ARTIFICIAL INTELLIGENCE

LAB MANUAL

BALAJI INSTITUTE OF TECHNOLOGY AND SCIENCE

(AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BALAJI INSTITUTE OF TECHNOLOGY & SCIENCE

(AUTONOMOUS)

22CS648PC: ARTIFICIAL INTELLIGENCE LAB (SYLLABUS)

Course Objectives:

- Become familiar with basic principles of AI toward problem solving,knowledge representation,and learning.

Course Outcomes:

- Apply basic principles of AI in solutions that require problem solving,knowledge representation,and learning.

List of Experiments:

Write a Program to Implement the following using Python.

1. BFS, DFS Search Strategy

2. Tic-Tac-Toe game

3. 8-Puzzle problem

4. Water-Jug problem

5. Travelling Salesman Problem

6. Tower of Hanoi

7. Monkey Banana Problem

8. AI Web Robot: Wayback Machine case study project

9. 8-Queens Problem

1. Write a program to implement the following

a) DFS.

b) BFS.

a) A Program to implement DFS

```
graph={
    'A':['B', 'C'],
    'B':['D','E'],
    'C':['F'],
    'D':[],
    'E':['F'],
    'F':[]
}

visited = set()
def dfs(visited, graph, node):
    if node not in visited:
        print(node,end="")
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

dfs(visited, graph, 'A')
```

output: ABDEFC

b) A program to implement BFS Algorithm

```
graph={
    'A':['B', 'C'],
    'B':['D','E'],
    'C':['F'],
    'D':[],
    'E':['F'],
    'F':[]
}

visited = []
queue = []
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        s=queue.pop(0)
        print(s, end=" ")
        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

bfs(visited, graph, 'A')
```

Output: ABCDEF

2. Write a program to implement Tic-Tac-Toe Game

```
# Tic-Tac-Toe Game with 3 consecutive marks to win


def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    """Checks if a player has won with 3 consecutive marks in a row, column, or diagonal."""
    # Check rows-horizontal and columns-vertical
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for j in range(3)):
            return True


    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True


    return False


def is_draw(board):
    """Checks if the game is a draw."""
    if all(board[i][j] in ['X', 'O'] for i in range(3) for j in range(3)):
```

```
        return True
    else:
        return False

def get_move(board, player):
    """Gets a valid move from the player."""
    while True:
        try:
            move = int(input(f"Player {player}, enter your move (1-9): ")) - 1
            row, col = divmod(move, 3)
            if 0 <= move < 9 and board[row][col] == ' ':
                return row, col
            else:
                print("Invalid move! That spot is already taken.")
        except ValueError:
            print("Invalid input! Enter a number between 1 and 9.")

def play_game():
    """Main function to play the game."""
    board = [[' ' for _ in range(3)] for _ in range(3)]
    players = ['X', 'O']
    turn = 0

    print("Welcome to Tic-Tac-Toe!")
    print_board(board)
```

```
    while True:
        player = players[turn % 2]
        row, col = get_move(board, player)
        board[row][col] = player
        print_board(board)

        if check_winner(board, player):
            print(f"Player {player} wins!")
            break
        elif is_draw(board):
            print("It's a draw!")
            break

        turn = turn + 1
# Run the game
play_game()
```

Output:

```
Welcome to Tic-Tac-Toe!
  |  |
---------
  |  |
---------
  |  |
---------
```

```
Player X, enter your move (1-9): 5

  |   |
---------
  | X |
---------
  |   |
---------
Player O, enter your move (1-9): 1

O |   |
---------
  | X |
---------
  |   |
---------
Player X, enter your move (1-9): 2

O | X |
---------
  | X |
---------
  |   |
---------
```

```
Player O, enter your move (1-9): 3

O | X | O

---------

  | X |

---------

  |   |

---------

Player X, enter your move (1-9): 8

O | X | O

---------

  | X |

---------

  | X |

---------

Player X wins!
```

