Sahyog College of Management Studies, Thane (W)
Affiliated to Mumbai University

Course : BSC (Information Technology)
Semester : V

Subject :Artificial Intelligence and Applications

Lab Manual

Practical No : 1

a) Implement depth first search algorithm..
b) Implement breadth first search algorithm

Solution :

a) Implement depth first search algorithm Solution :
Solution :

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
def dfs(graph, node, visited):
    if node not in visited:
        print(node)  # Visit the node
        visited.add(node)  # Mark the node as visited
        for neighbor in graph[node]:  # Recur for all the neighbors
            dfs(graph, neighbor, visited)
visited = set()  # Set to keep track of visited nodes
dfs(graph, 'A', visited)
```

b) Implement breadth first search algorithm Solution :
Solution :

```
from collections import deque

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])
```

```
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            queue.extend(graph[node])

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

bfs(graph, 'A')
```

Practical No : 2

a) Simulate 4-Queen / N-Queen problem
b) Solve tower of Hanoi problem.

a. Simulate 4-Queen / N-Queen problem

Solution :

```
def print_solution(board):
    for row in board:
        print(" ".join("Q" if cell else "." for cell in row))
    print()

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

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

    return True

def solve_n_queens_util(board, col):
    if col >= len(board):
        print_solution(board)
        return True

    res = False
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = True
            res = solve_n_queens_util(board, col + 1) or res
```

```
        board[i][col] = False  # Backtrack

    return res

def solve_n_queens(n):
    board = [[False] * n for _ in range(n)]
    solve_n_queens_util(board, 0)

# Run the 4-Queens problem
solve_n_queens(4)
```

b. Solve tower of Hanoi problem.

Solution :

```
def hanoi(n, from_rod, to_rod, aux_rod):
    if n == 1:
        print(f"Move disk 1 from {from_rod} to {to_rod}")
        return
    hanoi(n-1, from_rod, aux_rod, to_rod)
    print(f"Move disk {n} from {from_rod} to {to_rod}")
    hanoi(n-1, aux_rod, to_rod, from_rod)

hanoi(3, 'A', 'C', 'B')
```

Practical No : 3

c) **Implement A* Algorithm**
d) **Write Programs of Water Jug problem**

c) **Implement A* Algorithm :**
Solution :

```
# Python program for A* Search Algorithm
import math
import heapq

# Define the Cell class


class Cell:
    def __init__(self):
      # Parent cell's row index
        self.parent_i = 0
    # Parent cell's column index
        self.parent_j = 0
 # Total cost of the cell (g + h)
        self.f = float('inf')
    # Cost from start to this cell
        self.g = float('inf')
    # Heuristic cost from this cell to destination
        self.h = 0


# Define the size of the grid
ROW = 9
COL = 10

# Check if a cell is valid (within the grid)


def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked
```

```
def is_unblocked(grid, row, col):
    return grid[row][col] == 1

# Check if a cell is the destination


def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)


def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

# Trace the path from source to destination


def trace_path(cell_details, dest):
    print("The Path is ")
    path = []
    row = dest[0]
    col = dest[1]

    # Trace the path from destination to source using parent cells
    while not (cell_details[row][col].parent_i == row and
cell_details[row][col].parent_j == col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col

    # Add the source cell to the path
    path.append((row, col))
    # Reverse the path to get the path from source to destination
    path.reverse()

    # Print the path
    for i in path:
        print("->", i, end=" ")
```

```
    print()

# Implement the A* search algorithm


def a_star_search(grid, src, dest):
    # Check if the source and destination are valid
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return

    # Check if the source and destination are unblocked
    if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0],
dest[1]):
        print("Source or the destination is blocked")
        return

    # Check if we are already at the destination
    if is_destination(src[0], src[1], dest):
        print("We are already at the destination")
        return

    # Initialize the closed list (visited cells)
    closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
    # Initialize the details of each cell
    cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

    # Initialize the start cell details
    i = src[0]
    j = src[1]
    cell_details[i][j].f = 0
    cell_details[i][j].g = 0
    cell_details[i][j].h = 0
    cell_details[i][j].parent_i = i
    cell_details[i][j].parent_j = j

    # Initialize the open list (cells to be visited) with the start cell
    open_list = []
    heapq.heappush(open_list, (0.0, i, j))

    # Initialize the flag for whether destination is found
    found_dest = False
```

```
    # Main loop of A* search algorithm
    while len(open_list) > 0:
        # Pop the cell with the smallest f value from the open list
        p = heapq.heappop(open_list)

