AI LAB MANUAL

AIML DEPARTMENT (PACE)

1.Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

PROGRAM:

```
class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2

    def __hash__(self):
        return hash((self.jug1, self.jug2))

    def __str__(self):
        return f"({self.jug1},{self.jug2})"


class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def path(self):
        if self.parent is None:
            return [self.state]
        else:
            return self.parent.path() + [self.state]


def dfs(start_state, goal):
    visited = set()
    stack = [Node(start_state)]

    while stack:
        node = stack.pop()
        state = node.state

        if state == goal:
            return node.path()

        visited.add(state)

        actions = [
            (state.jug1, 4),
            (4, state.jug2),
            (0, state.jug2),
            (state.jug1, 0),
            (min(state.jug1 + state.jug2, 4), max(0, state.jug1 + state.jug2 - 4)),
```

```
            (max(0, state.jug1 + state.jug2 - 3), min(state.jug1 + state.jug2, 3))
        ]

        for action in actions:
            new_state = State(action[0], action[1])
            if new_state not in visited:
                stack.append(Node(new_state, node))

    return None


# Test the algorithm with an example
start_state = State(0, 0)  # Initial state: both jugs are empty
goal_state = State(2, 0)  # Goal state: jug1 has 2 units of water

print("Starting DFS for Water Jug Problem...")
path = dfs(start_state, goal_state)

if path:
    print("Solution found! Steps to reach the goal:")
    for i, state in enumerate(path):
        print(f"Step {i}: Jug1: {state.jug1}, Jug2: {state.jug2}")
else:
    print("No solution found!")
```

Output:
Starting DFS for Water Jug Problem...
Solution found! Steps to reach the goal:
Step 0: Jug1: 0, Jug2: 0
Step 1: Jug1: 4, Jug2: 0
Step 2: Jug1: 1, Jug2: 3
Step 3: Jug1: 1, Jug2: 0
Step 4: Jug1: 0, Jug2: 1
Step 5: Jug1: 4, Jug2: 1
Step 6: Jug1: 2, Jug2: 3
Step 7: Jug1: 2, Jug2: 0

2) Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems

PROGRAM:

```
from queue import PriorityQueue

# State representation: (left_missionaries, left_cannibals, boat_position)
INITIAL_STATE = (3, 3, 1)
GOAL_STATE = (0, 0, 0)

def is_valid_state(state):
    left_missionaries, left_cannibals, boat_position = state
    right_missionaries = 3 - left_missionaries
    right_cannibals = 3 - left_cannibals
    # Check if missionaries are outnumbered by cannibals on either side
    if left_missionaries > 0 and left_cannibals > left_missionaries:
        return False
    if right_missionaries > 0 and right_cannibals > right_missionaries:
        return False
    return True

def generate_next_states(state):
    next_states = []
    left_missionaries, left_cannibals, boat_position = state
    new_boat_position = 1 - boat_position
    for m in range(3):
        for c in range(3):
            if 1 <= m + c <= 2:
                if boat_position == 1:
                    new_left_m = left_missionaries - m
                    new_left_c = left_cannibals - c
                else:
                    new_left_m = left_missionaries + m
                    new_left_c = left_cannibals + c
                new_state = (new_left_m, new_left_c, new_boat_position)
                if is_valid_state(new_state):
                    next_states.append(new_state)
    return next_states

def bfs():
    frontier = PriorityQueue()
    frontier.put((0, INITIAL_STATE))
    came_from = {}
    cost_so_far = {INITIAL_STATE: 0}

    while not frontier.empty():
        _, current_state = frontier.get()
        if current_state == GOAL_STATE:
```

```
            break
        for next_state in generate_next_states(current_state):
            new_cost = cost_so_far[current_state] + 1
            if next_state not in cost_so_far or new_cost < cost_so_far[next_state]:
                cost_so_far[next_state] = new_cost
                priority = new_cost
                frontier.put((priority, next_state))
                came_from[next_state] = current_state

    # Reconstruct path
    current_state = GOAL_STATE
    path = [current_state]
    while current_state != INITIAL_STATE:
        current_state = came_from[current_state]
        path.append(current_state)
    path.reverse()
    return path

def print_path(path):
    for i, state in enumerate(path):
        left_missionaries, left_cannibals, boat_position = state
        right_missionaries = 3 - left_missionaries
        right_cannibals = 3 - left_cannibals
        print(f"Step {i}: ({left_missionaries}M, {left_cannibals}C, {'left' if boat_position == 1 else
'right'}) "
              f"-> ({right_missionaries}M, {right_cannibals}C, {'right' if boat_position == 1 else
'left'})")

if __name__ == "__main__":
    path = bfs()
    print("Solution path:")
    print_path(path)
```