4. Write a program to implement 8 Puzzle Problem

```
from collections import deque


def is_solvable(board):
    flat_board = [num for row in board for num in row]
    inversions = sum(
        1 for i in range(len(flat_board)) for j in range(i + 1, len(flat_board))
        if flat_board[i] and flat_board[j] and flat_board[i] > flat_board[j]
    )
    return inversions % 2 == 0


def get_neighbors(board):
    neighbors = []
    x, y = [(i, j) for i in range(3) for j in range(3) if board[i][j] == 0][0]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_board = [row[:] for row in board]
            new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
            neighbors.append(new_board)

    return neighbors
```

```
def solve_puzzle(start_board):
    goal_state = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
    queue = deque([(start_board, [])])
    visited = set()
    visited.add(tuple(map(tuple, start_board)))

    while queue:
        board, path = queue.popleft()
        if board == goal_state:
            return path + [board]

        for neighbor in get_neighbors(board):
            board_tuple = tuple(map(tuple, neighbor))
            if board_tuple not in visited:
                visited.add(board_tuple)
                queue.append((neighbor, path + [board]))

    return None

def print_puzzle_path(path):
    for step, board in enumerate(path):
        print(f"Step {step}:")
        for row in board:
            print(row)
        print()
```

```
# Example usage
initial_state =  [[1, 2, 5], [3, 4, 0], [6, 7, 8]]
if is_solvable(initial_state):
    solution_path = solve_puzzle(initial_state)
    if solution_path:
        print_puzzle_path(solution_path)
    else:
        print("No solution found.")
else:
    print("Puzzle is not solvable.")
```

Output:

```
Step 0:
[1, 2, 5]
[3, 4, 0]
[6, 7, 8]

Step 1:
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

Step 2:
[1, 0, 2]
```

[3, 4, 5]

[6, 7, 8]

Step 3:

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

4. Write a program to implement Water Jug Problem

```
#jug1 capacity = 5; jug2 cap = 3
def water_jug_problem(a, b, target):
    # Check if the target is achievable
    if target > max(a, b):
        print("Target is greater than both jug capacities")
        return False

    # Use a set to keep track of visited states (amount of water in each jug)
    visited = set()

    # Create a queue to simulate the process (state: (amount in jug 1, amount in jug 2))
    queue = [(0, 0)]  # Start with both jugs empty

    while queue:
        jug1, jug2 = queue.pop(0)
        print(jug1,end=",")
        print(jug2)

        # If the target is reached, return True
        if jug1 == target or jug2 == target:
            return True

        # List all possib actions: fill jug1, fill jug2, empty jug1, empty jug2, transfer from jug1 to
jug2, and transfer from jug2 to jug1
```

```
        possible_states = [
            (a, jug2),  # Fill jug1
            (jug1, b),  # Fill jug2
            (0, jug2),  # Empty jug1
            (jug1, 0),  # Empty jug2
            (jug1 - min(jug1, b - jug2), jug2 + min(jug1, b - jug2)),  # Transfer from jug1 to jug2
            (jug1 + min(jug2, a - jug1), jug2 - min(jug2, a - jug1))   # Transfer from jug2 to jug1
        ]
        # Check each new state, add it to the queue if it's not visited
        for state in possible_states:
            if state not in visited:
                visited.add(state)
                queue.append(state)

    return False

# Example usage  main program
a = 5  # Jug 1 capacity
b = 3  # Jug 2 capacity
target = 4  # Target amount of water

if water_jug_problem(a, b, target):
    print("Target is achievable!")
else:
    print("Target is not achievable.")
```

Output:

0,0

5,0

0,3

0,0

5,3

2,3

3,0

2,0

3,3

0,2

5,1

5,2

0,1

4,3

Target is achievable!