        # Mark the cell as visited
        i = p[1]
        j = p[2]
        closed_list[i][j] = True

        # For each direction, check the successors
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0),
                      (1, 1), (1, -1), (-1, 1), (-1, -1)]
        for dir in directions:
            new_i = i + dir[0]
            new_j = j + dir[1]

            # If the successor is valid, unblocked, and not visited
            if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and not closed_list[new_i][new_j]:
                # If the successor is the destination
                if is_destination(new_i, new_j, dest):
                    # Set the parent of the destination cell
                    cell_details[new_i][new_j].parent_i = i
                    cell_details[new_i][new_j].parent_j = j
                    print("The destination cell is found")
                    # Trace and print the path from source to destination
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    # Calculate the new f, g, and h values
                    g_new = cell_details[i][j].g + 1.0
                    h_new = calculate_h_value(new_i, new_j, dest)
                    f_new = g_new + h_new

                    # If the cell is not in the open list or the new f value is smaller
                    if cell_details[new_i][new_j].f == float('inf') or cell_details[new_i][new_j].f > f_new:
                        # Add the cell to the open list
                        heapq.heappush(open_list, (f_new, new_i, new_j))
```

```
                        # Update the cell details
                        cell_details[new_i][new_j].f = f_new
                        cell_details[new_i][new_j].g = g_new
                        cell_details[new_i][new_j].h = h_new
                        cell_details[new_i][new_j].parent_i = i
                        cell_details[new_i][new_j].parent_j = j

    # If the destination is not found after visiting all cells
    if not found_dest:
        print("Failed to find the destination cell")

# Driver Code


def main():
    # Define the grid (1 for unblocked, 0 for blocked)
    grid = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]

    # Define the source and destination
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)


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

d) Solve Water Jug Problem

Solution :

```
print ("Water Jug Problem")
x=int (input("Enter X:"))
y=int (input("Enter Y:"))

# Initial capacities
capacity_a = 4
capacity_b = 3

# Current water in jugs
jug_a = 0
jug_b = 0

while True:
  rno=int (input ("Enter the Rule No"))
  if rno==1:
    # Step 1: Fill Jug A
    jug_a = capacity_a
    print(f"Filled Jug A: [A: {jug_a}, B: {jug_b}]")
  if rno==2:
    # Step 2: Fill Jug B
    jug_b = capacity_b
    print(f"Filled Jug B: [A: {jug_a}, B: {jug_b}]")
  if rno==3:
    # Step 3: Empty Jug A
    jug_a = 0
    print(f"Emptied Jug A: [A: {jug_a}, B: {jug_b}]")
  if rno==4:
    # Step 4: Empty Jug B
    jug_b = 0
    print(f"Emptied Jug B: [A: {jug_a}, B: {jug_b}]")
  if rno==5:
    # Step 5: Transfer from B to A
    transfer_b_to_a = min(jug_b, capacity_a - jug_a)
    jug_a += transfer_b_to_a
    jug_b -= transfer_b_to_a
    print(f"Transferred from B to A: [A: {jug_a}, B: {jug_b}]")
  if rno==6:
    # Step 6: Transfer from A to B
    transfer_a_to_b = min(jug_a, capacity_b - jug_b)
    jug_b += transfer_a_to_b
```

```
    jug_a -= transfer_a_to_b
    print(f"Transferred from A to B: [A: {jug_a}, B: {jug_b}]")
if rno==7:
    # Step 7: Transfer from B to A until A is full
    transfer_b_to_a = min(jug_b, capacity_a - jug_a)
    jug_a += transfer_b_to_a
    jug_b -= transfer_b_to_a
    print(f"Transferred from B to A until A is full: [A: {jug_a}, B: {jug_b}]")
if rno==8:
    # Step 8: Transfer from A to B until B is full
    transfer_a_to_b = min(jug_a, capacity_b - jug_b)
    jug_b += transfer_a_to_b
    jug_a -= transfer_a_to_b
    print(f"Transferred from A to B until B is full: [A: {jug_a}, B: {jug_b}]")
if jug_a==2:
    print(" The result is a Goal state")
    break
```

Practical No : 4

a) Simulate tic – tac – toe game
b) Shuffle deck of cards

a) Simulate tic tac toe game :

