VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANASANGAMA, BELAGAVI-590018

Lab Manual for
"ARTIFICIAL INTELLIGENCE"
Subject code: BCS402
4th sem "CSE:AI AND ML"

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING(AI &ML)

KVG
COLLEGE OF ENGINEERING
Kurunjibag, Sullia, Dakshina Kannada, Karnataka, INDIA-574 327
(Approved by AICTE New Delhi, Affiliated to VTU Belagavi)
DEPARTMENT OF COMPUTER SCIENCE (AI & ML)
ESTD: 1986
www.kvgengg.com

Particulars of the experiments performed

Sl. NO.	Experiments
1	Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem
2	Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python
3	Implement A* Search algorithm
4	Implement AO* Search algorithm
5	Solve 8-Queens Problem with suitable assumptions
6	Implementation of TSP using heuristic approach
7	Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining
8	Implement resolution principle on FOPL related problems
9	Implement Tic-Tac-Toe game using Python
10	Build a bot which provides all the information related to text in search box
11	Implement any Game and demonstrate the Game playing strategies

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

```
def dfs(x_capacity, y_capacity, target, visited=set(), current=(0, 0)):
    # Current state
    x_current, y_current = current

    # Check if the current state solves the problem or has been visited
    if (x_current == target or y_current == target):
        return [current]   # Return path containing only the current state if it's a solution
    if current in visited:
        return None

    # Mark the current state as visited
    visited.add(current)

    # List of possible new states
    states = [
        (x_capacity, y_current),  # Fill X completely
        (x_current, y_capacity),  # Fill Y completely
        (0, y_current),           # Empty X
        (x_current, 0),           # Empty Y
        ((x_current - min(x_current, y_capacity - y_current)), (y_current +
min(x_current, y_capacity - y_current))),  # Pour X to Y
        ((x_current + min(y_current, x_capacity - x_current)), (y_current -
min(y_current, x_capacity - x_current)))   # Pour Y to X
    ]

    # Try all possible moves
    for state in states:
        if state not in visited:
            result = dfs(x_capacity, y_capacity, target, visited, state)
            if result is not None:
                return result + [current]  # Append current state to path if succeeding

    # If no state is valid, backtrack by unvisiting the current state
    visited.remove(current)
    return None

def solve_water_jug(x_capacity, y_capacity, target):
```

```
    result = dfs(x_capacity, y_capacity, target)
    if result is None:
        return "No solution"
    else:
        return result[::-1]  # Reverse to display steps from start to finish

# Example usage
x_capacity = 4
y_capacity = 3
target = 2
result = solve_water_jug(x_capacity, y_capacity, target)

print("Steps to achieve target:")
if isinstance(result, str):
    print(result)
else:
    for step in result:
        print(step)
```

output:

```
Steps to achieve target:
(0, 0)
(4, 0)
(4, 3)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

2. Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python.

```
from collections import deque
class State:
    def __init__(self, missionaries, cannibals, boat, parent=None):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat = boat  # 1 if boat is on the original side, 0 if on the other side
        self.parent = parent  # Track the parent of each state for path tracing
    def is_valid(self):
        # Check if the state is valid: missionaries should not be outnumbered by cannibals
        if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibals > 3:
            return False
        # Check that the number of missionaries is never less than cannibals if there are missionaries present
        if self.missionaries < self.cannibals and self.missionaries > 0:
            return False
        if (3 - self.missionaries) < (3 - self.cannibals) and (3 - self.missionaries) > 0:
            return False
        return True
    def is_goal(self):
        return self.missionaries == 0 and self.cannibals == 0 and self.boat == 0
    def __hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat))
    def __eq__(self, other):
        return (self.missionaries == other.missionaries and
                self.cannibals == other.cannibals and
                self.boat == other.boat)
    def __str__(self):
        return f"{self.missionaries}M {self.cannibals}C on left, {3-self.missionaries}M {3-self.cannibals}C on right, {'boat on left' if self.boat else 'boat on right'}"
def successors(state):
    children = []
    if state.boat == 1:  # Boat is on the left side
        movements = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
    else:  # Boat is on the right side
        movements = [(-1, 0), (-2, 0), (0, -1), (0, -2), (-1, -1)]
```

