Output :-

Following is the Breadth-First Search
5 3 7 2 4 8

Practical - 1

1a) Write Python Program to implement breadth first search algorithm

- Intro - BFS is a graph traversal algorithm that explores a graph or tree level by level. It starts at a designed source node and systematically visits all its immediate neighbors then all their unvisited neighbors and so on.

- code :-

```
graph = {
    '5' : ['3', '7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : []
}
visited = []
queue = []

def bfs (visited, graph, node):
    visited.append(node)
    queue.append(node)

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

        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs (visited, graph, '5')
```

No. : 02

Date :

1b) Write Python Program to implement depth first Search algorithm.

- Intro - DFS in AI is a fundamental algorithm used for traversing or searching through graph and tree data structures. It's a type of uninformed search algorithm meaning it does not use any huristic information

- Code :-

```
graph = {
  '5' : ['3', '7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['3'],
  '8' : []
}
visited = set ()

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

Print("Following is the Depth first Search")
dfs (visited, graph, '5')
```

Output :-

Following is the depth First Search
5
3
2
4
7

Output :-

Move disk 1 from Source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from Source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from Source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from Source A to destination C
Move disk 4 from Source A to destination B
Move disk 1 from source C to destination B
Move disk 4 from sourc C to destination A
Move disk 1 from source B to destination A
destination B
Move disk 2 from source C to destination B
Move disk 1 from source B to destination C
Move disk 3 from source C to destination B
Move disk 1 from source A to destination B
Move disk 2 from source A to destionon C
Move disk 1 from source C to destination B

No. : 03
Date :

Practical - 2

2a Write Python Program to implement Tower of Hanoi Problem

- Intro - The Tower of Hanoi is a classic mathematical Puzzle frequently used in AI and Computer science to illustrate and teach fundamental concepts such as recursion, problem - Solving and algorithmic design

- Code :-

```
def Tower of Hanoi (n, source, destination, auxiliary):
    if n==1'.
        print ("Move disk 1 form Source", source,
                      "to destination ", destination)
        return
    TowerOfHanoi (n-1, Source, destination, auxiliary)
    Print("Move disk", n, "from source,
                  "to desination", destination)
    TowerOfHanoi (n-1, auxiliary, destination, source)
n = 4
TowerOfHanoi (n, 'A', 'B', 'C')
```

2b Write Python Program to Solve Water jug Problem

- Intro - The Water Jug Problem is a classic AI puzzle that involves measuring a specific amount of water using two jugs of different capacities. The goal is to find a sequence of operations to reach a desired amount of water in one of the jugs. illustrating State-space search and problem-solving techniques.

- Code :-

```
def pour(jug1, jug2):
    max1, max2, fill = 5, 7, 4

    print("%d\t%d" % (jug1, jug2))
    if jug2 is fill:
        return

    elif jug2 is max2:
        pour(0, jug1)
    elif jug1 != 0 and jug2 is 0:
        pour(0, jug1)
    elif jug1 is fill:
        pour(jug1, 0)
    elif jug1 < max1:
        pour(max1, jug2)
    elif jug1 < (max2 - jug2):
        pour(0, (jug1 + jug2))
    else:
        pour(jug1 - (max2 - jug2), (max2 - jug2) + jug2)

print("JUG1\tJUG2")
pour(0, 0)
```

Output :-

JUG1	JUG2
0	0
5	0
0	5
5	5
3	7
0	3
5	3
1	7
0	1
5	1
0	6
5	6
4	7
0	4

Scanned with OKEN Scanner

Practical - 3

Simulation of Tic Tac Toe in python

- Intro - The Tic Tac Toe problem in AI involves creating an unbeatable player by developing algorithms that analyze game states, predict moves, and choose optimal actions to win or avoid losing, utilizing techniques

```
import numpy as np
import random
from time import sleep

def create-board():
    return np.zeros((3,3), dtype = int)

def possibilities (board):
    return [(i, j) for i in range (3) for j in
            range(3) if board [i][j] == 0]

def random-place(board, player):
    loc = random.choice (Possibilities (board))
    board [loc] = player
    return board

def row-win (board, player):
    return any (all (cell == Player for cell in row)
                for row in board)

def col-win (board, player):
    return any (all (row [i] == player for row in
                board) for i in range(3))

def diag-win (board, player):
    return all (board [i][i] == Player for i in rage(3))
                or \
```

Output :-

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after move1:
[[0 0 0]
 [1 0 0]
 [0 0 0]]
Board after move2:
[[0 0 0]
 [1 0 0]
 [0 0 2]]
Board after move3:
[[0 0 0]
 [1 0 1]
 [0 0 2]]
Board after move4:
[[0 0 0]
 [1 0 1]
 [2 0 2]]
Board after move5:
[[0 0 1]
 [1 0 1]
 [2 0 2]]
Board after move 6:
[[0 0 1]
 [1 0 1]
 [2 2 2]]
winner is: 2
```

Scanned with OKEN Scanner

No. : 06

Date :