Solution :

```
import os
import time

# Initialize the board
board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
player = 1

# Win conditions
Win = 1
Draw = -1
Running = 0
Stop = 1

# Game state
Game = Running
Mark = 'X'

# This Function Draws Game Board
def DrawBoard():
    print(" %c | %c | %c " % (board[1], board[2], board[3]))
    print("   |   |   ")
    print(" %c | %c | %c " % (board[4], board[5], board[6]))
    print("   |   |   ")
    print(" %c | %c | %c " % (board[7], board[8], board[9]))
    print("   |   |   ")

# This Function Checks if position is empty or not
def CheckPosition(x):
    if board[x] == ' ':
        return True
    else:
```

```
        return False

# This Function Checks if a player has won
def CheckWin():
    global Game
    # Horizontal winning condition
    if (board[1] == board[2] and board[2] == board[3] and board[1] != ' '):
        Game = Win
    elif (board[4] == board[5] and board[5] == board[6] and board[4] != ' '):
        Game = Win
    elif (board[7] == board[8] and board[8] == board[9] and board[7] != ' '):
        Game = Win
    # Vertical winning condition
    elif (board[1] == board[4] and board[4] == board[7] and board[1] != ' '):
        Game = Win
    elif (board[2] == board[5] and board[5] == board[8] and board[2] != ' '):
        Game = Win
    elif (board[3] == board[6] and board[6] == board[9] and board[3] != ' '):
        Game = Win
    # Diagonal winning condition
    elif (board[1] == board[5] and board[5] == board[9] and board[5] != ' '):
        Game = Win
    elif (board[3] == board[5] and board[5] == board[7] and board[5] != ' '):
        Game = Win
    # Match Tie or Draw condition
    elif (board[1] != ' ' and board[2] != ' ' and board[3] != ' ' and
          board[4] != ' ' and board[5] != ' ' and board[6] != ' ' and
          board[7] != ' ' and board[8] != ' ' and board[9] != ' '):
        Game = Draw
    else:
        Game = Running

# Start the game
print("Tic-Tac-Toe Game")
print("Player 1 [X] --- Player 2 [O]\n")
print()
print("Please Wait...")
time.sleep(1)

# Game loop
while Game == Running:
    os.system('cls' if os.name == 'nt' else 'clear')
    DrawBoard()
```

```
    if player % 2 != 0:
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'

    # Get the player's move
    choice = int(input("Enter the position between [1-9] where you want to
mark: "))

    # Check if the position is valid
    if CheckPosition(choice):
        board[choice] = Mark
        player += 1
        CheckWin()

# After game over, display the result
os.system('cls' if os.name == 'nt' else 'clear')
DrawBoard()

if Game == Draw:
    print("Game Draw")
elif Game == Win:
    player -= 1
    if player % 2 != 0:
        print("Player 1 Won")
    else:
print("Player 2 Won")
```

b) Shuffle deck of cards

Solution :

```
import random, itertools
deck =
list(itertools.product(range(1,14),["Spade","Club","Hearts","Diamond"]))
random.shuffle((deck))
print (deck)

for i in range(5):
    print (deck[i][0],"of",deck[i][1])
```

Practical No : 5

a) Design an application to simulate number puzzle problem

Solution :

```
import random

class Puzzle:
    def __init__(self):
        self.state = self.initialize_puzzle()

    def initialize_puzzle(self):
        """Initialize the puzzle with a random configuration."""
        puzzle = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        random.shuffle(puzzle)
        return puzzle

    def display(self):
        """Display the current state of the puzzle."""
        print("\nCurrent puzzle state:")
        for i in range(3):
            print(self.state[i * 3:(i + 1) * 3])
        print()

    def find_zero(self):
        """Find the position of the blank space (0) in the puzzle."""
        return self.state.index(0)

    def is_solvable(self):
        """Check if the puzzle is solvable."""
        inversions = 0
        for i in range(len(self.state)):
            for j in range(i + 1, len(self.state)):
                if self.state[i] != 0 and self.state[j] != 0 and self.state[i] > self.state[j]:
                    inversions += 1
        return inversions % 2 == 0

    def move(self, direction):
        """Move the blank space in the specified direction."""
```

```
        zero_index = self.find_zero()
        row, col = divmod(zero_index, 3)

        if direction == "up" and row > 0:
            self.state[zero_index], self.state[zero_index - 3] = self.state[zero_index -
3], self.state[zero_index]
        elif direction == "down" and row < 2:
            self.state[zero_index], self.state[zero_index + 3] = self.state[zero_index
+ 3], self.state[zero_index]
        elif direction == "left" and col > 0:
            self.state[zero_index], self.state[zero_index - 1] = self.state[zero_index -
1], self.state[zero_index]
        elif direction == "right" and col < 2:
            self.state[zero_index], self.state[zero_index + 1] = self.state[zero_index
+ 1], self.state[zero_index]

    def play(self):
        """Start the game."""
        print("Welcome to the 8-Puzzle Game!")
        self.display()

        while True:
            if self.state == [1, 2, 3, 4, 5, 6, 7, 8, 0]:
                print("Congratulations! You've solved the puzzle!")
                break
            move = input("Enter your move (up, down, left, right) or 'quit' to exit:
").strip().lower()
            if move == "quit":
                print("Thanks for playing!")
                break
            elif move in ["up", "down", "left", "right"]:
                self.move(move)
                self.display()
            else:
                print("Invalid move. Please try again.")

if __name__ == "__main__":
    puzzle = Puzzle()

    # Ensure the puzzle is solvable
    while not puzzle.is_solvable():
        puzzle = Puzzle()
puzzle.play()
```