```
    for m, c in movements:
        new_state = State(state.missionaries - m, state.cannibals - c, 1 - state.boat, state)
        if new_state.is_valid():
            children.append(new_state)
    return children
def best_first_search():
    initial_state = State(3, 3, 1)
    frontier = deque([initial_state])
    explored = set()
    while frontier:
        state = frontier.popleft()
        if state.is_goal():
            return state
        explored.add(state)
        for child in successors(state):
            if child not in explored and child not in frontier:
                frontier.append(child)
        frontier = deque(sorted(frontier, key=lambda x: -(x.missionaries + x.cannibals)))
    return None
def print_solution(solution):
    path = []
    while solution:
        path.append(solution)
        solution = solution.parent
    for state in path[::-1]:
        print(state)
# Execute the search and print the solution path
solution = best_first_search()
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution exists.")
```

output:

Solution found:
3M 3C on left, 0M 0C on right, boat on left
3M 1C on left, 0M 2C on right, boat on right
3M 2C on left, 0M 1C on right, boat on left
3M 0C on left, 0M 3C on right, boat on right
3M 1C on left, 0M 2C on right, boat on left
1M 1C on left, 2M 2C on right, boat on right
2M 2C on left, 1M 1C on right, boat on left
0M 2C on left, 3M 1C on right, boat on right
0M 3C on left, 3M 0C on right, boat on left
0M 1C on left, 3M 2C on right, boat on right
1M 1C on left, 2M 2C on right, boat on left
0M 0C on left, 3M 3C on right, boat on right

3. Implement A* Search algorithm .

```
class Graph:
    def __init__(self,adjac_lis):
        self.adjac_lis = adjac_lis
    def get_neighbours(self,v):
        return self.adjac_lis[v]
    def h(self,n):
        H={'A':1,'B':1, 'C':1,'D':1}
        return H[n]
    def a_star_algorithm(self,start,stop):
        open_lst = set([start])
        closed_lst = set([])
        dist ={}
        dist[start] = 0
        prenode ={}
        prenode[start] =start
        while len(open_lst)>0:
            n = None
            for v in open_lst:
                if n==None or dist[v]+self.h(v)<dist[n]+self.h(n):
                    n=v;
            if n==None:
                print("path doesnot exist")
                return None
            if n==stop:
                reconst_path=[]
                while prenode[n]!=n:
                    reconst_path.append(n)
                    n = prenode[n]
                reconst_path.append(start)
                reconst_path.reverse()
                print("path found:{}".format(reconst_path))
                return reconst_path
            for (m,weight) in self.get_neighbours(n):
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    prenode[m] = n
                    dist[m] = dist[n]+weight
                else:
```

```
                    if dist[m]>dist[n]+weight:
                        dist[m] = dist[n]+weight
                        prenode[m]=n
                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)
            open_lst.remove(n)
            closed_lst.add(n)
        print("Path doesnot exist")
        return None
adjac_lis ={'A':[('B',1),('C',3),('D',7)],'B':[('D',5)],'C':[('D',12)]}
graph1=Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

output:

path found:['A', 'B', 'D']

4. Implement AO* Search algorithm .

```
def Cost(H, condition, weight = 1):
        cost = {}
        if 'AND' in condition:
                AND_nodes = condition['AND']
                Path_A = ' AND '.join(AND_nodes)
                PathA = sum(H[node]+weight for node in AND_nodes)
                cost[Path_A] = PathA
        if 'OR' in condition:
                OR_nodes = condition['OR']
        Path_B =' OR '.join(OR_nodes)
                PathB = min(H[node]+weight for node in OR_nodes)
                cost[Path_B] = PathB
        return cost
    def update_cost(H, Conditions, weight=1):
        Main_nodes = list(Conditions.keys())
        Main_nodes.reverse()
        least_cost= {}
        for key in Main_nodes:
                condition = Conditions[key]
                print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
                c = Cost(H, condition, weight)
                H[key] = min(c.values())
                least_cost[key] = Cost(H, condition, weight)
        return least_cost
    def shortest_path(Start,Updated_cost, H):
        Path = Start
        if Start in Updated_cost.keys():
                Min_cost = min(Updated_cost[Start].values())
                key = list(Updated_cost[Start].keys())
                values = list(Updated_cost[Start].values())
                Index = values.index(Min_cost)
                Next = key[Index].split()
                if len(Next) == 1:
                        Start =Next[0]
                        Path += ' = ' +shortest_path(Start, Updated_cost, H)
                else:
```

```
                Path +='=('+key[Index]+') '
                Start = Next[0]
                Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '
                Start = Next[-1]
                Path += shortest_path(Start, Updated_cost, H) + ']'
    return Path
H1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
Conditions = {
'A': {'OR': ['D'], 'AND': ['B', 'C']},
'B': {'OR': ['G', 'H']},
'C': {'OR': ['J']},
'D': {'AND': ['E', 'F']},
'G': {'OR': ['I']}
}
weight = 1
print('Updated Cost :')
Updated_cost = update_cost(H1, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H1))
```

output:

```
Updated Cost :
G : {'OR': ['I']} > {'I': 8}
D : {'AND': ['E', 'F']} > {'E AND F': 5}
C : {'OR': ['J']} > {'J': 2}
B : {'OR': ['G', 'H']} > {'G OR H': 8}
A : {'OR': ['D'], 'AND': ['B', 'C']} > {'B AND C': 12, 'D': 6}
*********************************************************************
******
Shortest Path :
 A = D=(E AND F) [E + F]
```