output

Solution path:

Step 0: (3M, 3C, left) -> (0M, 0C, right)

Step 1: (2M, 2C, right) -> (1M, 1C, left)

Step 2: (3M, 2C, left) -> (0M, 1C, right)

Step 3: (3M, 0C, right) -> (0M, 3C, left)

Step 4: (3M, 1C, left) -> (0M, 2C, right)

Step 5: (1M, 1C, right) -> (2M, 2C, left)

Step 6: (2M, 2C, left) -> (1M, 1C, right)

Step 7: (0M, 2C, right) -> (3M, 1C, left)

Step 8: (0M, 3C, left) -> (3M, 0C, right)

Step 9: (-1M, 2C, right) -> (4M, 1C, left)

Step 10: (0M, 2C, left) -> (3M, 1C, right)

Step 11: (0M, 0C, right) -> (3M, 3C, left)

3. Implement A* Search algorithm

PROGRAM:

```
import heapq

class Node:
    def __init__(self, state, parent=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def total_cost(self):
        return self.cost + self.heuristic

def astar_search(start_state, goal_state, neighbors_fn, heuristic_fn):
    open_set = []
    closed_set = set()
    start_node = Node(start_state, None, 0, heuristic_fn(start_state))
    heapq.heappush(open_set, (start_node.total_cost(), id(start_node), start_node))

    while open_set:
        _, _, current_node = heapq.heappop(open_set)
        if current_node.state == goal_state:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(current_node.state)

        for neighbor_state in neighbors_fn(current_node.state):
            if neighbor_state in closed_set:
                continue

            neighbor_node = Node(neighbor_state)
            neighbor_node.parent = current_node
            neighbor_node.cost = current_node.cost + 1  # Assuming uniform cost
            neighbor_node.heuristic = heuristic_fn(neighbor_state)

            if any(neighbor_node.state == node.state for _, _, node in open_set):
                continue

            heapq.heappush(open_set, (neighbor_node.total_cost(), id(neighbor_node),
neighbor_node))

    return None
```

```
def neighbors(state):
    x, y = state
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    return [(x + dx, y + dy) for dx, dy in movements]

def heuristic(state):
    x, y = state
    return abs(x) + abs(y)

start_state = (0, 0)
goal_state = (4, 4)
path = astar_search(start_state, goal_state, neighbors, heuristic)
print("Path:", path)
```