5. Write a program to implement Travelling Salesman Problem.

```
from itertools import permutations


# Distance matrix representing the cost between cities
distance_matrix = [
    [0, 10, 15, 20],  # Distances from city 0
    [10, 0, 35, 25],  # Distances from city 1
    [15, 35, 0, 30],  # Distances from city 2
    [20, 25, 30, 0]   # Distances from city 3
]


num_cities = len(distance_matrix)
cities = list(range(num_cities))  # List of city indices


def calculate_route_distance(route):
    """Calculate the total distance of a given route."""
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    total_distance += distance_matrix[route[-1]][route[0]]  # Returning to start
    return total_distance


# Brute-force approach: Try all possible city orderings
min_distance = float('inf')
best_route = None
```

```
for perm in permutations(cities[1:]):  # Fix city 0 as the start point
    route = (0,) + perm  # Start from city 0
    dist = calculate_route_distance(route)
    if dist < min_distance:
        min_distance = dist
        best_route = route

# Output the best route and distance
print("Best Route:", " -> ".join(map(str, best_route)), "-> 0")
print("Minimum Distance:", min_distance)
```

Output:

Best Route: 0 -> 1 -> 3 -> 2 -> 0

Minimum Distance: 80

6. Write a program to implement Towers of Hanoi

```
class Tower:
    def __init__(self):
        self.terminate = 1

    def printMove(self, source, destination):
        print("{} -> {}".format(source, destination))

    def move(self, disc, source, destination, auxiliary):
        if disc == self.terminate:
            self.printMove(source, destination)
        else:
            self.move(disc - 1, source, auxiliary, destination)
            self.move(1, source, destination, auxiliary)
            self.move(disc - 1, auxiliary, destination, source)
t = Tower();
t.move(3, 'A', 'B', 'C')
```

Output:

```
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```

7. Write a program to implement Monkey-Banana Problem

```
class MonkeyBananaBoxProblem:
    def __init__(self):
        # The monkey starts on the ground (height 0)
        self.monkey_height = 0
        # The box starts at position 0 and must be moved to the correct position under the banana
        self.box_position = 0
        # The banana is hanging at height 3
        self.banana_height = 3
        # The box is initially on the ground (height 1)
        self.box_height = 1

    def can_reach_banana(self):
        # The monkey can reach the banana if it is at or above the height of the banana
        return self.monkey_height >= self.banana_height

    def move_box(self):
        # Move the box closer to the banana (stop when it's directly under the banana)
        if self.box_position < self.banana_height - 1:
            self.box_position += 1
            print(f"Box moved to position: {self.box_position}")
        else:
            print("Box is positioned under the banana.")
            # Set the box height to be directly beneath the banana
            self.box_height = self.banana_height - 1
```

```
    def climb_box(self):
        # The monkey climbs the box, increasing its height by 1
        self.monkey_height = self.box_height + 1
        print(f"Monkey climbs the box! Current height: {self.monkey_height}")

    def attempt_to_get_banana(self):
        # Try to get the banana, check if the monkey is at or above the banana's height
        if self.can_reach_banana():
            print("Monkey has reached the banana and got it!")
        else:
            print("Monkey cannot reach the banana. Moving the box.")
            # Move the box toward the banana until it is positioned beneath it
            self.move_box()

            # Once the box is beneath the banana, the monkey climbs on top of it
            if self.box_position == self.banana_height - 1:
                self.climb_box()

            # After climbing, the monkey attempts to get the banana again
            self.attempt_to_get_banana()


# Instantiate the problem
problem = MonkeyBananaBoxProblem()
```

Start the process of attempting to get the banana

problem.attempt_to_get_banana()

Output:

Monkey cannot reach the banana. Moving the box.

Box moved to position: 1

Monkey cannot reach the banana. Moving the box.

Box moved to position: 2

Monkey climbs the box! Current height: 2

Monkey cannot reach the banana. Moving the box.

Box is positioned under the banana.

Monkey climbs the box! Current height: 3

Monkey has reached the banana and got it!

8. Write a program to implement AI Web Robot: Wayback Machine Case Study

