DRONACHARYA
College of Engineering

LABORATORY MANUAL

B.Tech. Semester- VI

ARTIFICIAL INTELLIGENCE LAB USING PYTHON
Subject code: LC-CSE-326G

Prepared by:	**Checked by:**	**Approved by:**
Prof. Renu	Dr. Ashima Mehta	Name : Prof. (Dr.) Isha Malhotra
Sign.: ……	**Sign.: ……**	**Sign.: ……**

DEPARTMENT OF CSE/CSIT/IT/IOT
DRONACHARYA COLLEGE OF ENGINEERING
KHENTAWAS, FARRUKH NAGAR, GURUGRAM (HARYANA)

Table of Contents

Vision and Mission of the Institute

Vision:

To impart Quality Education, to give an enviable growth to seekers of learning, to groom them as World Class Engineers and managers competent to match the expending expectations of the Corporate World has been ever enlarging vision extending to new horizons of Dronacharya College of Engineering

Mission:

1. To prepare students for full and ethical participation in a diverse society and encourage lifelong learning by following the principle of 'Shiksha evam Sahayata' i.e. Education & Help.
2. To impart high-quality education, knowledge and technology through rigorous academic programs, cutting-edge research, & Industry collaborations, with a focus on producing engineers& managers who are socially responsible, globally aware, & equipped to address complex challenges.
3. Educate students in the best practices of the field as well as integrate the latest research into the academics.
4. Provide quality learning experiences through effective classroom practices, innovative teaching practices and opportunities for meaningful interactions between students and faculty.
5. To devise and implement programmes of education in technology that are relevant to the changing needs of society, in terms of breadth of diversity and depth of specialization.

Vision and Mission of the Department

Vision:

"To become a Centre of Excellence in teaching and research in Information Technology for producing skilled professionals having a zeal to serve society"

Mission:

M1: To create an environment where students can be equipped with strong fundamental concepts, programming and problem-solving skills.

M2: To provide an exposure to emerging technologies by providing hands on experience for generating competent professionals.

M3: To promote Research and Development in the frontier areas of Information Technology and encourage students for pursuing higher education

M4: To inculcate in students ethics, professional values, team work and leadership skills.

Programme Educational Objectives (PEOs)

PEO1: To provide students with a sound knowledge of mathematical, scientific and engineering fundamentals required to solve real world problems.

PEO2: To develop research oriented analytical ability among students and to prepare them for making technical contribution to the society.

PEO3: To develop in students the ability to apply state-of-the–art tools and techniques for designing software products to meet the needs of Industry with due consideration for environment friendly and sustainable development.

PEO4: To prepare students with effective communication skills, professional ethics and managerial skills.

PEO5: To prepare students with the ability to upgrade their skills and knowledge for life-long learning.

Programme Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmentalconsiderations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO1: Analyze, identify and clearly define a problem for solving user needs by selecting,creating and evaluating a computer-based system through an effective project plan.

PSO2: Design, implement and evaluate processes, components and/or programs using modern techniques, skills and tools of core Information Technologies to effectively integratesecure IT-based solutions into the user environment.

PSO3: Develop impactful IT solutions by using research-based knowledge and research methods in the fields of integration, interface issues, security & assurance and implementation.

University Syllabus

1. Write a Program to Implement Breadth First Search using Python.

2. Write a Program to Implement Depth First Search using Python.

3. Write a Program to Implement Tic-Tac-Toe game using Python.

4. Write a Program to Implement 8-Puzzle problem using Python.

5. Write a Program to Implement Water-Jug problem using Python.

6. Write a Program to Implement Travelling Salesman Problem using Python.

7. Write a Program to Implement Tower of Hanoi using Python.

8. Write a Program to Implement Monkey Banana Problem using Python.

9. Write a Program to Implement Missionaries-Cannibals Problems using Python.

10. Write a Program to implement 8-Queens Problem using Python.

11. Write a Program to implement Hill Climbing Algorithm.

12. Write a Program to implement A* Algorithm.

Course Outcomes (COs)

Upon successful completion of the course, the students will be able to:

C326.1: To Understand the concept of Artificial intelligence.

C326.2: To apply various search algorithms of artificial intelligence.

C326.3: To apply knowledge representation and reasoning techniques.

C326.4: To understand & apply different types of machine learning and models.

C326.5: To Design various reinforcement algorithms to solve real-time complex problems.

CO-PO Mapping

	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
C326.1	1	3	1	2	1	2			1		3	1
C326.2	1	2	3	1		1	1	1			2	
C326.3	1	2	2	2	2	2	1		1	1	2	1
C326.4	1	3	3	2		2	1		2	2	2	2
C326.5	1	2	1	2	2	2	2	2		1	3	1
C326	**1**	**2.4**	**2**	**1.8**	**1**	**1.8**	**1**	**0.6**	**0.8**	**0.8**	**2.4**	**1**

CO-PSO Mapping

	PSO1	**PSO2**	**PSO3**
C326.1	2	3	2
C326.2	2	2	2
C326.3	2	2	2
C326.4	2	1	2
C326.5	2	2	2
C326	**2**	2	**2**

Course Overview

This course teaches what every student should know about Artificial Intelligence. AI is a fast-moving technology with impacts and implications for both our individual lives and society as a whole. In this course, students will get a basic introduction to the building blocks and components of artificial intelligence, learning about concepts like algorithms, machine learning, and neural networks. Students will also explore how AI is already being used, and evaluate problem areas of AI, such as bias. The course also contains a balanced look at AI's impact on existing jobs, as well as its potential to create new and exciting career fields in the future. Students will leave the course with a solid understanding of what AI is, how it works, areas of caution and what they can do with the technology.

AI is transforming how we live, work, and play. By enabling new technologies like self-driving cars and recommendation systems or improving old ones like medical diagnostics and search engines, the demand for expertise in AI and machine learning is growing rapidly. This course will enable students to take the first step toward solving important real-world problems and future-proofing your career.

Artificial Intelligence with Python explores the concepts and algorithms at the foundation of modern artificial intelligence, diving into the ideas that give rise to technologies like game-playing engines, handwriting recognition, and machine translation. Through hands-on projects, students gain exposure to the theory behind graph search algorithms, classification, optimization, reinforcement learning, and other topics in artificial intelligence and machine learning as they incorporate them into their own Python programs. By course's end, students emerge with experience in libraries for machine learning as well as knowledge of artificial intelligence principles that enable them to design intelligent systems of their own.