output

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

4. Implement AO* Search algorithm

PROGRAM:

```
import heapq

class Node:
    def __init__(self, state, parent=None, cost=0, g=0, h=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.g = g
        self.h = h

    def total_cost(self):
        return self.g + self.h

def ao_star_search(start_state, goal_state, neighbors_fn, heuristic_fn, epsilon):
    open_set = []
    closed_set = set()
    start_node = Node(start_state, None, 0, 0, heuristic_fn(start_state))
    heapq.heappush(open_set, (start_node.total_cost(), id(start_node), start_node))

    while open_set:
        _, _, current_node = heapq.heappop(open_set)
        if current_node.state == goal_state:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(current_node.state)

        for neighbor_state in neighbors_fn(current_node.state):
            if neighbor_state in closed_set:
                continue

            neighbor_node = Node(neighbor_state)
            neighbor_node.parent = current_node
            neighbor_node.g = current_node.g + 1  # Assuming uniform cost
            neighbor_node.h = heuristic_fn(neighbor_state)

            if any(neighbor_node.state == node.state for _, _, node in open_set):
                continue

            heapq.heappush(open_set, (neighbor_node.total_cost() + epsilon *
neighbor_node.h,
```

```
                                    id(neighbor_node), neighbor_node))

    return None

def neighbors(state):
    x, y = state
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    return [(x + dx, y + dy) for dx, dy in movements]

def heuristic(state):
    x, y = state
    return abs(x) + abs(y)

start_state = (0, 0)
goal_state = (4, 4)
epsilon = 1.0
path = ao_star_search(start_state, goal_state, neighbors, heuristic, epsilon)
print("Path:", path)
```

output

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

5. Solve 8-Queens Problem with suitable assumptions

PROGRAM:

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i] == j:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col+1, 8)):
        if board[i] == j:
            return False
    return True

def solve_queens_util(board, row):
    if row >= 8:
        return True
    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if solve_queens_util(board, row + 1):
                return True
            board[row] = -1
    return False

def solve_queens():
    board = [-1] * 8
    if not solve_queens_util(board, 0):
        print("Solution does not exist")
        return False
    print("Solution:")
    for i in range(8):
        for j in range(8):
            if board[i] == j:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()
    return True

solve_queens()
```

output

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

6. Implementation of TSP using heuristic approach

PROGRAM:

```
import numpy as np

def tsp_nearest_neighbor(distances):
    num_cities = distances.shape[0]
    visited = [False] * num_cities
    tour = []
    current_city = 0
    tour.append(current_city)
    visited[current_city] = True

    for _ in range(num_cities - 1):
        nearest_city = None
        nearest_distance = float('inf')

        for next_city in range(num_cities):
            if not visited[next_city] and distances[current_city, next_city] < nearest_distance:
                nearest_city = next_city
                nearest_distance = distances[current_city, next_city]

        current_city = nearest_city
        tour.append(current_city)
        visited[current_city] = True

    tour.append(tour[0])
    return tour

if __name__ == "__main__":
    distances = np.array([
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ])
    tour = tsp_nearest_neighbor(distances)
    print("Tour:", tour)
```

output

Tour: [0, 1, 3, 2, 0]

7. Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining

PROGRAM:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def tell_fact(self, fact):
        self.facts.add(fact)

    def tell_rule(self, antecedent, consequent):
        self.rules.append((antecedent, consequent))

    def forward_chaining(self, goal):
        inferred_facts = set()
        agenda = []
        agenda.extend(self.facts)

        while agenda:
            current_fact = agenda.pop(0)
            if current_fact == goal:
                return True

            for antecedent, consequent in self.rules:
                if antecedent in inferred_facts:
                    if consequent not in inferred_facts and consequent not in agenda:
                        agenda.append(consequent)

            inferred_facts.add(current_fact)
        return False

if __name__ == "__main__":
    kb = KnowledgeBase()
    kb.tell_fact("A")
    kb.tell_fact("B")
    kb.tell_rule("A", "C")
    kb.tell_rule("B", "C")
    kb.tell_rule("C", "D")
    goal = "D"
    if kb.forward_chaining(goal):
        print(f"The goal '{goal}' is reachable.")
    else:
        print(f"The goal '{goal}' is not reachable.")
```

