Hindi Vidya Prachar Samiti's

RAMNIRANJAN JHUNJHUNWALA COLLEGE OF ARTS, SCIENCE & COMMERCE

(EMPOWERED AUTONOMOUS COLLEGE)

AFFILIATED TO
UNIVERSITY OF MUMBAI

DEPARTMENT OF INFORMATION TECHNOLOGY
2024-2025

M.SC. (IT) PART I SEM I

PAPER RJSPIT104P – ARTIFICIAL INTELLIGENCE

Name :- Rajbhar Sudesh Dinesh SushilaDevi

Roll No. 6623

Hindi Vidya Prachar Samiti's

RAMNIRANJAN JHUNJHUNWALA COLLEGE

Ghatkopar (W), Mumbai-400 086

Certificate

This is to certify that Mr./Ms. Rajbhar Sudesh Dinesh SushilaDevi Roll No 6623 of M.Sc.(I.T.) Part-1 class has completed the required number of experiments in the subject of Artificial Intelligence in the Department of Information Technology during the academic year 2024 - 2025.

Professor In-Charge

Co-ordinator of IT Department
Prof. Bharati Bhole

College Seal & Date

Examiner

INDEX

Artificial Intelligence

Sr. No	Practical	Date
1	**Implementation of following search algorithms for the given tree:** a. Breadth First Search b. Depth First Search	Jul 15, 2024
2	**Implementation of Heuristic search algorithms:** a. A* Search b. AO*	Sep 16, 2024
3	**Implementation of** a. Tic-Tac-Toe game problem b. Water-Jug problem	Jul 22, 2024
4	**Implementation of** a. Map/Region Coloring Problem in AI. b. Construct a Maze Problem Solver	Jul 29, 2024
5	Implementation of N-Queens Problem.	Sep 16, 2024
6	Implementation of constraint satisfaction problems using Prolog.	Sep 23, 2024
7	Implementation of logic programming using Prolog.	Aug 12, 2024
8	Implementation of a fuzzy-based application using Python/R.	Sep 16, 2024

Practical No 1: Implementation of search algorithms

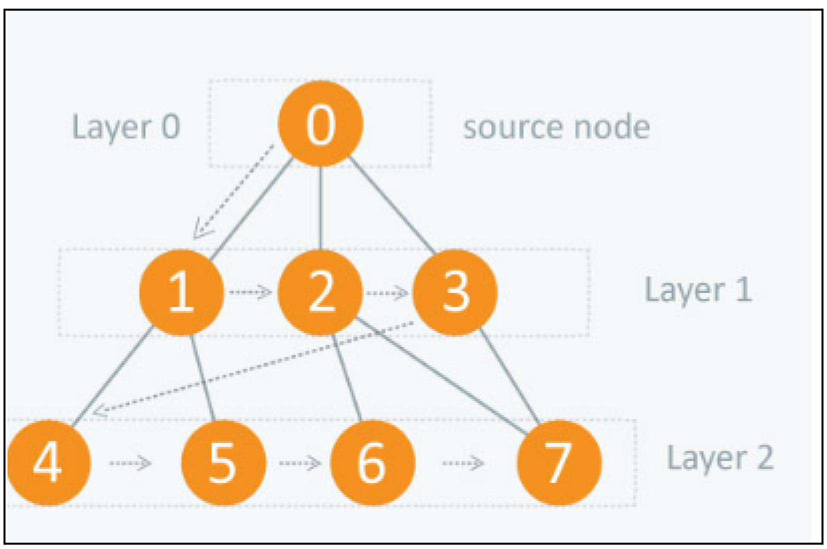
---

a. Breadth First Search

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes.

Algorithm:

- Start with the source node.
- Add that node at the front of the queue to the visited list.
- Make a list of the nodes as visited that are close to that vertex.
- And dequeue the nodes once they are visited.
- Repeat the actions until the queue is empty.



BFS.py:

```
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)

graph = {
  1: [2, 3, 4],
  2: [5, 6],
  3: [],
  4: [7, 8],
  5: [9, 10],
  6: [],
  7: [11, 12],
  8: [],
  9: [],
  10: [],
  11: [],
  12: []
}
print("Breadth-First Search Result : ", end = "")
bfs(visited, graph, 1)
```

Output:

```
[Running] python -u "d:\New folder\MSCIT_6623\BFS.py"
Breadth-First Search Result : 1 2 3 4 5 6 7 8 9 10 11 12
[Done] exited with code=0 in 0.127 seconds
```

0 ⚠ 0 ⓘ 4 0 Share Code Link Explain Code Comment Code Find Bugs

```
print("Breadth-First Search Result : ", end = "")
bfs(visited, graph, 4)
```

```
[Running] python -u "d:\New folder\MSCIT_6623\tempCodeRunnerFile.py"
Breadth-First Search Result : 4 7 8 11 12
[Done] exited with code=0 in 0.17 seconds
```