```
all(board[i][2-i]==player for i in range(3))

def evaluate(board):
    for player in [1,2]:
        if row-win(board, player) or col-win(board, player)
                        or diag-win(board, player):
            return player
    return -1 if np.all(board != 0) else 0

def play-game():
    board, winner, move = create-board(), 0, 1
    print(board)
    sleep(1)

    while winner == 0:
        for player in [1,2]:
            board = random-place(board, player)
            print(f"\nBoard after move {move}:\n
                                        {board}")

            sleep(1)
            move += 1
            winner = evaluate(board)
            if winner != 0:
                break

        return winner

    print(f"\nWinner is: {Play-game()}")
```

Output :-

```
MMM CCC|---|
Left side ->right side river travel
Enter number of missionaries to travel=>1
Enter number of Cannibals to travel =>1

Left side: MMCC|-->|Right side: MC
Right side: left side river travel
Enter number of missionaries to travel=>1
Enter number of Cannibals to travel => 0

Left side: MMMCC|<---|Right side: C
Left side => right side river travel
Enter number of missionaries to travel=>0
Enter number of Cannibals to travel=>2

Left side: MMM|--->|Right side: CCC
Right side -> Left side river travel
Enter number of missionaries to travel=>0
Enter number of cannibals to travel=>1

Left side: MMMC|<--|Right side: CC
Left side => right side river travel
Enter number of missionaries to travel=>2
Enter number of Cannibals to travel => 0

Left side: MC|--->|Right side: MMCC
Right side->left side river travel
Enter number of missionaries to travel=>1
Enter number of cannibals to travel => 1

Left side: MMCC|<--|Right Side: MC
Left side -> Right Side river travel
Enter number of missionaries to travel => 2
Enter number of cannibals to travel => 0
You won the game: Congratulations
```

Date :

Practical - 4

Write a python program to solve Missionaries and Cannibals problem.

- Intro - The Missionaries and cannibals problem is a classic AI puzzle where missionaries and Cannibals must cross a river without Cannibals outnumbering missionaries on either side. It used to demonstrate search algorithms and problem - solving techiques in AI

- Code :-

```
lm = 3
lc = 3
rM = 0
rC = 0
k = 0

print ("\nM M M C C C | --- | \n")
try:
    while True:
        while True:
            print ("Left side -> right side river travel")
            uM = int(input ("Enter number of missionaries
                 to travel => "))
            uC = int(input ("Enter number of Cannibals to
                 travel => "))

            if (uM == 0 and uC == 0):
                print ("Empty travel not possible. Re- enter:")
            elif ((uM + uC) <= 2 and (lM - uM) >= 0 and (lC - uC) >= 0):

                lM -= uM
                lC -= uC
                rM += uM
                rC += uC
                K += 1
```

```
        break
    else:
        print("Wrong input , re-enter:")

print ("\n")
print("left side:", "M" * lM +"C" * lC, end =" ")
print("|--->|", end="")
print("Right side: ", "M" * rM +" C " * rC)

if ((lC ==3 and lM in [1,2]) or(rC ==3 and rM in
                                    [1,2])):
    print("Cannibals Outnumber Missionaries: You lost
                            the game")
    break

while True: if (rM + rC == 6):
    print ("You won the game:\n\tCongratulations")
    print ("Total attempts:", k)
    break

while True:
    print("Right side -> left side river travel")
    uM = int(input("Enter nunber of missionaries to
                travel => "))
    uC = int(input(" Enter number of cannibals travel=>"))
if (uM == 0 and uC == 0):
    print ("Empty travel not possible. Re-enter:")
elif ((uM+uC) <=2 and (rM -uM )>=0 and (rC - uC)>=0):

    lM += uM
    lC += uC
    rM -= uM
    rC -= uC
    k += 1
    break
else:
    print("Wrong input , re-enter:")
```

No. : 09

Date :

```
print ("\n")
print ("Left side :", "M" * lM + "C " * lC , end ="").
print ("|<---|", end=" ")
print ("Right side :", "M" * rM + "C" * rC)

    if(lc == 3 and lM in[1,2]) or (rC ==3 and rM
                        in [1,2])):
        print ("Cannibals Outnumber Missonaries : You lost
                        the game")
        break

except ValueError :
    print ("\n\nValid input, plase retry!")
```

Practical 5

5a. Derive the expression based on associative law

- Intro :- The associative law in AI's Boolean algebra states that the grouping of variables doesn't affect the outcome of OR and AND operations, enabling simplification of logical expressions and optimization of AI algorithms.

Code :-