```
import requests


def get_wayback_snapshots(url):
    """
    Fetch archived snapshots of a given URL from the Wayback Machine.
    """
    base_url = "http://web.archive.org/cdx/search/cdx"
    params = {
        "url": url,
        "output": "json",
        "fl": "timestamp,original",
        "collapse": "timestamp",
    }


    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()
        data = response.json()


        if len(data) > 1:
            return data[1:]  # Exclude the header row
        else:
            print("No archives found for this URL.")
            return []
```

```
    except requests.exceptions.RequestException as e:
        print("Error fetching data:", e)
        return []

def get_latest_archive(url):
    """
    Fetch the latest archived version of the given URL.
    """
    snapshots = get_wayback_snapshots(url)
    if snapshots:
        latest_snapshot = snapshots[-1]  # Get the most recent entry
        timestamp, archived_url = latest_snapshot
        archived_url = f"http://web.archive.org/web/{timestamp}/{archived_url}"
        return archived_url
    else:
        return None

# Script executes automatically
website_url = input("Enter the website URL to check Wayback Machine archives: ")


# Fetch and display snapshots
snapshots = get_wayback_snapshots(website_url)
if snapshots:
    print("\nAvailable Snapshots:")
```

```
    for timestamp, original in snapshots[:5]:  # Display first 5 snapshots
        print(f"Date: {timestamp[:4]}-{timestamp[4:6]}-{timestamp[6:8]}, Link: http://web.archive.org/web/{timestamp}/{original}")

    # Get and display the latest archive
    latest_archive = get_latest_archive(website_url)
    if latest_archive:
        print("\nLatest Archived Version:", latest_archive)
    else:
        print("\nNo archived version found.")
```

Output:

Enter the website URL to check Wayback Machine archives: example.com

Available Snapshots:

Date: 1998-12-06, Link: http://web.archive.org/web/19981206052836/http://example.com

Date: 2001-02-08, Link: http://web.archive.org/web/20010208083200/http://example.com

Date: 2005-07-22, Link: http://web.archive.org/web/20050722041041/http://example.com

Latest Archived Version: http://web.archive.org/web/20240315000000/http://example.com

9. Write a program to implement 8 Queens Problem

```
N = 8  # Size of the chessboard (8x8)


# Function to print the chessboard configuration
def print_board(board):
    for row in board:
        for col in row:
            print("Q " if col == 1 else ". ", end="")
        print()


# Function to check if a queen can be placed at board[row][col]
def is_safe(board, row, col):
    # Check the column
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check the left diagonal
    for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check the right diagonal
    for i, j in zip(range(row - 1, -1, -1), range(col + 1, N)):
        if board[i][j] == 1:
```

```
            return False

    return True

# Function to solve the 8-Queens problem using backtracking
def solve_n_queens(board, row):
    if row == N:
        return True  # All queens are placed

    # Try placing a queen in each column of the current row
    for col in range(N):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place the queen

            # Recur to place the next queen
            if solve_n_queens(board, row + 1):
                return True

            # If placing queen doesn't lead to a solution, backtrack
            board[row][col] = 0  # Remove the queen (backtrack)

    return False  # If no queen can be placed in any column

# Main function to solve the 8-Queens problem and print the solution
def main():
```

```
    board = [[0 for _ in range(N)] for _ in range(N)]  # Initialize the chessboard with 0's

    if solve_n_queens(board, 0):
        print_board(board)  # Print the solution
    else:
        print("Solution does not exist.")

main()
```

Output:

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

Programs beyond the Syllabus

1. Write a program to find the solution for wampus world problem

```
import random


# Define the 4x4 grid environment

size = 4

world = [[' ' for _ in range(size)] for _ in range(size)]


# Place Wumpus, Pit, and Gold at random positions

def place_randomly(symbol):

    while True:

        x, y = random.randint(0, size - 1), random.randint(0, size - 1)

        if world[x][y] == ' ' and (x, y) != (0, 0):  # Ensure agent's starting position is safe

            world[x][y] = symbol

            return (x, y)


wumpus_pos = place_randomly('W')

pit_pos = place_randomly('P')

gold_pos = place_randomly('G')