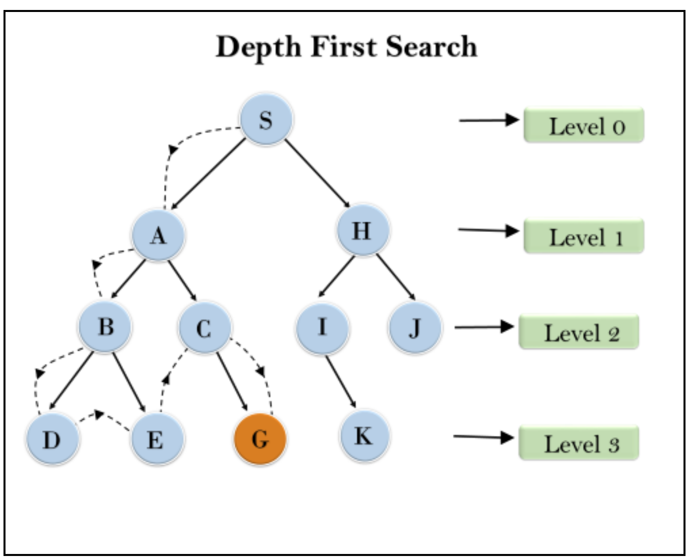
0 ⚠ 0 ⓘ 4 0 Share Code Link Explain Code Comment Code Find Bugs Code Chat

b. Depth First Search

Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Algorithm:

- Initialize an empty stack for storage of nodes, S.
- For each vertex u, define u.visited to be false.
- Push the root (first node to be visited) onto S.
- While S is not empty:
-     Pop the first element in S, u.
-     If u.visited = false, then:
-         U.visited = true
-         for each unvisited neighbor w of u:
-             Push w into S.
- End process when all nodes have been visited.



DFS.py:

```python
graph = {
    '1': ['2', '3', '4'],
    '2': ['5', '6'],
    '3': [],
    '4': ['7', '8'],
    '5': ['9', '10'],
    '6': [],
    '7': ['11', '12'],
    '8': [],
    '9': [],
    '10': [],
    '11': [],
    '12': []
}

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

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, '1')  # Function calling
```

```
[Running] python -u "d:\New folder\MSCIT_6623\tempCodeRunnerFile.py"
Following is the Depth-First Search
1 2 5 9 10 6 3 4 7 11 12 8
[Done] exited with code=0 in 0.156 seconds
```

Practical No 2: Implementation of Heuristic search algorithms

---

a. A* Search

- The most widely known form of best-first search is called A∗ Search
- It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

  f(n) = g(n) + h(n) **.**

  Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

  f(n) = estimated cost of the cheapest solution through n .
- A∗ search is both complete and optimal.

Astar.py

```
import heapq
import math

def heuristic_distance(point1, point2):
    #using manhaten distance
    #return abs(point1[0] - point2[0]) + abs(point1[1] -
point2[1])
    #using eucledian distance
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1]
- point2[1]) ** 2)

def a_star(grid, start, goal):
    open_list = [(0, start)]
    came_from = {}
    g_score = {node: float('inf') for node in grid}
    g_score[start] = 0

    while open_list:
        _, current = heapq.heappop(open_list)
```

```
        if current == goal:
            path = []
            while current in came_from:
                path.insert(0, current)
                current = came_from[current]
            path.insert(0, start)
            return path

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            neighbor = current[0] + dx, current[1] + dy
            if neighbor in grid:
                tentative_g_score = g_score[current] + 1
                if tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    heapq.heappush(open_list,
(g_score[neighbor] + heuristic_distance(neighbor, goal),
neighbor))

    return None  # No path found

# Example grid (dict with (x, y) as keys)
grid = {
    (0, 0), (0, 1), (0, 2),(0,3),
    (1, 0), (1, 1), (1, 2),(1,3),
    (2, 0), (2, 1), (2, 2),(2,3)
}

start = (1, 0)
goal = (0, 3)
path = a_star(grid, start, goal)

if path:
```

```python
    print("Path found:", path)
else:
    print("No path found.")
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR

```
[Running] python -u "e:\New folder\MSCIT_6623\AI\Astar.py"
Path found: [(1, 0), (1, 1), (1, 2), (0, 2), (0, 3)]

[Done] exited with code=0 in 0.213 seconds
```

b. AO*

- AO Search* is an extension of the A* algorithm
- The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept.
- The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.

AOstar.py

```
import heapq

def ao_star(start, goal, heuristic, neighbors):
    open_list = [(0, start)]
# Priority queue initialized with the start node
    g_costs = {start: 0}
# Dictionary to store the cost of the shortest path to each
node
    came_from = {}
 # Dictionary to reconstruct the path

    while open_list:
        _, current = heapq.heappop(open_list)

        if current == goal:
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            return list(reversed(path))
# Return the path from start to goal

        for neighbor in neighbors(current):
            tentative_g = g_costs[current] + 1
```