List of Experiments mapped with COs

Si No.	Name of the Experiment	Course Outcome
1	Write a Program to Implement Breadth First Search using Python.	C326.2
2	Write a Program to Implement Depth First Search using Python.	C326.2
3	. Write a Program to Implement Tic-Tac-Toe game using Python.	C326.3
4	Write a Program to Implement 8-Puzzle problem using Python.	C326.3
5	Write a Program to Implement Water-Jug problem using Python.	C326.1
6	Write a Program to Implement Travelling Salesman Problem using Python.	C326.5
7	Write a Program to Implement Tower of Hanoi using Python.	C326.3
8	Write a Program to Implement Monkey Banana Problem using Python.	C326.3
9	Write a Program to Implement Missionaries-Cannibals Problems using Python.	C326.3
10	Write a Program to Implement 8-Queens Problem using Python.	C326.4
11	Write a Program to implement Hill Climbing Algorithm.	C326.2
12	Write a Program to implement A* Algorithm.	C326.2

DOs and DON'Ts

DOs

1. Login-on with your username and password.
2. Log off the computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

General Safety Precautions

Precautions (In case of Injury or Electric Shock)

1. To break the victim with live electric source, use an insulator such as fire wood or plastic to break the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.
2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.
3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chestin and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.
4. Immediately call medical emergency and security. Remember! Time is critical; be best.

Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.
2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering it with a heavy cloth if possible isolate the burning equipment from the other surroundingequipment.
3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.
4. Call security and emergency department immediately:

Emergency : Reception

Security : Front Gate

Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows: -

1) All files must contain a title page followed by an index page. ***The files will not be signed by the faculty without an entry in the index page.***

2) Student's Name, Roll number and date of conduction of experiment must be written on allpages.

3) For each experiment, the record must contain the following

(i) Aim/Objective of the experiment
(ii) Pre-experiment work (as given by the faculty)
(iii) Lab assignment questions and their solutions
(iv) Test Cases (if applicable to the course)
(v) Results/ output

Note:

1. Students must bring their lab record along with them whenever they come for the lab.
2. Students must ensure that their lab record is regularly evaluated.

Lab Assessment Criteria

An estimated 10 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment aswell as internal marks in the lab course.

Grading Criteria	Exemplary (4)	Competent (3)	Needs Improvement (2)	Poor (1)
AC1: Pre-Lab written work (this may be assessed through viva)	Complete procedure with underlined concept is properly written	Underlined concept is written but procedureis incomplete	Not able to write concept and procedure	Underlined concept is not clearly understood
AC2: Program Writing/ Modeling	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied, Program/solution written is readable	Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied	Assigned problem is properly analyzed & correct solution designed	Assigned problem is properly analyzed
AC3: Identification & Removal of errors/ bugs	Able to identify errors/ bugs and remove them	Able to identify errors/ bugs and remove them with little bit of guidance	Is dependent totally on someone for identification of errors/ bugs and their removal	Unable to understand the reason for errors/ bugs even after they are explicitly pointed out
AC4:Execution & Demonstration	All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained	All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained	Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained	Solution is not well demonstrated and implemented concept is not clearlyexplained
AC5:Lab Record Assessment	All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of	More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and	Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and	Less than 40 % of the assigned problemsare well recorded with objective, design contracts and solution along with Performance analysisis done with allvariants of input andoutput

	input and output	output	output	

LAB EXPERIMENTS

LAB EXPERIMENT 1

OBJECTIVE:

Write a Program to Implement Breadth First Search using Python.

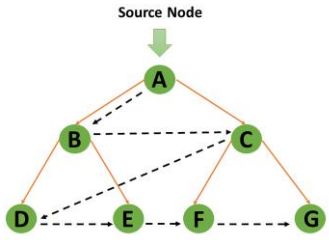
BRIEF DESCRIPTION:

Breadth-First Search Algorithm or BFS is the most widely utilized method.

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

According to the BFS, you must traverse the graph in a breadthwise direction:

- To begin, move horizontally and visit all the current layer's nodes.
- Continue to the next layer.



Breadth-First Search uses a queue data structure to store the node and mark it as "visited" until it marks all the neighboring vertices directly related to it. The queue operates on the First in First out (FIFO) principle, so the node's neighbors will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

PRE-EXPERIMENT QUESTIONS:

1. What is searching?
2. What is QUEUE data structure?

Explanation:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

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

# Driver Code
```

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

Output:

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

POST EXPERIMENT QUESTIONS:

1 .What do you understand by BFS?

2. What is the time and space complexity of BFS?

LAB EXPERIMENT 2

OBJECTIVE:

Write a Program to Implement Depth First Search using Python.

BRIEF DESCRIPTION:

DFS is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

- First, create a stack with the total number of vertices in the graph.
- Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
- After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
- Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
- If no vertex is left, go back and pop a vertex from the stack.
- Repeat steps 2, 3, and 4 until the stack is empty.

PRE EXPERIMENT QUESTIONS:

1. What is tree and graph data structure?
2. What do you understand by Stack data structure?

Explanation:

```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
```

```
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

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

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

POST EXPERIMENT QUESTIONS:

1. What do you understand by DFS?
2. What is the time and space complexity of DFS?
3. What is the difference between BFS & DFS?

LAB EXPERIMENT 3

OBJECTIVE:

Write a Program to Implement Tic-Tac-Toe game using Python.

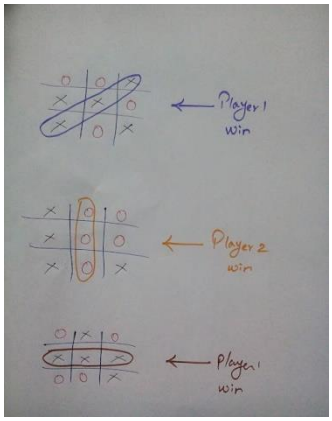
BRIEF DESCRIPTION:

The game Tic Tac Toe is also known as Noughts and Crosses or Xs and Os ,the player needs to take turns marking the spaces in a 3x3 grid with their own marks,if 3 consecutive marks (Horizontal, Vertical,Diagonal) are formed then the player who owns these moves get won.