# Agent starts at (0,0)

agent_x, agent_y = 0, 0


# Function to display the world

def display_world():
```

```
    for row in world:
        print(' | '.join(row))
    print("\n")

# Game loop
while True:
    print(f"Agent is at ({agent_x}, {agent_y})")

    # Check surroundings
    if (agent_x, agent_y) == wumpus_pos:
        print("Game Over! The Wumpus ate you!")
        break
    elif (agent_x, agent_y) == pit_pos:
        print("Game Over! You fell into a pit!")
        break
    elif (agent_x, agent_y) == gold_pos:
        print("Congratulations! You found the gold!")
        break

    # Move agent
    move = input("Move (W/A/S/D): ").upper()
    if move == 'W' and agent_x > 0:
        agent_x -= 1
    elif move == 'S' and agent_x < size - 1:
        agent_x += 1
```

```
    elif move == 'A' and agent_y > 0:
        agent_y -= 1
    elif move == 'D' and agent_y < size - 1:
        agent_y += 1
    else:
        print("Invalid move! Try again.")

    display_world()
```

Output:

Agent is at (0, 0)

Move (W/A/S/D): S

Agent is at (1, 0)

Move (W/A/S/D): D

Agent is at (1, 1)

Game Over! The Wumpus ate you!

2. Write a program to implement Hill Climbing Algorithm

```
import random


# Define the function to maximize
def objective_function(x):
    return -(x - 3) ** 2 + 9  # A simple quadratic function


# Hill Climbing Algorithm
def hill_climb(start_x, step_size=0.1, max_iterations=100):
    current_x = start_x  # Start at a random point
    current_value = objective_function(current_x)

    for _ in range(max_iterations):
        # Generate neighbors (small changes in x)
        new_x = current_x + random.choice([-step_size, step_size])
        new_value = objective_function(new_x)

        # If the new value is better, move to new_x
        if new_value > current_value:
            current_x, current_value = new_x, new_value
        else:
            break  # Stop if no improvement
    return current_x, current_value


# Run Hill Climbing from a random starting point
```

```
start = random.uniform(0, 5)  # Start in the range [0,5]
best_x, best_value = hill_climb(start)

# Print the result
print(f"Starting Point: {start:.2f}")
print(f"Optimal x: {best_x:.2f}, Maximum Value: {best_value:.2f}")
```

Output:

Starting Point: 4.12

Optimal x: 3.00, Maximum Value: 9.00

3. Write a program to implement A* Algorithm

```
from queue import PriorityQueue


# Define the grid (0 = free space, 1 = obstacle)
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0]
]


# Define possible movements (up, down, left, right)
moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]


# Heuristic function (Manhattan Distance)
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


# A* Algorithm
def astar(start, goal):
    open_set = PriorityQueue()
    open_set.put((0, start))  # (priority, node)


    came_from = {}  # Store the path
```

```python
    g_score = {start: 0}  # Cost from start
    f_score = {start: heuristic(start, goal)}  # Estimated total cost

    while not open_set.empty():
        _, current = open_set.get()

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]  # Return reversed path

        # Explore neighbors
        for move in moves:
            neighbor = (current[0] + move[0], current[1] + move[1])

            # Check if neighbor is within bounds and not an obstacle
            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and
grid[neighbor[0]][neighbor[1]] == 0:
                temp_g_score = g_score[current] + 1

                if neighbor not in g_score or temp_g_score < g_score[neighbor]:
```

```
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + heuristic(neighbor, goal)
                open_set.put((f_score[neighbor], neighbor))
                came_from[neighbor] = current

    return None  # No path found

# Define start and goal positions
start = (0, 0)
goal = (4, 4)

# Run A* Algorithm
path = astar(start, goal)

# Print result
if path:
    print("Path found:", path)
else:
    print("No path found.")
```

Output:

Path found: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4)]