```
# Assuming uniform cost for each step
            if tentative_g < g_costs.get(neighbor,
float('inf')):
                g_costs[neighbor] = tentative_g
                f_cost = tentative_g + heuristic(neighbor,
goal)
                came_from[neighbor] = current
                heapq.heappush(open_list, (f_cost, neighbor))

    return None
# Return None if no path is found

# Example usage:

def heuristic(point1, point2):
    # Manhattan distance heuristic for grid-based pathfinding
    return abs(point1[0] - point2[0]) + abs(point1[1] -
point2[1])

def neighbors(node):
    # Example neighbor function for a grid (4-connected)
    x, y = node
    return [(x + dx, y + dy) for dx, dy in [(-1, 0), (1, 0),
(0, -1), (0, 1)]]

# Example grid and start/goal points
start = (0, 0)
goal = (2, 2)

path = ao_star(start, goal, heuristic, neighbors)
print("Path found:", path)
```

Output:

```
[Done] exited with code=0 in 0.213 seconds

[Running] python -u "e:\New folder\MSCIT_6623\AI\AOstar.py"
Path found: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]

[Done] exited with code=0 in 0.152 seconds
```

Practical No 3: Implementation of Goal State Environment.

---

a.Tic-Tac-Toe game problem

Description

Tic-Tac-Toe is a two-player game played on a 3x3 grid. Players take turns placing their markers (X or O) in empty cells. The goal is to get three of their markers in a row, either horizontally, vertically, or diagonally.

Steps:

1. Initialize the Game Board: Create a 3x3 grid to represent the game board and set all cells to empty.
2. Set Current Player: Start with Player X.
3. Game Loop:Display the current state of the board.
4. End Game: If a player wins or if the game is a draw, display the result and terminate the game.

1. **Initialize the Board:**
   - Create a 3x3 grid initialized with empty spaces to represent an empty board.
2. **Define Winning Conditions:**
   - Identify all possible ways to win the game, including rows, columns, and diagonals.
3. **Check for Winner:**
   - After each move, check if the current player has met any of the winning conditions.
4. **Check if Board is Full:**
   - Determine if there are any empty spaces left on the board.
5. **Play the Game:**
   - Alternate turns between Player X and Player O.
   - Display the board after each move.
   - Prompt the current player to enter their move (row and column).
   - Validate the move to ensure it is within the board and the chosen cell is empty.
   - Place the marker on the board if the move is valid.
6. **End the Game:**
   - If a player wins, announce the winner and display the final board state.
   - If the board is full and there is no winner, declare the game a tie.

Code:

```
# Function to print the Tic Tac Toe board
def print_board(board):
    for row in board:
        print(" | ".join(row))  # Join elements of each row with ' | ' and print
        print("-" * 9)          # Print a horizontal line as a separator

# Function to check if a player has won
def check_winner(board, player):
    # Define all possible winning combinations on the board
    win_conditions = [
        [board[0][0], board[0][1], board[0][2]],  # Top row
        [board[1][0], board[1][1], board[1][2]],  # Middle row
        [board[2][0], board[2][1], board[2][2]],  # Bottom row
        [board[0][0], board[1][0], board[2][0]],  # Left column
        [board[0][1], board[1][1], board[2][1]],  # Middle column
        [board[0][2], board[1][2], board[2][2]],  # Right column
        [board[0][0], board[1][1], board[2][2]],  # Diagonal from top-left to
bottom-right
        [board[0][2], board[1][1], board[2][0]]   # Diagonal from top-right to
bottom-left
    ]
    # Check if any of the win conditions are fulfilled by the player
    return [player, player, player] in win_conditions

# Function to check if the board is full
def is_board_full(board):
    # Return True if there are no empty spaces (' ') left on the board
    return all(cell != ' ' for row in board for cell in row)

def tic_tac_toe():
    board = [[' ']*3 for i in range(3)]  # Initialize the board with empty spaces
    players = ['X', 'O']  # Define the players ('X' goes first, 'O' goes second)
```

```
    turn = 0  # Initialize turn counter

    # Continue the game until there's a winner or the board is full
    while not (check_winner(board, 'X') or check_winner(board, 'O')) and not
is_board_full(board):
        # Print the current state of the board
        print_board(board)
        # Determine whose turn it is based on the turn counter
        current_player = players[turn % 2]
        print(f"Player {current_player}'s turn.")

        # Prompt the current player to input their move
        while True:
            try:
                row, col = map(int, input("Enter row and column numbers (e.g., 0 0):
").split())
                # Check if the chosen position is empty (' ')
                if board[row][col] == ' ':
                    break  # Valid move, exit the loop
                else:
                    print("That position is already taken! Try again.")
            except (ValueError, IndexError):
                print("Invalid input! Please enter valid row and column numbers.")

        # Place the current player's marker ('X' or 'O') on the board
        board[row][col] = current_player
        turn += 1  # Increment the turn counter

    # Game ended, print the final state of the board
    print_board(board)

    # Check and print the result of the game
    if check_winner(board, 'X'):
```

```python
        print("Player X wins!")
    elif check_winner(board, 'O'):
        print("Player O wins!")
    else:
        print("It's a tie!")