Assume ,

Player 1 - X

Player 2 - O



So,a player who gets 3 consecutive marks first, they will win the game .

PRE EXPERIMENT QUESTIONS:

1. What do you understand by BFS?
2. What do you understand by DFS?

Explanation:

Function to print Tic Tac Toe

def print_tic_tac_toe(values):

```
    print("\n")
    print("\t     |     |")
    print("\t  {}  |  {}  |  {}".format(values[0], values[1], values[2]))
    print('\t_____|_____|_____')

    print("\t     |     |")
    print("\t  {}  |  {}  |  {}".format(values[3], values[4], values[5]))
    print('\t_____|_____|_____')

    print("\t     |     |")

    print("\t  {}  |  {}  |  {}".format(values[6], values[7], values[8]))
    print("\t     |     |")
    print("\n")


# Function to print the score-board
def print_scoreboard(score_board):
    print("\t--------------------------------")
    print("\t            SCOREBOARD       ")
    print("\t--------------------------------")

    players = list(score_board.keys())
    print("\t   ", players[0], "\t    ", score_board[players[0]])
    print("\t   ", players[1], "\t    ", score_board[players[1]])

    print("\t--------------------------------\n")
```

```
# Function to check if any player has won
def check_win(player_pos, cur_player):

    # All possible winning combinations
    soln = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 4, 7], [2, 5, 8], [3, 6, 9], [1, 5, 9], [3, 5, 7]]

    # Loop to check if any winning combination is satisfied
    for x in soln:
        if all(y in player_pos[cur_player] for y in x):

            # Return True if any winning combination satisfies
            return True
    # Return False if no combination is satisfied
    return False

# Function to check if the game is drawn
def check_draw(player_pos):
    if len(player_pos['X']) + len(player_pos['O']) == 9:
        return True
    return False

# Function for a single game of Tic Tac Toe
def single_game(cur_player):

    # Represents the Tic Tac Toe
    values = [' ' for x in range(9)]

    # Stores the positions occupied by X and O
```

```
player_pos = {'X':[], 'O':[]}


# Game Loop for a single game of Tic Tac Toe
while True:
    print_tic_tac_toe(values)

    # Try exception block for MOVE input
    try:
        print("Player ", cur_player, " turn. Which box? : ", end="")
        move = int(input())
    except ValueError:
        print("Wrong Input!!! Try Again")
        continue

    # Sanity check for MOVE inout
    if move < 1 or move > 9:
        print("Wrong Input!!! Try Again")
        continue

    # Check if the box is not occupied already
    if values[move-1] != ' ':
        print("Place already filled. Try again!!")
        continue

    # Update game information

    # Updating grid status
    values[move-1] = cur_player
```

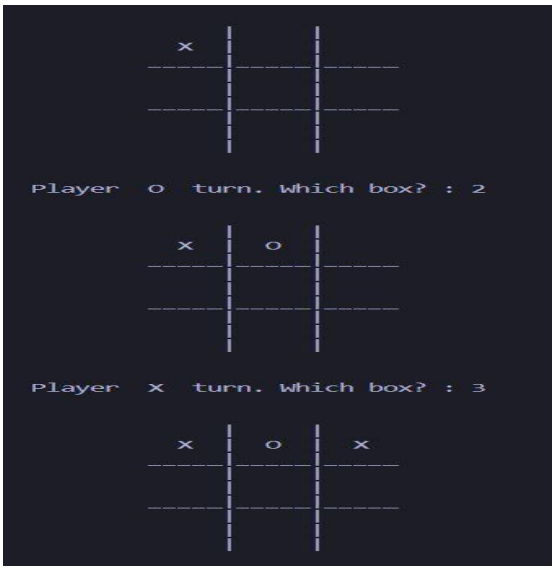
```
        # Updating player positions
        player_pos[cur_player].append(move)

        # Function call for checking win
        if check_win(player_pos, cur_player):
            print_tic_tac_toe(values)
            print("Player ", cur_player, " has won the game!!")
            print("\n")
            return cur_player

        # Function call for checking draw game
        if check_draw(player_pos):
            print_tic_tac_toe(values)
            print("Game Drawn")
            print("\n")
            return 'D'

        # Switch player moves
        if cur_player == 'X':
            cur_player = 'O'
        else:
            cur_player = 'X'

if _name_ == "_main_":

    print("Player 1")
    player1 = input("Enter the name : ")
```

```
print("\n")


print("Player 2")
player2 = input("Enter the name : ")
print("\n")


# Stores the player who chooses X and O
cur_player = player1


# Stores the choice of players
player_choice = {'X' : "", 'O' : ""}


# Stores the options
options = ['X', 'O']


# Stores the scoreboard
score_board = {player1: 0, player2: 0}
print_scoreboard(score_board)


# Game Loop for a series of Tic Tac Toe
# The loop runs until the players quit
while True:


    # Player choice Menu
    print("Turn to choose for", cur_player)
    print("Enter 1 for X")
    print("Enter 2 for O")
    print("Enter 3 to Quit")
```

```
# Try exception for CHOICE input
try:
    choice = int(input())
except ValueError:
    print("Wrong Input!!! Try Again\n")
    continue

# Conditions for player choice
if choice == 1:
    player_choice['X'] = cur_player
    if cur_player == player1:
        player_choice['O'] = player2
    else:
        player_choice['O'] = player1

elif choice == 2:
    player_choice['O'] = cur_player
    if cur_player == player1:
        player_choice['X'] = player2
    else:
        player_choice['X'] = player1

elif choice == 3:
    print("Final Scores")
    print_scoreboard(score_board)
    break
```

```
else:
    print("Wrong Choice!!!! Try Again\n")
 # Stores the winner in a single game of Tic Tac Toe
winner = single_game(options[choice-1])
        # Edits the scoreboard according to the winner
if winner != 'D' :
    player_won = player_choice[winner]
    score_board[player_won] = score_board[player_won] + 1
 print_scoreboard(score_board)
# Switch player who chooses X or O
if cur_player == player1:
    cur_player = player2
else:
    cur_player = player1
```

Output:



POST EXPERIMENT QUESTIONS:

1. How to implement tic-tac-toe using BFS?
2. How to implement tic-tac-toe using DFS?

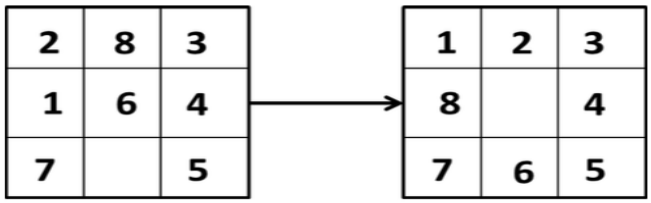
LAB EXPERIMENT 4

OBJECTIVE:

Write a Program to implement 8-Puzzle problem using Python.