5. Solve 8-Queens Problem with suitable assumptions.

```
def solve_queens(n):
    def is_safe(board, row, col):
        # Check this row on left side
        for i in range(col):
            if board[row][i] == 'Q':
                return False

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

        # Check lower diagonal on left side
        for i, j in zip(range(row, n, 1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False

        return True

    def solve(board, col):
        # If all queens are placed then return true
        if col >= n:
            return True

        # Consider this column and try placing this queen in all rows one by one
        for i in range(n):
            if is_safe(board, i, col):
                board[i][col] = 'Q'
                # Recur to place rest of the queens
                if solve(board, col + 1):
                    return True
                board[i][col] = '.' # Backtrack

        return False

    # Initialize the board
    board = [['.' for _ in range(n)] for _ in range(n)]
    if solve(board, 0):
```

```
        return board
    else:
        return None

# Solve the 8-queens problem
solution = solve_queens(8)
if solution:
    for row in solution:
        print(" ".join(row))
else:
    print("No solution found")
```

output:

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

6. Implementation of TSP using heuristic approach.

```
import math
import random

# Function to calculate Euclidean distance between two points
def euclidean_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Nearest Neighbor Algorithm
def nearest_neighbor(cities):
    if not cities:
        return []
    # Start from the first city
    tour = [0]
    unvisited = set(range(1, len(cities)))
    current_city = 0

    while unvisited:
        next_city = min(unvisited, key=lambda city:
euclidean_distance(cities[current_city], cities[city]))
        tour.append(next_city)
        unvisited.remove(next_city)
        current_city = next_city

    # Return to the starting city
    tour.append(0)
    return tour

# Function to calculate the total distance of the tour
def calculate_tour_distance(tour, cities):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += euclidean_distance(cities[tour[i]], cities[tour[i + 1]])
    return total_distance

# Example cities (x, y) coordinates
cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(10)]

# Find the tour using Nearest Neighbor
```

```
tour = nearest_neighbor(cities)

# Calculate the distance of the tour
tour_distance = calculate_tour_distance(tour, cities)

# Print results
print("Tour:", tour)
print("Total Distance:", tour_distance)
```

output:

Tour: [0, 4, 5, 1, 3, 8, 9, 7, 6, 2, 0]
Total Distance: 275.0601303160944

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

Forward chaining:

```
def forward_chaining(facts, rules):
    inferred = set()
    while True:
        new_facts = False
        for rule in rules:
            premise, conclusion = rule
            if premise.issubset(facts) and conclusion not in facts:
                facts.add(conclusion)
                inferred.add(conclusion)
                new_facts = True
        if not new_facts:
            break
    return inferred

# Example usage
rules = [
    ({"fever", "cough"}, "flu"),
    ({"fever", "skin rash"}, "measles"),
    ({"headache", "nausea"}, "migraine")
]

symptoms = {"fever", "cough", "skin rash"}
diagnoses = forward_chaining(symptoms, rules)
print("Diagnosed Conditions:", diagnoses)
```

ouput:

Diagnosed Conditions: {'measles', 'flu'}

Backward chaining:

```
def backward_chaining(goals, rules):
    def is_provable(goal, proven=set()):
        if goal in proven:
            return True
        for rule in rules:
            premise, conclusion = rule
            if conclusion == goal and all(is_provable(symptom, proven) for symptom in premise):
                proven.add(goal)
                return True
        return False

    diagnoses = set()
    for goal in goals:
        if is_provable(goal):
            diagnoses.add(goal)
    return diagnoses

# Example usage
rules = [
    ({"fever", "cough"}, "flu"),
    ({"fever", "skin rash"}, "measles"),
    ({"headache", "nausea"}, "migraine")
]

goals = {"flu", "measles", "migraine"}
symptoms = {"fever", "cough", "skin rash"}
diagnoses = backward_chaining(goals, rules)
print("Diagnosed Conditions:", diagnoses)
```