output

The goal 'D' is reachable

8. Implement resolution principle on FOPL related problems

PROGRAM:

```
class Predicate:
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __eq__(self, other):
        return isinstance(other, Predicate) and self.name == other.name and self.args ==
other.args

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

    def __str__(self):
        return f"{self.name}({', '.join(self.args)})"

    def __lt__(self, other):
        if not isinstance(other, Predicate):
            return NotImplemented
        if self.name < other.name:
            return True
        elif self.name == other.name:
            return self.args < other.args
        else:
            return False


class Clause:
    def __init__(self, literals):
        self.literals = set(literals)

    def __eq__(self, other):
        return isinstance(other, Clause) and self.literals == other.literals

    def __hash__(self):
        return hash(tuple(sorted(self.literals)))

    def __str__(self):
        return " | ".join(str(lit) for lit in self.literals)


def resolve(clause1, clause2):
    resolvents = set()
    for lit1 in clause1.literals:
        for lit2 in clause2.literals:
            if lit1.name == lit2.name and lit1.args != lit2.args:
                new_clause_literals = (clause1.literals | clause2.literals) - {lit1, lit2}
```

```
                new_clause = Clause(new_clause_literals)
                resolvents.add(new_clause)
    return resolvents


def resolve_algorithm(knowledge_base, query):
    clauses_to_resolve = list(knowledge_base)
    while clauses_to_resolve:
        clause1 = clauses_to_resolve.pop(0)
        for clause2 in list(knowledge_base):
            if clause1 != clause2:
                resolvents = resolve(clause1, clause2)
                for resolvent in resolvents:
                    if resolvent not in knowledge_base:
                        clauses_to_resolve.append(resolvent)
                        knowledge_base.add(resolvent)
                        if not resolvent.literals:
                            return True
                        if query in resolvent.literals:
                            return True
    return False


if __name__ == "__main__":
    knowledge_base = {
        Clause({Predicate("P", ["a", "b"]), Predicate("Q", ["a"])}),
        Clause({Predicate("P", ["x", "y"])}),
        Clause({Predicate("Q", ["y"]), Predicate("R", ["y"])}),
        Clause({Predicate("R", ["z"])}),
    }
    query = Predicate("R", ["a"])
    result = resolve_algorithm(knowledge_base, query)
    if result:
        print("Query is satisfiable.")
    else:
        print("Query is unsatisfiable.")
```

output

The given set of clauses is satisfiable.

9. Implement Tic-Tac-Toe game using Python

PROGRAM:

```
class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_player = 'X'

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def make_move(self, position):
        if self.board[position] == ' ':
            self.board[position] = self.current_player
            if self.check_winner(position):
                print(f"Player {self.current_player} wins!")
                return True
            elif ' ' not in self.board:
                print("It's a tie!")
                return True
            else:
                self.current_player = 'O' if self.current_player == 'X' else 'X'
                return False
        else:
            print("That position is already taken!")
            return False

    def check_winner(self, position):
        row_index = position // 3
        col_index = position % 3
        # Check row
        if all(self.board[row_index*3 + i] == self.current_player for i in range(3)):
            return True
        # Check column
        if all(self.board[col_index + i*3] == self.current_player for i in range(3)):
            return True
        # Check diagonal
        if row_index == col_index and all(self.board[i*3 + i] == self.current_player for i in range(3)):
            return True
        # Check anti-diagonal
        if row_index + col_index == 2 and all(self.board[i*3 + (2-i)] == self.current_player for i in range(3)):
            return True
        return False

def main():
    game = TicTacToe()
```

```
    while True:
        game.print_board()
        position = int(input(f"Player {game.current_player}, enter your position (0-8): "))
        if game.make_move(position):
            game.print_board()
            break

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

output

```
| | | |

| | | |
| | | |
Player X, enter your position (0-8): 0
| X | | |
| | | |
| | | |
Player O, enter your position (0-8): 1
| X | O | |
| | | |
| | | |
Player X, enter your position (0-8): 3
| X | O | |
| X | | |
| | | |
Player O, enter your position (0-8): 4
| X | O | |
| X | O | |
| | | |
Player X, enter your position (0-8): 6
Player X wins!
```

THANK YOU