BRIEF DESCRIPTION:

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.



Initial State **Goal State**

The program is to change the initial configuration into the goal configuration.

A solution to the problem is an appropriate sequence of moves, such as “move tile 5 to the right, move tile 7 to the left, move tile 6 to the down” etc…

To solve a problem, we must specify the global database, the rules, and the control strategy.

For the 8 puzzle problem that correspond to three components.

These elements are the problem states, moves and goal.

In this problem each tile configuration is a state.

The set of all possible configuration in the problem space, consists of 3,62,880 different configurations of the 8 tiles and blank space.

For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. Initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

A move transforms one problem state into another state.

PRE EXPERIMENT QUESTIONS:

1. What do you understand by BFS?

2. What do you understand by DFS?

3. What is branch and bound?

Explanation:

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
    def _init_(self):
        self.heap = []
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
        def empty(self):
        if not self.heap:
            return True
        else:
            return False
class node:
        def _init_(self, parent, mat, empty_tile_pos,
                   cost, level):
                   self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
```

```
        def __lt__(self, nxt):
                return self.cost < nxt.cost
        def calculateCost(mat, final) -> int:
                count = 0
        for i in range(n):
                for j in range(n):
                        if ((mat[i][j]) and
                                (mat[i][j] != final[i][j])):
                                count += 1
                        return count
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                        level, parent, final) -> node:
                new_mat = copy.deepcopy(mat)
        x1 = empty_tile_pos[0]
        y1 = empty_tile_pos[1]
        x2 = new_empty_tile_pos[0]
        y2 = new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
        cost = calculateCost(new_mat, final)
        new_node = node(parent, new_mat, new_empty_tile_pos,
                                        cost, level)
        return new_node
def printMatrix(mat):
        for i in range(n):
                for j in range(n):
                        print("%d " % (mat[i][j]), end = " ")
                        print()
def isSafe(x, y):
```

```
        return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):
                if root == None:
                return
                printPath(root.parent)
        printMatrix(root.mat)
        print()
def solve(initial, empty_tile_pos, final):
        pq = priorityQueue()
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
        pq.push(root)
        while not pq.empty():
                minimum = pq.pop()
                if minimum.cost == 0:
                                printPath(minimum)
                        return
                for i in range(4):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]
                                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
                                                child = newNode(minimum.mat,
                                                                minimum.empty_tile_pos,
                                                                new_tile_pos,
                                                                minimum.level + 1,
                                                                minimum, final,)
```

```
pq.push(child)
initial = [ [ 1, 2, 3 ],
                    [ 5, 6, 0 ],
                    [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],
                [ 5, 8, 6 ],
                [ 0, 7, 4 ] ]
empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)
```

Output:

```
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

POST EXPERIMENT QUESTIONS:

1. Which algorithm is used in 8-Puzzle problem?
2. How many states are there in the 8-puzzle problem?
3. What are the components of 8-puzzle problem?

LAB EXPERIMENT 5

OBJECTIVE:

Write a Program to Implement Water-Jug problem using Python.

BRIEF DESCRIPTION:

There are two jugs of volume A litre and B litre. Neither has any measuring mark on it.There is a pump that can be used to fill the jugs with water. How can you get exactly x litre of water into the A litre jug. Assuming that we have unlimited supply of water.

Note: Let's assume we have A=4 litre and B= 3 litre jugs. And we want exactly 2 Litre water into jug A (i.e. 4 litre jug) how we will do this.

The state space for this problem can be described as the set of ordered pairs of integers (x, y)

Where,

x represents the quantity of water in the 4-gallon jug x= 0,1,2,3,4

y represents the quantity of water in 3-gallon jug y=0,1,2,3

Start State: (0,0)

Goal State: (2,0)

Generate production rules for the water jug problem

We basically perform three operations to achieve the goal.

1. Fill water jug.
2. Empty water jug
3. and Transfer water jug

Production Rules:

Rule	State	Process
1	(X,Y | X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y |Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y |X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y | Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y | X+Y>=4 ^ Y>0)	(4,Y-(4-X))

		{Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y | X+Y>=3 ^X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	(X,Y | X+Y<=4 ^Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y | X+Y <=3^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Initialization:

Start State: (0,0)

Apply Rule 2:

(X,Y | Y<3) ->

(X,3)

{Fill 3-gallon jug}

Now the state is (X,3)

Iteration 1:

Current State: (X,3)

Apply Rule 7:

(X,Y | X+Y<=4 ^Y>0)

(X+Y,0)

{Pour all water from 3-gallon jug into 4-gallon jug}

Now the state is (3,0)

Iteration 2:

Current State : (3,0)

Apply Rule 2:

(X,Y | Y<3) ->

(3,3)

{Fill 3-gallon jug}

Now the state is (3,3)

Iteration 3:

Current State:(3,3)

Apply Rule 5:

(X,Y | X+Y>=4 ^ Y>0)

(4,Y-(4-X))

{Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}

Now the state is (4,2)

Iteration 4:

Current State : (4,2)

Apply Rule 3:

(X,Y | X>0)

(0,Y)

{Empty 4-gallon jug}

Now state is (0,2)

Iteration 5:

Current State : (0,2)

Apply Rule 9:

(0,2)

(2,0)

{Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Now the state is (2,0)

Goal Achieved.