```
def associative_law():
    a = float(input("Enter a value for a:"))
    b = float(input("Enter a value for b:"))
    c = float(input("Enter a value for c:"))

    add_LHS = (a+b)+c
    add_RHS = a+(b+c)

    mul_LHS = (a*b)*c
    mul_RHS = a*(b*c)

    print("\nResults:")

    print(f"\n Addition:")
    print(f"({a} + {b}) + {c} = {add_LHS}")
    print(f"{a} + ({b} + {c}) = {add_RHS}")
    print("Associative law for addition is", "Varified"
        if add_LHS == add_RHS else "not verified")

    print(f"\nMultiplication:")
    print(f"({a} * {b}) * {c} = {mul_LHS}")
    print(f"({a} * ({b} * {c}) = {mul_RHS}")
    print("Associative law for multiplication is",
    "Varified" if mul_LHS == mul_RHS else "not
                        varified")

associative_law()
```

Output :-

Enter Value for a: 3
Enter Value for b: 6
Enter Value for c: 5

Results:
Addition:
(3.0 + 6.0) + 5.0 = 14.0
3.0 + (6.0 + 5.0) = 14.0
Associative law is verified

Mutiplication:
(3.0 * 6.0) * 5.0 = 90.0
3.0 * (6.0 * 5.0) = 90.0
Associative law is verified in multiplication

Scanned with OKEN Scanner

Output :-

Enter a Value for a:3
Enter a Value for b:56
Enter a Value for c : 45
distributive law is verified

5b Derive the expression based on Distributive law

- Intro - The distributive law in Boolean algebra simplifies complex expressions by distributing operations, aiding digital circuit design and AI's reasoning and decision - making.

- Code :-

```
def distributed_law():
    a= float (input ("Enter a Value for a:"))
    b= float (input (" Enter a Value for b:"))
    c = float (input ("Enter a value for c:"))
    LHS = a* (b+c)
    RHS = (a*b)+(a* c)

    if (LHS = RHS):
        print (" distributed law is verified")

    else :
        print (" distributed law is not verified")

distributed_law()
```

Practical-6

Write a program to Simulate N-Queen Problem.

Intro - The N-Queens Problem is a puzzle in Computer Science and mathematics where the goal is to place N queens on an N×N Chessboard so that no two queen can attack each other. This means that no two queen can share the same row, column, or diagonal.

Code -

```
global N
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end='')
        print()

def isSafe(board, row, col):

    for i in range(col):
        if board[row][i] == 1:
            return False

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

    for i, j in zip(range(row, N, 1),
range(col, -1, -1)):
        if board[i][j] = = 1:
            return False

    return True
def solveNQUtil(board, col):
```

```
if col >= N:
    return True
for i in range(N):
    if isSafe(board, i, col):
        board[i][col] = 1

    if solveNQUtil(board, col + 1) == True:
        return True

    board[i][col] = 0
    Return False

def solveNQ():
    board = [[0,0,0,0],
             [0,0,0,0],
             [0,0,0,0],
             [0,0,0,0]]

if solveNQUtil(board, 0) = False:
    print "Solution does not exist"
    return False

printSolution(board)
return True

solveNQ()
```

Output :-

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
True
```

Practical. No. 7

Implement Hill climbing algorithm.

Introduction

Hill climbing algorithm is a heuristic search algorithm used in Artificial Intelligence and optimization problems.

The Idea comes from the real world act of climbing a hill.
You keep moving up hill until you reach a peak where no higher neighbor exists.
It is a kind of local search algorithm - meaning it focuses only on the current state of its immediate neighbors instead of exploring the whole search space

Code

```
v = [4, 1, 3, 7, 5 9]
i= int(input("Start index: "))
while True:
    n=[(i-1, v[i-1]) if i>0 else None,(i+1, v[i+1])
    if (i<len(v)-1 else None]
    n = [x for x in n if x]
    m = max(n, key = lambda x: x[1])
    if m[1] <= v[i] : Break
    i = m[0]
print(f" Local max at index = {i}, value = {v[i]}")
```

Output

Start index: 0
Local max at index = 3, value = 7

Practical. No. 8

Implement Travelling Salesman (Solo travller) algorithm

Introduction

The Travelling Saleman problem is a classic optimization problem:

A salesman has to visit all given cities exactly once and return to the starting city.

The goal is to find the shortest possible route

Algorithm approaches:-

1] Start from a chosen city

2] At each step, visit the nearest unvisited city

3] Repeat until all cities are visited

Code

```
import math
cities = [(0,0), (1,2), (4,3), (6,1)]

visited = [0]
while len(visited) < len(cities):
        last = visited[-1]
        next-city = min((i for i in range (len (cities)).
        if i not in visited),
            key = lambda i: math.hypot(cities[i][0]-cities
            [last][0], cities [i][1]-cities[last][1]))
            visited.append (next_city)
print ("visit order :", visited)
```

Output

Visit Order : [0,1,2,3]