# Entry point of the program
if __name__ == "__main__":
    tic_tac_toe()
```

OUTPUT:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X's turn.
Enter row and column numbers (e.g., 0 0): 0 0
X |   |
---------
  |   |
---------
  |   |
---------
Player O's turn.
Enter row and column numbers (e.g., 0 0): 1 1
X |   |
---------
  | O |
---------
  |   |
---------
Player X's turn.
Enter row and column numbers (e.g., 0 0): 0 1
X | X |
---------
  | O |
---------
  |   |
---------
Player O's turn.
Enter row and column numbers (e.g., 0 0): 2 1
X | X |
---------
  | O |
---------
  | O |
---------
Player X's turn.
Enter row and column numbers (e.g., 0 0): 0 2
X | X | X
---------
  | O |
```

```
---------
  | O |
---------
Player X wins!
PS D:\New folder\MSCIT_6623\AI>
```

b. Water-Jug Problem

Given:

Two jugs with capacities **jug1** and **jug2** (denoted as **cap1** and **cap2**).
An integer **target** that represents the amount of water you want to measure.

Objective:

Determine whether it is possible to measure exactly **target** units of water using these two jugs, and if possible, describe a sequence of steps to achieve this measurement.

Jug Operations:

- You can fill either jug completely to its capacity.
- You can empty either jug completely.
- You can pour water from one jug into the other until the first jug is empty or the second jug is full.

Initial State:

Both jugs are initially empty.

Goal:

To measure exactly **target** units of water in either of the jugs.

Example:

cap1 = 4 (capacity of the first jug)
cap2 = 3 (capacity of the second jug)
target = 2 (the amount of water to measure)

defaultdict: A defaultdict is a dictionary that returns a default value if you try to access a key that doesn't exist.

Example :

visited = defaultdict(lambda: False)

visited is initialized as a defaultdict with a default value of False for any key that doesn't exist. This is done using lambda: False.

Algorithm for Water Jug Problem

1. **Setup:**
   - Define the capacities of the two jugs and the target amount of water.
   - Create a system to track which states (amounts of water in the jugs) have been processed.
2. **Recursive Function (`waterJugSolver`):**
   - **Check Solution:** If one jug has the target amount and the other is empty, print the state and indicate success.
   - **Process State:** If the state hasn't been processed before:
     - Print the state.
     - Mark it as processed.
     - **Try Actions:**
       - Empty either jug.
       - Fill either jug.
       - Pour water between jugs in both directions.
   - **Skip Processed States:** Return false if the state has been processed before.
3. **Start Search:**
   - Begin with both jugs empty and explore all possible states.
4. **Display Results:**
   - Print the steps as they are discovered.

Code:

```
from collections import defaultdict

# Jug capacities
jug1, jug2 = 4, 3

# target amount
```

```python
aim = 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):
    # Check if we've reached the target amount in either jug with the other empty
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # If this state has not been visited yet
    if not visited[(amt1, amt2)]:
        print(amt1, amt2)  # Print the current state
        visited[(amt1, amt2)] = True

        # Recursively try all possible actions
        return (
            waterJugSolver(0, amt2) or   # Empty jug1
            waterJugSolver(amt1, 0) or   # Empty jug2
            waterJugSolver(jug1, amt2) or # Fill jug1
            waterJugSolver(amt1, jug2) or # Fill jug2
            waterJugSolver(amt1 + min(amt2, jug1 - amt1),
                           amt2 - min(amt2, jug1 - amt1)) or # Pour from jug2 to jug1
            waterJugSolver(amt1 - min(amt1, jug2 - amt2),
                           amt2 + min(amt1, jug2 - amt2)) # Pour from jug1 to jug2
        )
    else:
        return False


print("Steps: ")
waterJugSolver(0, 0)
```

Output:

```python
# Jug capacities
jug1, jug2 = 4, 3
# target amount
aim = 2
```