PRE EXPERIMENT QUESTIONS:

1. What do you understand by BFS?

2. What do you understand by DFS?

Explanation:

```
from collections import deque
def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))
    while (len(q) > 0):
        u = q.popleft()
        if ((u[0], u[1]) in m):
            continue
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if (u[0] == target or u[1] == target):
            isSolvable = True
            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])
```

```
                    sz = len(path)
                    for i in range(sz):
                        print("(", path[i][0], ",",
                              path[i][1], ")")
                    break

            q.append([u[0], b])
            q.append([a, u[1]])
            for ap in range(max(a, b) + 1):
                c = u[0] + ap
                d = u[1] - ap
                if (c == a or (d == 0 and d >= 0)):
                    q.append([c, d])
                c = u[0] - ap
                d = u[1] + ap
                if ((c == 0 and c >= 0) or d == b):
                    q.append([c, d])
            q.append([a, 0])
            q.append([0, b])
    if (not isSolvable):
        print("Solution not possible")
if _name_ == '_main_':
    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")
    Solution(Jug1, Jug2, target)
```

Output:

```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
PS C:\Users\mayan\OneDrive\Desktop\VS Code>
```

POST EXPERIMENT QUESTIONS:

1. How do you implement a water jug problem?
2. What is the time complexity of water jug problem?
3. What are the conditions for the water jug problem in AI?
4. Which algorithm is used to solve the water jug problem?

LAB EXPERIMENT 6

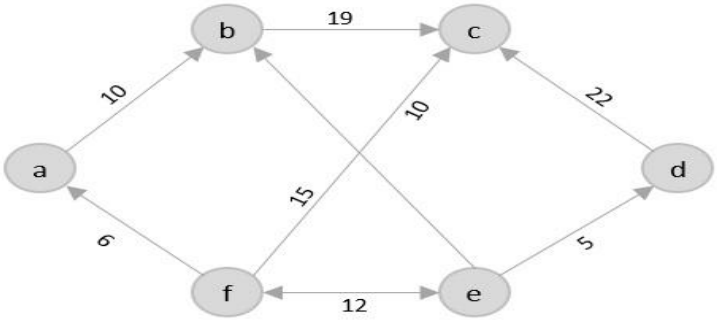
OBJECTIVE:

Write a Program to Implement Travelling Salesman Problem using Python.

BRIEF DESCRIPTION:

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the vertex 'a', they need to travel through all the remaining vertices b, c, d, e, f and get back to 'a' while making sure that the cost taken is minimum.



PRE EXPERIMENT QUESTIONS:

1. What is Brute –Force Searching?

2. What do you understand by Greedy approach?

3. What do you understand by Dynamic approach?

Explanation:

```
from sys import maxsize

from itertools import permutations

V = 4

def travellingSalesmanProblem(graph, s):

        vertex = []

        for i in range(V):
```

```
            if i != s:
                vertex.append(i)

    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        current_pathweight = 0

        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        min_path = min(min_path, current_pathweight)

    return min_path
if _name_ == "_main_":

    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

Output:

```
PS C:\Users\mayan\OneDrive\Desktop\VS Code> &
80
PS C:\Users\mayan\OneDrive\Desktop\VS Code>
```

POST EXPERIMENT QUESTIONS:

1. What are the practical applications of the travelling salesman problem?
2. What is the best algorithm for the travelling salesman problem?
3. How many possible routes are there in travelling salesman problem?
4. What is the time complexity of travelling salesman problem?

LAB EXPERIMENT 7

OBJECTIVE:

Write a Program to Implement Tower of Hanoi using Python.

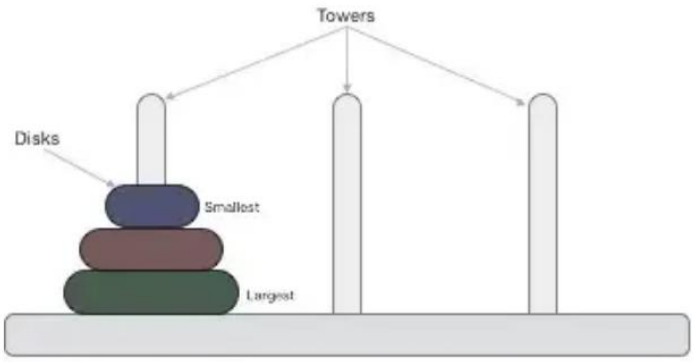
BRIEF DESCRIPTION:

Tower of Hanoi is mathematical game puzzle where we have three pile (pillars) and n numbers of disk.

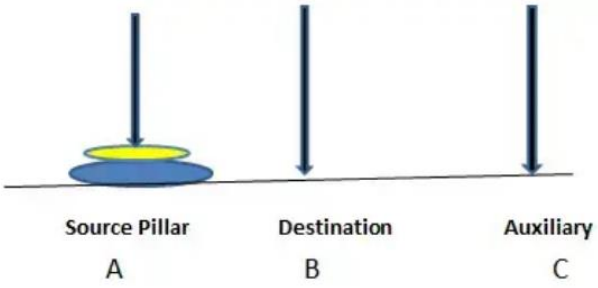
This game has some rules (Rules of game)

- Only one disk will move at a time.
- The larger disk should always be on the bottom and the smaller disk on top of it.(Even during intermediate move)
- Move only the uppermost disk.
- All disk move to destination pile from source pile.

So, here we are trying to solve that how many moves are required to solve a problem (It depends on number of disk).



When we have two disk and 3 pillars (pile, A, B, C)



In the above diagram, following the rule of the game our target is move the disks from source pile (pillar) to the destination pillar. (Let's take a look how many steps/ moves are required to make this happen).

Step1: Move small disk to the auxiliary pillar (A).

Step2: Move large disk to the Destination pillar (B).

Step3: Move small disk to the Destination pillar (B).4

So, basically when we have 2 disks we required 3 move to reach the destination.

What if we have 3, 4, 5...n disks?

Eventually, you figure out that there is some pattern to the puzzle and with each increment in disks; the pattern could be followed recursively.

Total move required to reach destination pillar is formula of moves means if we have 3 disks we required (4 moves to reach destination pillar), if 4 disks 8 moves required and so on...

PRE EXPERIMENT QUESTIONS:

1. What do you understand by Recursion?

2. What is Backtracking?

Explanation:

```
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
```

POST EXPERIMENT QUESTIONS:

1. What is the practical use of Tower of Hanoi?
2. Which data structure is used in the Tower of Hanoi?
3. Can we solve Tower of Hanoi problem without recursion?
4. Which algorithm approach is used to solve the Tower of Hanoi problem?

LAB EXPERIMENT 8

OBJECTIVE:

Write a Program to Implement Monkey Banana Problem using Python.

BRIEF DESCRIPTION:

The monkey and banana problem is a famous toy problem in artificial intelligence, particularly in logic programming and planning. A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey? The problem seeks to answer the question of whether monkeys are intelligent. Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter, or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other. Monkeys have the ability not only to remember how to hunt and gather but to learn new things, as is the case with the monkey and the bananas: despite the fact that the monkey may never have been in an identical situation, with the same artifacts at hand, a monkey is capable of concluding that it needs to make a ladder, position it below the bananas, and climb up to reach for them.

Suppose the problem is as given below −

- A hungry monkey is in a room, and he is near the door.
- The monkey is on the floor.
- Bananas have been hung from the center of the ceiling of the room.
- There is a block (or chair) present in the room near the window.
- The monkey wants the banana, but cannot reach it.

So how can the monkey get the bananas?

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case −

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

PRE EXPERIMENT QUESTIONS:

1. What is logic programming?

2. What do you understand by Planning?

Explanation:

```
from poodle import Object, schedule

from typing import Set

class Position(Object):

 def __str__(self):

 if not hasattr(self, "locname"): return "unknown"

 return self.locname

class HasHeight(Object):

 height: int

class HasPosition(Object):

 at: Position

class Monkey(HasHeight, HasPosition): pass

class PalmTree(HasHeight, HasPosition):

 def __init__(self, *args, **kwargs):

 super().__init__(*args, **kwargs)
```

```
 self.height = 2
class Box(HasHeight, HasPosition): pass
class Banana(HasHeight, HasPosition):
 owner: Monkey
 attached: PalmTree
class World(Object):
 locations: Set[Position]
p1 = Position()
p1.locname = "Position A"
p2 = Position()
p2.locname = "Position B"
p3 = Position()
p3.locname = "Position C"
w = World()
w.locations.add(p1)
w.locations.add(p2)
w.locations.add(p3)
m = Monkey()
m.height = 0 # ground
m.at = p1
box = Box()
box.height = 2
box.at = p2
p = PalmTree()
p.at = p3
b = Banana()
b.attached = p
```

```
def go(monkey: Monkey, where: Position):
 assert where in w.locations
 assert monkey.height < 1, "Monkey can only move while on the ground"
 monkey.at = where
 return f"Monkey moved to {where}"
def push(monkey: Monkey, box: Box, where: Position):
 assert monkey.at == box.at
 assert where in w.locations
 assert monkey.height < 1, "Monkey can only move the box while on the ground"
 monkey.at = where
 box.at = where
 return f"Monkey moved box to {where}"
def climb_up(monkey: Monkey, box: Box):
 assert monkey.at == box.at
 monkey.height += box.height
 return "Monkey climbs the box"
def grasp(monkey: Monkey, banana: Banana):
 assert monkey.height == banana.height
 assert monkey.at == banana.at
 banana.owner = monkey
 return "Monkey takes the banana"
def infer_owner_at(palmtree: PalmTree, banana: Banana):
 assert banana.attached == palmtree
 banana.at = palmtree.at
 return "Remembered that if banana is on palm tree, its location is where palm tree is"
def infer_banana_height(palmtree: PalmTree, banana: Banana):
 assert banana.attached == palmtree
```

```
 banana.height = palmtree.height
 return "Remembered that if banana is on the tree, its height equals tree's height"
print('\n'.join(x() for x in schedule(
 [go, push, climb_up, grasp, infer_banana_height, infer_owner_at],
 [w,p1,p2,p3,m,box,p,b],
 goal=lambda: b.owner == m)))
```

Output:

```
$ pip install poodle
$ python ./monkey.py
Monkey moved to Position B
Remembered that if banana is on the tree, its height equals tree's height
Remembered that if banana is on palm tree, its location is where palm tree is
Monkey moved box to Position C
Monkey climbs the box
Monkey takes the banana
```

POST EXPERIMENT QUESTIONS:

1. How do you solve monkey banana problem?
2. Which algorithm is used to solve monkey banana problem?

LAB EXPERIMENT 9

OBJECTIVE:

Write a Program to Implement Missionaries-Cannibals Problems using Python.

BRIEF DESCRIPTION:

In the missionaries and cannibals problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.

First let us consider that both the missionaries (M) and cannibals(C) are on the same side of the river. Left Right Initially the positions are : 0M , 0C and 3M , 3C (B) Now let's send 2 Cannibals to left of bank : 0M , 2C (B) and 3M , 1C Send one cannibal from left to right : 0M , 1C and 3M , 2C (B) Now send the 2 remaining Cannibals to left : 0M , 3C (B) and 3M , 0C Send 1 cannibal to the right : 0M , 2C and 3M , 1C (B) Now send 2 missionaries to the left : 2M , 2C (B) and 1M . 1C Send 1 missionary and 1 cannibal to right : 1M , 1C and 2M , 2C (B) Send 2 missionaries to left : 3M , 1C (B) and 0M , 2C Send 1 cannibal to right : 3M , 0C and 0M , 3C (B) Send 2 cannibals to left : 3M , 2C (B) and 0M , 1C Send 1 cannibal to right : 3M , 1C and 0M , 2C (B)' Send 2 cannibals to left : 3M , 3C (B) and 0M , 0C • Here (B) shows the position of the boat after the action is performed. Therefore all the missionaries and cannibals have crossed the river safely.

PRE EXPERIMENT QUESTIONS:

1. What is BFS searching technique?

2. What is logic programming?

Explanation:

#Python program to illustrate Missionaries & cannibals Problem

