints, showcasing a heightened level of difficulty due to the intricacies involved in managing the compositions of missionaries and cannibals during each crossing.

Answer:

```
from collections import deque

class State:
    def __init__(self, missionaries_left, cannibals_left, boat, missionaries_right,
cannibals_right):
        self.missionaries_left = missionaries_left
        self.cannibals_left = cannibals_left
        self.boat = boat
        self.missionaries_right = missionaries_right
        self.cannibals_right = cannibals_right

    def is_valid(self):
        # Check if the state is valid
        if (self.missionaries_left < 0 or self.cannibals_left < 0 or
            self.missionaries_right < 0 or self.cannibals_right < 0 or
            (self.missionaries_left != 0 and self.missionaries_left < self.cannibals_left) or
            (self.missionaries_right != 0 and self.missionaries_right <
self.cannibals_right)):
            return False
        return True

    def is_goal(self):
        # Check if the state is the goal state
        return self.missionaries_left == 0 and self.cannibals_left == 0

    def __eq__(self, other):
        return (self.missionaries_left == other.missionaries_left and
                self.cannibals_left == other.cannibals_left and
                self.boat == other.boat and
                self.missionaries_right == other.missionaries_right and
                self.cannibals_right == other.cannibals_right)

    def __hash__(self):
        return hash((self.missionaries_left, self.cannibals_left, self.boat,
self.missionaries_right, self.cannibals_right))
```

```
def get_next_states(current_state):
    states = []

    # Possible boat passengers: 0, 1, or 2 people
    for missionaries in range(3):
        for cannibals in range(3):
            if 0 < missionaries + cannibals <= 2:
                # Calculate new state based on passengers in the boat
                if current_state.boat == 'left':
                    new_state = State(
                        current_state.missionaries_left - missionaries,
                        current_state.cannibals_left - cannibals,
                        'right',
                        current_state.missionaries_right + missionaries,
                        current_state.cannibals_right + cannibals
                    )
                else:
                    new_state = State(
                        current_state.missionaries_left + missionaries,
                        current_state.cannibals_left + cannibals,
                        'left',
                        current_state.missionaries_right - missionaries,
                        current_state.cannibals_right - cannibals
                    )

                # Check if the new state is valid and add to the list of possible next
states
                if new_state.is_valid():
                    states.append(new_state)

    return states

def bfs():
    initial_state = State(3, 3, 'left', 0, 0)
    goal_state = State(0, 0, 'right', 3, 3)

    queue = deque([([initial_state], [])])

    while queue:
        path, actions = queue.popleft()
        current_state = path[-1]

        if current_state.is_goal():
            return path, actions

        for next_state in get_next_states(current_state):
            if next_state not in path:
```

```
            queue.append((path + [next_state], actions + [(next_state.boat,
next_state.missionaries_left, next_state.cannibals_left)]))

    return None, None

def print_solution(path, actions):
    for i in range(len(path)):
        state = path[i]
        action = actions[i] if i < len(actions) else None

        print(f"Step {i + 1}:")
        print(f"Missionaries Left: {state.missionaries_left}, Cannibals Left:
{state.cannibals_left}, Boat: {state.boat}")
        print(f"Missionaries Right: {state.missionaries_right}, Cannibals Right:
{state.cannibals_right}")
        if action:
            print(f"Action: Move {action[1]} missionaries and {action[2]} cannibals to the
{action[0]} side.")
        print()

def main():
    path, actions = bfs()

    if path:
        print("Solution found:")
        print_solution(path, actions)
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

4. Create a program to systematically illustrate the steps for a farmer to safely transport a tiger, a goat, and a cabbage across a river using a small boat. The boat can only carry one belonging at a time, and the farmer faces the challenge of ensuring that, at no point during the crossings, the tiger is left alone with the goat or the goat is left alone with the cabbage. Otherwise, the tiger would eat the goat, or the goat would eat the

cabbage. The program must generate a sequence of movements that adhere to these constraints, demonstrating a heightened level of difficulty due to the complex interactions between the farmer, the tiger, the goat, and the cabbage during each crossing.

Answer:

```
from collections import deque
```

```
def is_valid_state(state):
    f, t, g, c = state
    if t == g and f != t: return False
    if g == c and f != g: return False
    return True

def get_next_states(current_state):
    next_states = []
    f, t, g, c = current_state
    new_f = 1 - f
    next_states.append((new_f, t, g, c))
    if f == t: next_states.append((new_f, new_f, g, c))
    if f == g: next_states.append((new_f, t, new_f, c))
    if f == c: next_states.append((new_f, t, g, new_f))
    return [state for state in next_states if is_valid_state(state)]

def bfs_farmer():
    initial_state = (0, 0, 0, 0)
    goal_state = (1, 1, 1, 1)
    frontier = deque([(initial_state, [])])
    explored = set()

    while frontier:
        current_state, path = frontier.popleft()
        if current_state == goal_state:
            return path + [current_state]
        explored.add(current_state)
        for next_state in get_next_states(current_state):
            if next_state not in explored:
                frontier.append((next_state, path + [current_state]))

def describe_step(prev_state, current_state):
    f1, t1, g1, c1 = prev_state
    f2, t2, g2, c2 = current_state
    if f1 != f2:
        if t1 != t2: return "Farmer takes the tiger across the river."
        elif g1 != g2: return "Farmer takes the goat across the river."
        elif c1 != c2: return "Farmer takes the cabbage across the river."
        else: return "Farmer crosses the river alone."
    return "Invalid move"

solution_farmer = bfs_farmer()

if solution_farmer:
    verbose_solution = [describe_step(solution_farmer[i-1], solution_farmer[i]) for i in
range(1, len(solution_farmer))]
    for step in verbose_solution:
        print(step)
```

```
else:
    print("No solution found.")
```