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
PS D:\New folder\MSCIT_6623\AI>
```

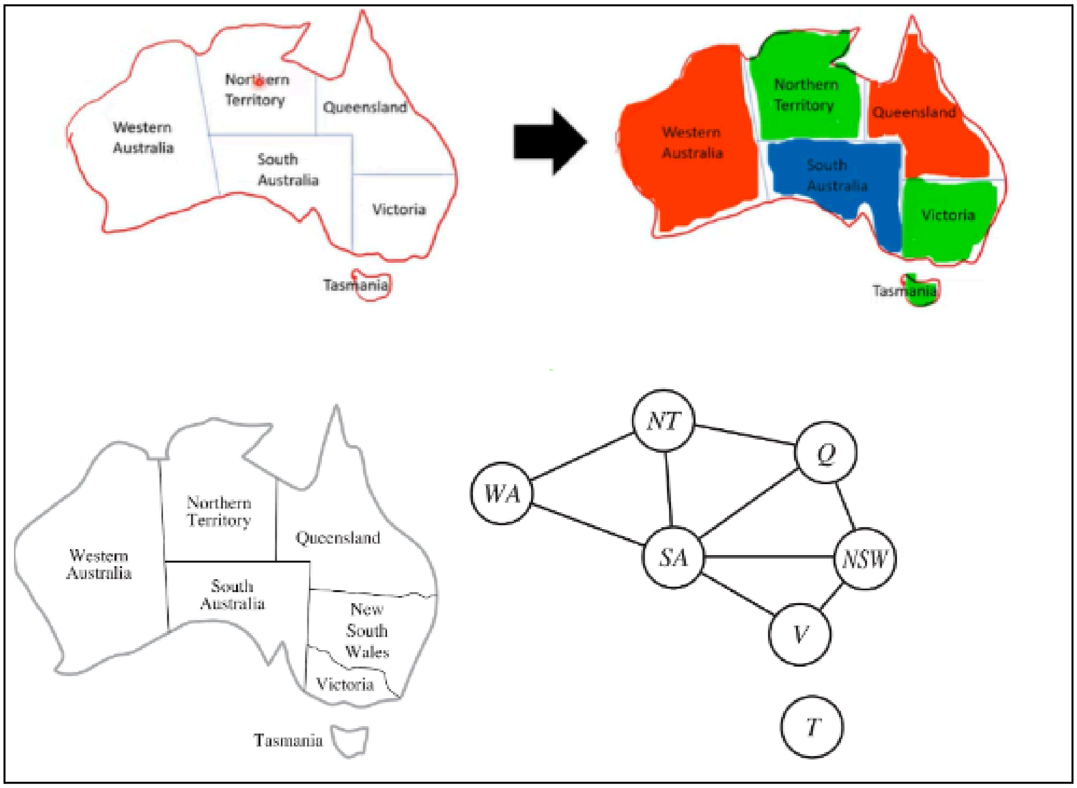
```python
# Jug capacities
jug1, jug2 = 6, 3
# target amount
aim = 4
```

```
Steps:
0 0
6 0
6 3
0 3
3 0
3 3
PS D:\New folder\MSCIT_6623\AI>
```

Practical No 4 : Implementation of AI.

---

a.Map/Region Coloring Problem in AI.



Pre requisites: pip install simpleai

Code:

```
from simpleai.search import CspProblem, backtrack

# Define the constraint function
def different_colors_constraint(names, values):
    return values[0] != values[1]
def main():
    variables = ['A', 'B', 'C'] # Define variables
    # Define domains (color choices)
    domains = {
        'A': ['red', 'green', 'blue'],
        'B': ['red', 'green', 'blue'],
        'C': ['red', 'green', 'blue']
    }
    # Define constraints
    constraints = [
        (('A', 'B'), different_colors_constraint),
        (('A', 'C'), different_colors_constraint),
        (('B', 'C'), different_colors_constraint)
    ]
    # Create CSP problem
    problem = CspProblem(variables, domains, constraints)
    # Solve the problem
    solution = backtrack(problem)
    print("Solution:", solution)
if __name__ == "__main__":
    main()
```

Output:

```
[Done] exited with code=1 in 0.149 seconds

[Running] python -u "d:\New folder\MSCIT_6623\AI\mapcolor.py"
Solution: {'A': 'red', 'B': 'green', 'C': 'blue'}

[Done] exited with code=0 in 0.277 seconds
```

b.Construct a Maze Problem Solver .

What is a Maze Problem?
A Maze Problem refers to an algorithm or system designed to find a path from a start point to an end point within a maze, connected in a complex pattern. The challenge lies in determining an optimal route, avoiding dead ends, and navigating through obstacles.

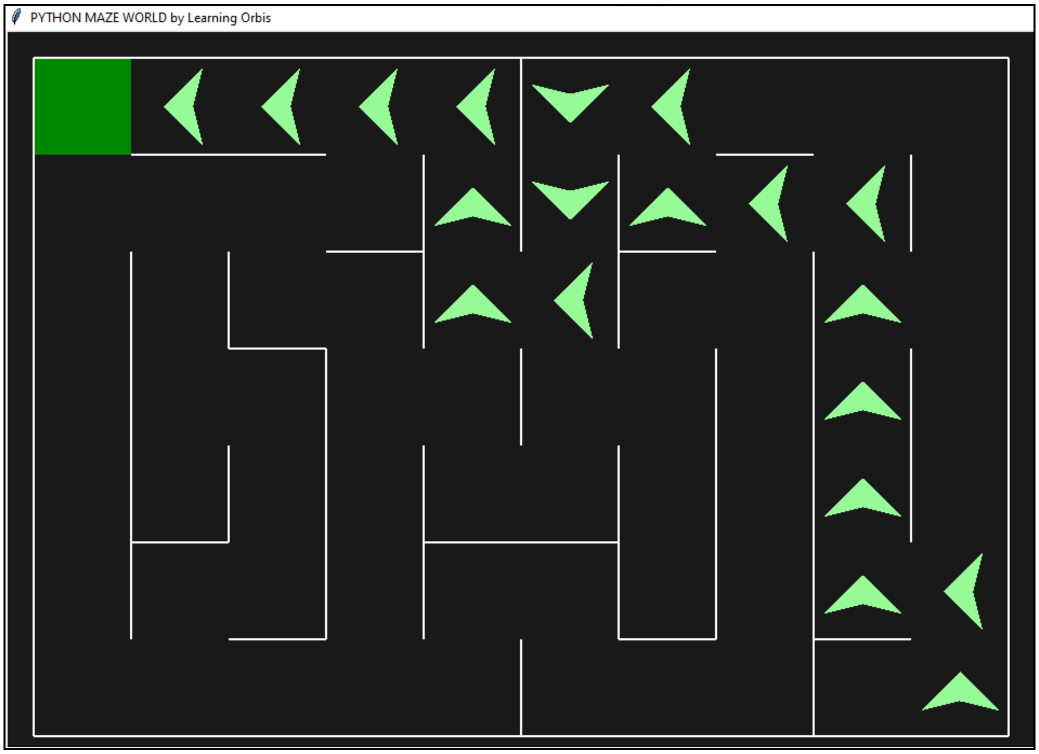
Goal: To efficiently move from the starting point to the destination, solving the maze.

Challenges:
Exploration: differentiate between valid paths and dead ends.
Efficiency: minimize the time to reach the goal.
Optimality: finding the shortest path.
Handling Loops: avoid revisiting nodes unnecessarily

pip install pyamaze