```
print("\n")

print("\tGame Start\nNow the task is to move all of them to right side of the river")

print("rules:\n1. The boat can carry at most two people\n2. If cannibals num greater than missionaries then the cannibals would eat the missionaries\n3. The boat cannot cross the river by itself with no people on board")

lM = 3          #lM = Left side Missionaries number

lC = 3          #lC = Laft side Cannibals number
```

```
rM=0           #rM = Right side Missionaries number
rC=0           #rC = Right side cannibals number
userM = 0      #userM = User input for number of missionaries for right to left side travel
userC = 0      #userC = User input for number of cannibals for right to left travel
k = 0
print("\nM M M C C C |     --- | \n")
try:
   while(True):
      while(True):
         print("Left side -> right side river travel")
         #uM = user input for number of missionaries for left to right travel
         #uC = user input for  number of cannibals for left to right travel
         uM = int(input("Enter number of Missionaries travel => "))
         uC = int(input("Enter number of Cannibals travel => "))
           if((uM==0)and(uC==0)):
             print("Empty travel not possible")
             print("Re-enter : ")
         elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
             break
         else:
             print("Wrong input re-enter : ")
      lM = (lM-uM)
      lC = (lC-uC)
      rM += uM
      rC += uC
        print("\n")
      for i in range(0,lM):
```

```
        print("M ",end="")
    for i in range(0,lC):
        print("C ",end="")
    print("| --> | ",end="")
    for i in range(0,rM):
        print("M ",end="")
    for i in range(0,rC):
        print("C ",end="")
    print("\n")
      k +=1
     if(((lC==3)and (lM == 1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM
== 1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
        print("Cannibals eat missionaries:\nYou lost the game")
          break
      if((rM+rC) == 6):
        print("You won the game : \n\tCongrats")
        print("Total attempt")
        print(k)
        break
    while(True):
        print("Right side -> Left side river travel")
        userM = int(input("Enter number of Missionaries travel => "))
        userC = int(input("Enter number of Cannibals travel => "))
              if((userM==0)and(userC==0)):
             print("Empty travel not possible")
             print("Re-enter : ")
        elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-userC)>=0)):
```

```
            break
        else:
            print("Wrong input re-enter : ")
    lM += userM
    lC += userC
    rM -= userM
    rC -= userC
     k +=1
    print("\n")
    for i in range(0,lM):
        print("M ",end="")
    for i in range(0,lC):
        print("C ",end="")
    print("| <-- | ",end="")
    for i in range(0,rM):
        print("M ",end="")
    for i in range(0,rC):
        print("C ",end="")
    print("\n")
      if(((lC==3)and (lM == 1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM
== 1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
        print("Cannibals eat missionaries:\nYou lost the game")
        break
except EOFError as e:
   print("\nInvalid input please retry !!")
```

Output:

Game Start

Now the task is to move all of them to right side of the river

rules:

1. The boat can carry at most two people

2. If cannibals num greater than missionaries then the cannibals would eat the missionaries

3. The boat cannot cross the river by itself with no people on board

M M M C C C | --- |

Left side -> right side river travel

Enter number of Missionaries travel =>

Invalid input please retry!!

POST EXPERIMENT QUESTIONS:

1. How do you solve the missionaries and cannibals problem in AI?

2. Which technique is used to solve missionaries and cannibals problem?

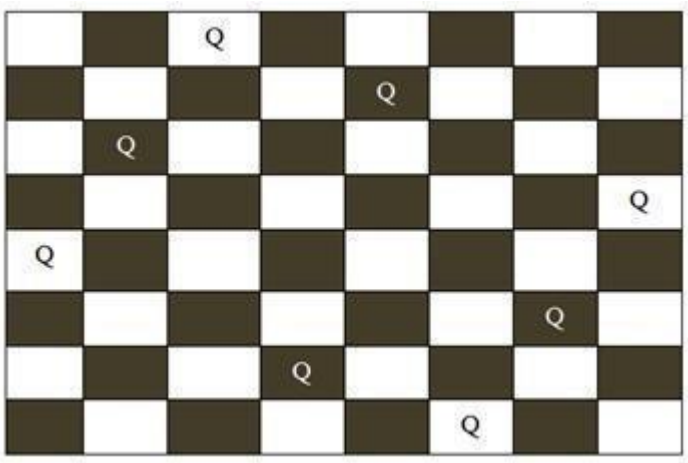
LAB EXPERIMENT 10

OBJECTIVE:

Write a Program to implement 8-Queens Problem using Python.

BRIEF DESCRIPTION:

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. There are 92 solutions. We have 8 queens and an 8x8 Chess board having al ternate black and white squares. The queens are placed on the chessboard. Any queen can attack any other queen placed on same row, or column or diagonal. We have to find the proper placement of queens on the Chess board in such a way that no queen attacks other queen".



In figure, the possible board configuration for 8-queen problem has been shown. The board has alternative black and white positions on it. The different positions on the board hold the queens.

The production rule for this game is you cannot put the same queens in a same row or same column or in same diagonal. After shifting a single queen from its position on the board, the user have to shift other queens according to the production rule.

Starting from the first row on the board the queen of their corresponding row and column are to be moved from their original positions to another position. Finally the player has to be ensured that no rows or columns or diagonals of on the table are same.

PRE EXPERIMENT QUESTIONS:

1. What is backtracking?

2. What do you understand by branch and bound?

Explanation:

```
# Python program to solve N Queen problem
global N
N = 4
def printSolution(board):
for i in range(N):
for j in range(N):
print board[i][j],
print
def isSafe(board, row, col):
# Check this row on left side
for i in range(col):
if board[row][i] == 1:
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] == 1:
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] == 1:
return False
return True
def solveNQUtil(board, col):
```