Practical No : 6

a) Solve constraint satisfaction problem.

Solution :

```
from constraint import Problem

# Define the problem
problem = Problem()

# Define variables (regions) and their domains (colors)
colors = ["Red", "Green", "Blue"]
regions = ["Western Australia", "Northern Territory", "South Australia",
           "Queensland", "New South Wales", "Victoria", "Tasmania"]

# Add variables to the problem, all regions can take any color
for region in regions:
    problem.addVariable(region, colors)

# Add constraints - adjacent regions should not have the same color
# Australia map constraints:
# Western Australia is adjacent to Northern Territory, South Australia
problem.addConstraint(lambda a, b: a != b, ("Western Australia", "Northern
Territory"))
problem.addConstraint(lambda a, b: a != b, ("Western Australia", "South
Australia"))

# Northern Territory is adjacent to Western Australia, South Australia,
Queensland
problem.addConstraint(lambda a, b: a != b, ("Northern Territory", "South
Australia"))
problem.addConstraint(lambda a, b: a != b, ("Northern Territory",
"Queensland"))

# South Australia is adjacent to Western Australia, Northern Territory,
Queensland, New South Wales, Victoria
problem.addConstraint(lambda a, b: a != b, ("South Australia", "Queensland"))
problem.addConstraint(lambda a, b: a != b, ("South Australia", "New South
Wales"))
```

```
problem.addConstraint(lambda a, b: a != b, ("South Australia", "Victoria"))

# Queensland is adjacent to Northern Territory, South Australia, New South
Wales
problem.addConstraint(lambda a, b: a != b, ("Queensland", "New South
Wales"))

# New South Wales is adjacent to Queensland, South Australia, Victoria
problem.addConstraint(lambda a, b: a != b, ("New South Wales", "Victoria"))

# Tasmania has no neighbors, no constraints needed for it.

# Solve the problem
solution = problem.getSolutions()

# Display the results
print("Possible colorings of the map of Australia:")
for sol in solution:
    print(sol)
```

Practical No : 7

a) Derive the expressions based on Associative Law.

Solution :

```
def associative_law():
    # Define three numbers
    a = 5
    b = 10
    c = 15

    # Verify the associative law for addition
    addition_left = (a + b) + c
    addition_right = a + (b + c)

    # Verify the associative law for multiplication
    multiplication_left = (a * b) * c
    multiplication_right = a * (b * c)

    # Print results
    print("Associative Law of Addition:")
    print(f"({a} + {b}) + {c} = {addition_left}")
    print(f"{a} + ({b} + {c}) = {addition_right}")
    print("Result of Addition: ", addition_left == addition_right)

    print("\nAssociative Law of Multiplication:")
    print(f"({a} * {b}) * {c} = {multiplication_left}")
    print(f"{a} * ({b} * {c}) = {multiplication_right}")
    print("Result of Multiplication: ", multiplication_left ==
multiplication_right)

# Call the function to verify associative law
associative_law()
```