ouput:

```
Diagnosed Conditions: set()
```

8. Implement resolution principle on FOPL related problems.

```
def resolve(c1, c2):
    # Find complementary literals
    for l1 in c1:
        for l2 in c2:
            if l1 == -l2:
                new_clause = set(c1)
                new_clause.update(c2)
                new_clause.remove(l1)
                new_clause.remove(l2)
                return new_clause
    return None  # No resolution possible
def resolution(clauses, query):
    # Negate the query and add to clauses
    clauses.append({-query})
    new_clauses = []
    while True:
        for i in range(len(clauses)):
            for j in range(i + 1, len(clauses)):
                resolvent = resolve(clauses[i], clauses[j])
                if resolvent is not None:
                    if not resolvent:  # Resolvent is an empty set
                        return True
                    if resolvent not in clauses and resolvent not in new_clauses:
                        new_clauses.append(resolvent)

        if not new_clauses:
            return False  # No new clauses were added, resolution failed

        clauses.extend(new_clauses)  # Add new clauses to the set

# Example usage for a simple problem
clauses = [{1, 2}, {-2, 3}, {-3}]
query = 1  # Trying to prove 1

if resolution(clauses, query):
    print("Query is provable.")
else:
    print("Query is not provable.")
```

output:

Query is provable.

9. Implement Tic-Tac-Toe game using Python.

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board, player):
    # Check rows, columns, and diagonals for a win
    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
    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 get_move(board):
    while True:
        try:
            row, col = map(int, input("Enter your move (row col): ").split())
            if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == ' ':
                return row, col
            else:
                print("Invalid move, try again.")
        except ValueError:
            print("Invalid input, please enter numbers.")

def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}, it's your turn.")

        row, col = get_move(board)
        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
```

```
            break

        if all(all(row != ' ' for row in col) for col in board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

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

output:

```
 | |
-----
 | |
-----
 | |
-----
Player X, it's your turn.
Enter your move (row col): 0 0
X| |
-----
 | |
-----
 | |
-----
Player O, it's your turn.
Enter your move (row col): 0 2
X| |O
-----
 | |
-----
 | |
-----
Player X, it's your turn.
Enter your move (row col): 1 0
X| |O
-----
X| |
-----
 | |
```

```
-----
Player O, it's your turn.
Enter your move (row col): 1 2
X |  | O
-----
X |  | O
-----
  |  |
-----
Player X, it's your turn.
Enter your move (row col): 2 0
X |  | O
-----
X |  | O
-----
X |  |
-----
Player X wins!
```

10. Build a bot which provides all the information related to text in search box.

```
def search_bot():
    print("Welcome to the InfoBot. Type 'quit' to exit.")

    data = {
        "python": "Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991.",
        "java": "Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It was originally developed by James Gosling at Sun Microsystems.",
        "javascript": "JavaScript is a programming language that conforms to the ECMAScript specification. It is high-level, often just-in-time compiled and multi-paradigm, enabling dynamic web content.",
        "html": "HTML (Hyper Text Markup Language) is the standard markup language for creating Web pages. It describes the structure of a web page and was created by Tim Berners-Lee.",
        "css": "CSS (Cascading Style Sheets) is a stylesheet language used for describing the presentation of a document written in HTML or XML. CSS describes how elements should be rendered on screen.",
        "sql": "SQL (Structured Query Language) is a domain-specific language used in programming and designed for managing data held in a relational database management system."
    }

    while True:
        query = input("Enter your search term: ").strip().lower()

        if query == "quit":
            print("Exiting the bot. Thank you for using InfoBot!")
            break

        response = data.get(query, "No information available for the search term. Please try another query.")
        print(response)

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

output:

Welcome to the InfoBot. Type 'quit' to exit.
Enter your search term: python
Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991.
Enter your search term: java
Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It was originally developed by James Gosling at Sun Microsystems.
Enter your search term: css
CSS (Cascading Style Sheets) is a stylesheet language used for describing the presentation of a document written in HTML or XML. CSS describes how elements should be rendered on screen.
Enter your search term: quit
Exiting the bot. Thank you for using InfoBot!

11. Implement any Game and demonstrate the Game playing strategies.

```
import random

def rock_paper_scissors():
    choices = ["rock", "paper", "scissors"]
    computer_choice = random.choice(choices)

    user_choice = input("Choose rock, paper, or scissors: ").lower()
    if user_choice not in choices:
        print("Invalid choice, please select rock, paper, or scissors.")
        return

    print(f"Computer chose: {computer_choice}")
    if user_choice == computer_choice:
        print("It's a tie!")
    elif (user_choice == "rock" and computer_choice == "scissors") or \
         (user_choice == "scissors" and computer_choice == "paper") or \
         (user_choice == "paper" and computer_choice == "rock"):
        print("You win!")
    else:
        print("You lose!")

# You can call the function to play the game
rock_paper_scissors()
```

ouput:

```
Choose rock, paper, or scissors: rock
Computer chose: scissors
You win!
```