```
# base case: If all queens are placed
# then return true
if col >= N:
return True
for i in range(N):
if isSafe(board, i, col):
# Place this queen in board[i][col]
board[i][col] = 1
# recur to place rest of the queens
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
# driver program to test above function
solveNQ()
```

Output:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

POST EXPERIMENT QUESTIONS:

1. What is the best method to solve the 8 queen problem?

2. What is the best method to solve the 8 queen problem?

LAB EXPERIMENT 11

OBJECTIVE:

. Write a Program to implement Hill Climbing Algorithm.

BRIEF DESCRIPTION:

A hill-climbing algorithm is an Artificial Intelligence (AI) algorithm that increases in value continuously until it achieves a peak solution. This algorithm is used to optimize mathematical problems and in other real-life applications like marketing and job scheduling.

A hill-climbing algorithm is a local search algorithm that moves continuously upward (increasing) until the best solution is attained. This algorithm comes to an end when the peak is reached.

This algorithm has a node that comprises two parts: state and value. It begins with a non-optimal state (the hill's base) and upgrades this state until a certain precondition is met. The heuristic function is used as the basis for this precondition. The process of continuous improvement of the current state of iteration can be termed as climbing. This explains why the algorithm is termed as a hill-climbing algorithm.

A hill-climbing algorithm's objective is to attain an optimal state that is an upgrade of the existing state. When the current state is improved, the algorithm will perform further incremental changes to the improved state. This process will continue until a peak solution is achieved. The peak state cannot undergo further improvements.

A hill-climbing algorithm has four main features:

- **It employs a greedy approach:** This means that it moves in a direction in which the cost function is optimized. The greedy approach enables the algorithm to establish local maxima or minima.
- **No Backtracking**: A hill-climbing algorithm only works on the current state and succeeding states (future). It does not look at the previous states.
- **Feedback mechanism**: The algorithm has a feedback mechanism that helps it decide on the direction of movement (whether up or down the hill). The feedback mechanism is enhanced through the generate-and-test technique.
- **Incremental change**: The algorithm improves the current solution by incremental changes.

POST EXPERIMENT QUESTIONS:

1. What is backtracking?

2. What do you understand by greedy approach?

3. What is local search algorithm?

Explanation:

```
import random
```

```
import numpy as np
coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])
def generate_matrix(coordinate):
    matrix = []
    for i in range(len(coordinate)):
        for j in range(len(coordinate)):
            p = np.linalg.norm(coordinate[i]-coordinate[j])
            matrix.append(p)
    matrix = np.reshape(matrix, (len(coordinate), len(coordinate)))
    return matrix

def solution(matrix):
    points = list(range(0, len(matrix)))
    solution = []
    for i in range(0,len(matrix)):
        random_point = points[random.randint(0, len(points)-1)]
        solution.append(random_point)
        points.remove(random_point)
    return solution

def path_length(matrix, solution):
    cycle_length = 0
    for i in range(0, len(solution)):
        cycle_length += matrix[solution[i]][solution[i-1]]
    return cycle_length

def neighbours(matrix, solution):
    neighbours = []
```

```
    for i in range(0, len(solution)):
        for j in range(i+1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)

    best_neighbour = neighbours[0]
    best_path = path_length(matrix, best_neighbour)

    for neighbour in neighbours:
        current_path = path_length(matrix, neighbour)
        if current_path < best_path:
            best_path = current_path
            best_neighbour = neighbour
    return best_neighbour, best_path

def hill_climbing(coordinate):
    matrix = generate_matrix(coordinate)
    current_solution = solution(matrix)
    current_path = path_length(matrix, current_solution)
    neighbour = neighbours(matrix, current_solution)[0]
    best_neighbour, best_neighbour_path = neighbours(matrix, neighbour)

    while best_neighbour_path < current_path:
        current_solution = best_neighbour
        current_path = best_neighbour_path
        neighbour = neighbours(matrix, current_solution)[0]
```

```
    best_neighbour, best_neighbour_path = neighbours(matrix, neighbour)

  return current_path, current_solution

final_solution = hill_climbing(coordinate)
print("The solution is \n", final_solution[1])
```

Output:

```
PS C:\Users\mayan\OneDrive\Desktop\VS Code> &
The solution is
 [8, 4, 2, 1, 10, 6, 0, 5, 7, 9, 3]
PS C:\Users\mayan\OneDrive\Desktop\VS Code>
```

POST EXPERIMENT QUESTIONS:

1. Which is the best algorithm to implement hill climbing?
2. What is the time complexity of hill climbing algorithm?
3. Is backtracking possible in hill climbing algorithm?

LAB EXPERIMENT 12

OBJECTIVE:

Write a Program to implement A* Algorithm.

BRIEF DESCRIPTION:

A* is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

PRE EXPERIMENT QUESTIONS

1. What do you understand by Informed Search?

2. What is backtracking?

Explanation:

```
from collections import deque


class Graph:
   def _init_(self, adjacency_list):
      self.adjacency_list = adjacency_list
      def get_neighbour(self, v):
      return self.adjacency_list[v]


   def h(self, n):
      H = {
         'A':1,
```

```
        'B':1,
        'C':1,
        'D':1
    }
    return H[n]

def a_star_algorithm(self, start_node, stop_node):
    open_list = set([start_node])
    closed_list = set([])

    g = {}

    g[start_node] = 0

    parent = {}
    parent[start_node] = start_node

    while len(open_list) > 0:
        n = None

        for v in open_list:
            if n==None or g[v] + self.h(v) < g[n] + self.h(n):
                n = v

        if n==None:
            print('Path does not exist')
            return None
```

```
if n==stop_node:
    reconst_path = []

    while parent[n]!=n:
        reconst_path.append(n)
        n = parent[n]

    reconst_path.append(start_node)

    reconst_path.reverse()

    print('Path found: {}'.format(reconst_path))

    return reconst_path

for (m, weight) in self.get_neighbour(n):
    if m not in open_list and m not in closed_list:
        open_list.add(m)
        parent[m] = n
        g[m] = g[n] + weight

    else:
        if g[m]>g[n]+weight:
            g[m] = g[n] + weight
            parent[m] = n

            if m in closed_list:
                closed_list.remove(m)
```

```
                open_list.add(m)

        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None

adjacency_list = {
  'A': [('B', 1), ('C', 3), ('D', 7)],
  'B': [('D', 5)],
  'C': [('D', 12)]
}

graph = Graph(adjacency_list)
graph.a_star_algorithm('A', 'D')
```

Output:

```
PS C:\Users\mayan\OneDrive\Desktop\VS Code> &
Path found: ['A', 'B', 'D']
PS C:\Users\mayan\OneDrive\Desktop\VS Code>
```

POST EXPERIMENT QUESTIONS:

1. What is the A * search algorithm based on?
2. What is the heuristic used in A * algorithm?
3. What is the time complexity of A* algorithm?

This lab manual has been updated by

Ms. Renu (renu@ggnindia.dronacharya.info)

Crosschecked By HOD CSE

Please spare some time to provide your valuable feedback.