b) Derive the expressions based on Distributive Law

Solution :

```
# Function to demonstrate the distributive law
def distributive_law(a, b, c):
    # Calculate both sides of the distributive law
```

```
    left_side = a * (b + c)
    right_side = (a * b) + (a * c)

    # Display the results
    print(f"Using a = {a}, b = {b}, c = {c}:")
    print(f"Left Side: a * (b + c) = {a} * ({b} + {c}) = {left_side}")
    print(f"Right Side: (a * b) + (a * c) = ({a} * {b}) + ({a} * {c}) = {right_side}")

    # Check if the law holds
    if left_side == right_side:
        print("The Distributive Law holds: a * (b + c) = (a * b) + (a * c)")
    else:
        print("The Distributive Law does not hold.")

# Input values
a = int(input("Enter the value of a: "))
b = int(input("Enter the value of b: "))
c = int(input("Enter the value of c: "))

# Call the function to demonstrate the distributive law
distributive_law(a, b, c)
```

Practical No : 8

a) Derive the predicate. (for e.g.: Sachin is batsman, batsman is cricketer) - > Sachin is Cricketer

Solution :

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

def is_batsman(person):
    # Predicate: Checks if the person is a batsman
    return person.name == "Sachin"

def is_cricketer(person):
    # Predicate: All batsmen are cricketers
    if is_batsman(person):
        return True
    return False

# Creating an instance for Sachin
sachin = Person("Sachin")

# Check if Sachin is a cricketer
if is_cricketer(sachin):
    print(f"{sachin.name} is a Cricketer.")
else:
    print(f"{sachin.name} is not a Cricketer.")
```

Practical No : 9

a) **Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin. Question: i. Draw Family Tree.**
ii. Define: Clauses, Facts,
Predicates and Rules with conjunction and disjunction

Solution :

```
# Define facts (predicates)
male = {"John", "Bob", "Charlie", "Tom", "Alex"}
female = {"Mary", "Alice", "Diana", "Sara", "Emily"}

# Define parent-child relationships
# (parent, child) tuples
parent = {
    ("John", "Charlie"),
    ("Mary", "Charlie"),
    ("John", "Diana"),
    ("Mary", "Diana"),
    ("Bob", "Tom"),
    ("Alice", "Tom"),
    ("Charlie", "Alex"),
    ("Sara", "Alex"),
}

# Define rules for family relations
def father(f, c):
    return (f, c) in parent and f in male

def mother(m, c):
    return (m, c) in parent and m in female

def grandfather(gf, c):
    for p in male:
        if (gf, p) in parent and (p, c) in parent:
            return True
    return False

def grandmother(gm, c):
```

```
    for p in female:
        if (gm, p) in parent and (p, c) in parent:
            return True
    return False

def brother(b, s):
    if b in male and s != b:
        for p in parent:
            if p[1] == b and (p[0], s) in parent:
                return True
    return False

def sister(sis, s):
    if sis in female and sis != s:
        for p in parent:
            if p[1] == sis and (p[0], s) in parent:
                return True
    return False

def uncle(u, n):
    for p in parent:
        if brother(u, p[1]) and (p[1], n) in parent:
            return True
    return False

def aunt(a, n):
    for p in parent:
        if sister(a, p[1]) and (p[1], n) in parent:
            return True
    return False

def nephew(n, p):
    return (brother(p, n) or sister(p, n)) and n in male

def niece(n, p):
    return (brother(p, n) or sister(p, n)) and n in female

def cousin(c1, c2):
    for p1 in parent:
        for p2 in parent:
            if p1[1] != p2[1] and brother(p1[0], p2[0]):
                if (p1[1], c1) in parent and (p2[1], c2) in parent:
                    return True
```

```
    return False

# Test the rules
print(f"Is John the father of Charlie? {father('John', 'Charlie')}")
print(f"Is Mary the mother of Diana? {mother('Mary', 'Diana')}")
print(f"Is John the grandfather of Alex? {grandfather('John', 'Alex')}")
print(f"Is Alice the grandmother of Tom? {grandmother('Alice', 'Tom')}")
print(f"Is Charlie the brother of Diana? {brother('Charlie', 'Diana')}")
print(f"Is Diana the sister of Charlie? {sister('Diana', 'Charlie')}")
print(f"Is Bob the uncle of Alex? {uncle('Bob', 'Alex')}")
print(f"Is Alice the aunt of Alex? {aunt('Alice', 'Alex')}")
print(f"Is Alex the nephew of Diana? {nephew('Alex', 'Diana')}")
print(f"Is Diana the niece of Bob? {niece('Diana', 'Bob')}")
print(f"Are Charlie and Tom cousins? {cousin('Charlie', 'Tom')}")
```