```
from pyamaze import maze,agent,COLOR
m=maze(7,10)
m.CreateMaze( pattern='v', loopPercent=40, theme=COLOR.dark)
a=agent(m,filled=True,shape = 'arrow', footprints=True,
color='green')
m.tracePath({a:m.path})
m.run()
```

Output:


PYTHON MAZE WORLD by Learning Orbis

Practical No 5 : Implementation of N-Queens Problem.

---

a.Queens N Problem

The N-queens problem is a chessboard problem that involves placing N queens on an NxN chessboard so that no two queens attack each other:

Goal

Place N queens on an NxN chessboard so that no two queens are under attack along any row, column, or diagonal

Difficulty

The N-queens problem is challenging in algorithm design and has been proven NP-complete

Origin

The original eight-queens problem was first posed in 1848 by German chess player Bezzel in the Berliner Schachzeitung (Berlin Chess Newspaper)

Solution methods

Some methods for solving the N-queens problem include genetic algorithms, backtracking, and recursion

Code:

```
def is_safe(board, row, col):
    # Check the left side of the current row
    for i in range(col):
        if board[row][i] == 1:
            return False
    # Check upper diagonal on the left
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Check lower diagonal on the left
    for i, j in zip(range(row, len(board), 1), range(col, -1,
-1)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(board, col):
    if col >= len(board):
        return True
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_n_queens(board, col + 1):
                return True
            board[i][col] = 0
    return False

def n_queens(n):
    board = [[0] * n for _ in range(n)]
    if not solve_n_queens(board, 0):
        print("No solution found.")
        return
```

```python
    for row in board:
        print(" ".join(["Q" if cell == 1 else "." for cell in 
row]))
if __name__ == "__main__":
    n = 8  # Change this to the desired board size
    n_queens(n)
```

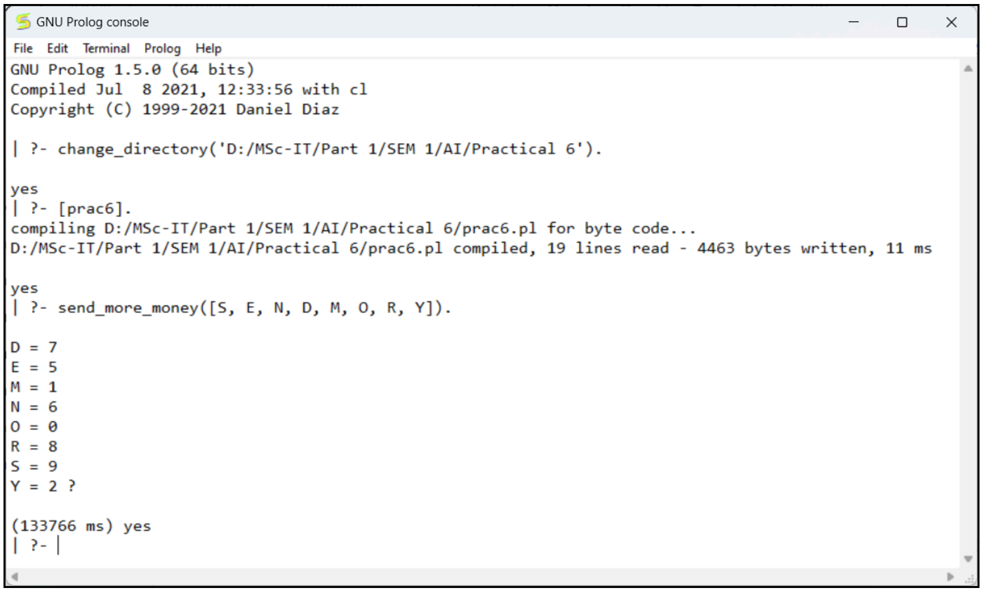
Output:

```
[Running] python -u "e:\New folder\MSCIT_6623\AI\queen.py"
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

[Done] exited with code=0 in 0.304 seconds
```

Practical No 6 : Implementation of constraint satisfaction problems using Prolog.

Code:

```
% Define the constraint that ensures no two variables have the same digit.
all_different([]).

all_different([H|T]) :- \+ member(H, T), all_different(T).

% Define the main predicate for solving the puzzle.

send_more_money([S, E, N, D, M, O, R, Y]) :-
    Digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    member(S, Digits), S > 0,
    member(E, Digits),
    member(N, Digits),
    member(D, Digits),
    member(M, Digits), M > 0,
    member(O, Digits),
    member(R, Digits),
    member(Y, Digits),
    all_different([S, E, N, D, M, O, R, Y]),
    1000 * S + 100 * E + 10 * N + D +
    1000 * M + 100 * O + 10 * R + E =:=
    10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```


GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul 8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz
| ?- change_directory('D:/MSc-IT/Part 1/SEM 1/AI/Practical 6').
yes
| ?- [prac6].
compiling D:/MSc-IT/Part 1/SEM 1/AI/Practical 6/prac6.pl for byte code...
D:/MSc-IT/Part 1/SEM 1/AI/Practical 6/prac6.pl compiled, 19 lines read - 4463 bytes written, 11 ms
yes
| ?- send_more_money([S, E, N, D, M, O, R, Y]).
D = 7
E = 5
M = 1
N = 6
O = 0
R = 8
S = 9
Y = 2 ?
(133766 ms) yes
| ?-

Practical No 7 : Implementation of logic programming using Prolog.

Define apple, orange, banana, grapes etc… as fruits
Eg- fruits(apple).
Define tomato, chilli, potato, capsicum etc… as veg
Eg- veg(tomato).
Define some fruits as sweet and some fruits as sour.
Eg- sweet(apple).
Eg- sour(grapes).
Write two rules for stating 'I like sweet fruits' and 'i don't like sour fruits'
Eg. like(X) :- fruit(X), sweet(X)
dont_like(X):-fruit(X), sour(X)
Query- which fruit you like or don't like??

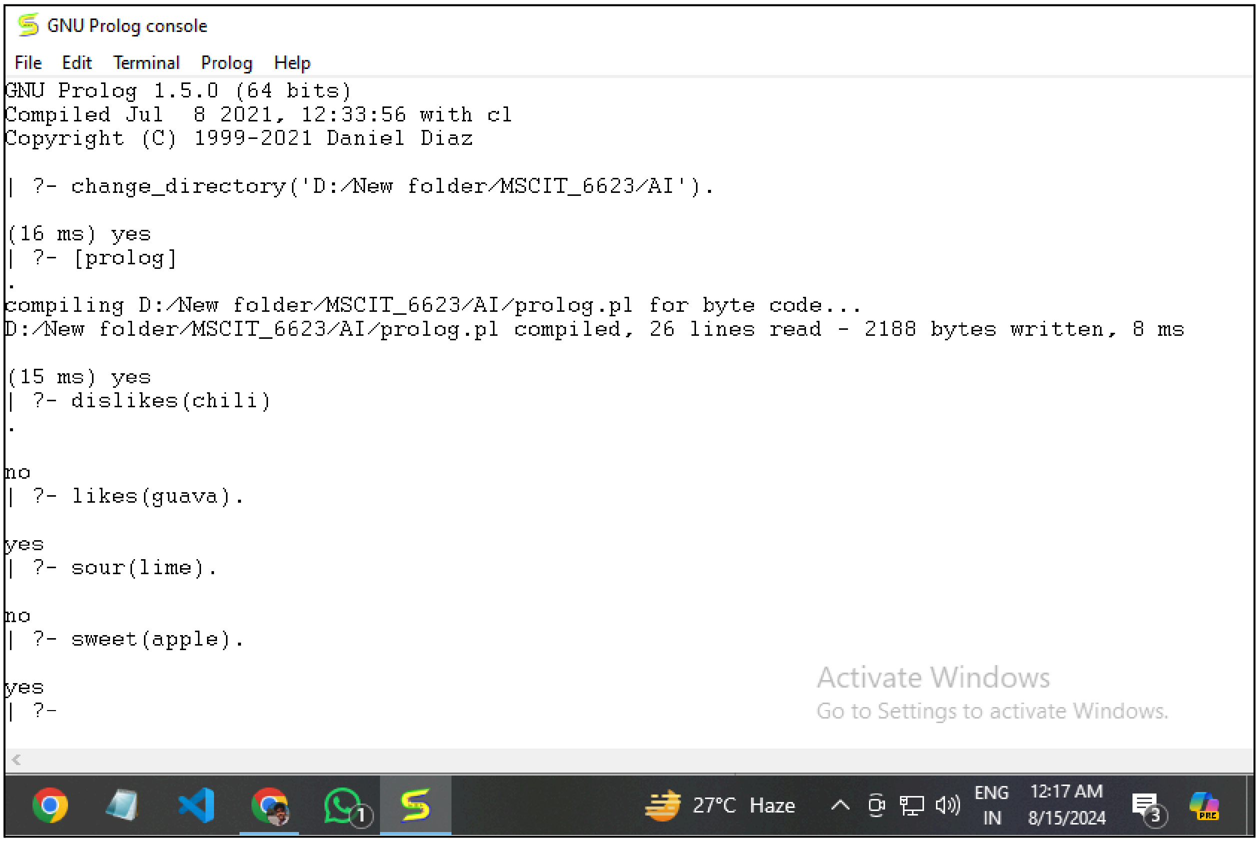
Rules of prolog :

- Statement terminated using . (period operator / full stop)
- To compile a file give cmd like [filename].

prolog.pl

```
fruits(apple).
fruits(kiwi).
fruits(pomegranate).
fruits(strawberry).
fruits(orange).
fruits(grapes).
fruits(bananas).
fruits(guava).

sweet(guava).
sweet(kiwi).
sweet(pomegranate).
sweet(strawberry).
sweet(banana).
sweet(apple).

sour(grapes).
sour(orange).

veg(tomato).
```

```
veg(chilli).
veg(potato).
veg(capsicum).

likes(X) :- fruits(X),sweet(X).
dislikes(Y) :- fruits(Y),sour(Y).
```



Practical No 8 : Implementation of a fuzzy-based application (Python / R)

---

pip install skfuzzy
pip install -U scikit-fuzzy
Pip install scipy
pip install networkx

a.Temperature Problem

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Create input and output variables
temperature = ctrl.Antecedent(np.arange(0, 101, 1),
'temperature')
fan_speed = ctrl.Consequent(np.arange(0, 101, 1), 'fan_speed')

# Define fuzzy sets for temperature
temperature['cold'] = fuzz.trimf(temperature.universe, [0, 0,
50])
temperature['warm'] = fuzz.trimf(temperature.universe, [0, 50,
100])
temperature['hot'] = fuzz.trimf(temperature.universe, [50, 100,
100])

# Define fuzzy sets for fan_speed
fan_speed['low'] = fuzz.trimf(fan_speed.universe, [0, 0, 50])
fan_speed['medium'] = fuzz.trimf(fan_speed.universe, [0, 50,
100])
fan_speed['high'] = fuzz.trimf(fan_speed.universe, [50, 100,
100])

# Define rules
```

```python
rule1 = ctrl.Rule(temperature['cold'], fan_speed['low'])
rule2 = ctrl.Rule(temperature['warm'], fan_speed['medium'])
rule3 = ctrl.Rule(temperature['hot'], fan_speed['high'])

# Create the control system
fan_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

# Create a simulation
fan_sim = ctrl.ControlSystemSimulation(fan_ctrl)

# Input temperature value
temperature_input = 15

# Set the input temperature
fan_sim.input['temperature'] = temperature_input

# Compute the fan speed
fan_sim.compute()

# Get the fan speed value
fan_speed_output = fan_sim.output['fan_speed']

print(f"For a temperature of {temperature_input} degrees:")
print(f"Fan Speed: {fan_speed_output:.2f}%")
```

```
========================= RESTART: D:\6223\AI\fuzzy.py =========================
For a temperature of 15 degrees:
Fan Speed: 37.56%
```

Result ENG 10:47 23-09-2024

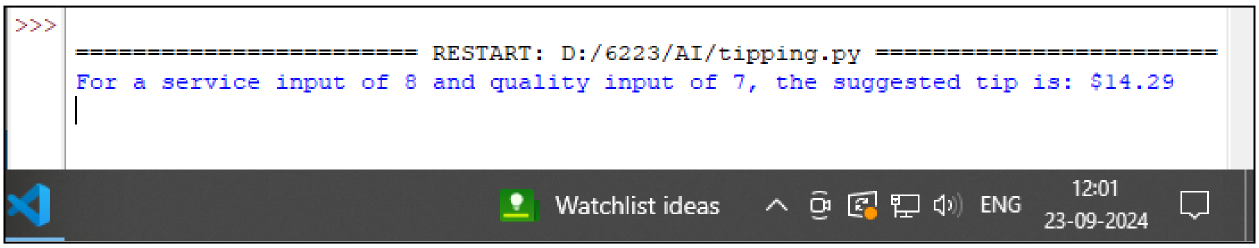
b.Tipping Problem

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt


# Create fuzzy variables outside the function
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-generate membership functions for quality and service
quality.automf(3)
service.automf(3)

# Define fuzzy sets for tips
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# Define fuzzy rules
rule1 = ctrl.Rule(quality['poor'] | service['poor'],
tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'],
tip['high'])

# New rules
rule4 = ctrl.Rule(service['good'] & quality['average'],
tip['medium'])
rule5 = ctrl.Rule(service['average'] & quality['poor'],
tip['low'])
```

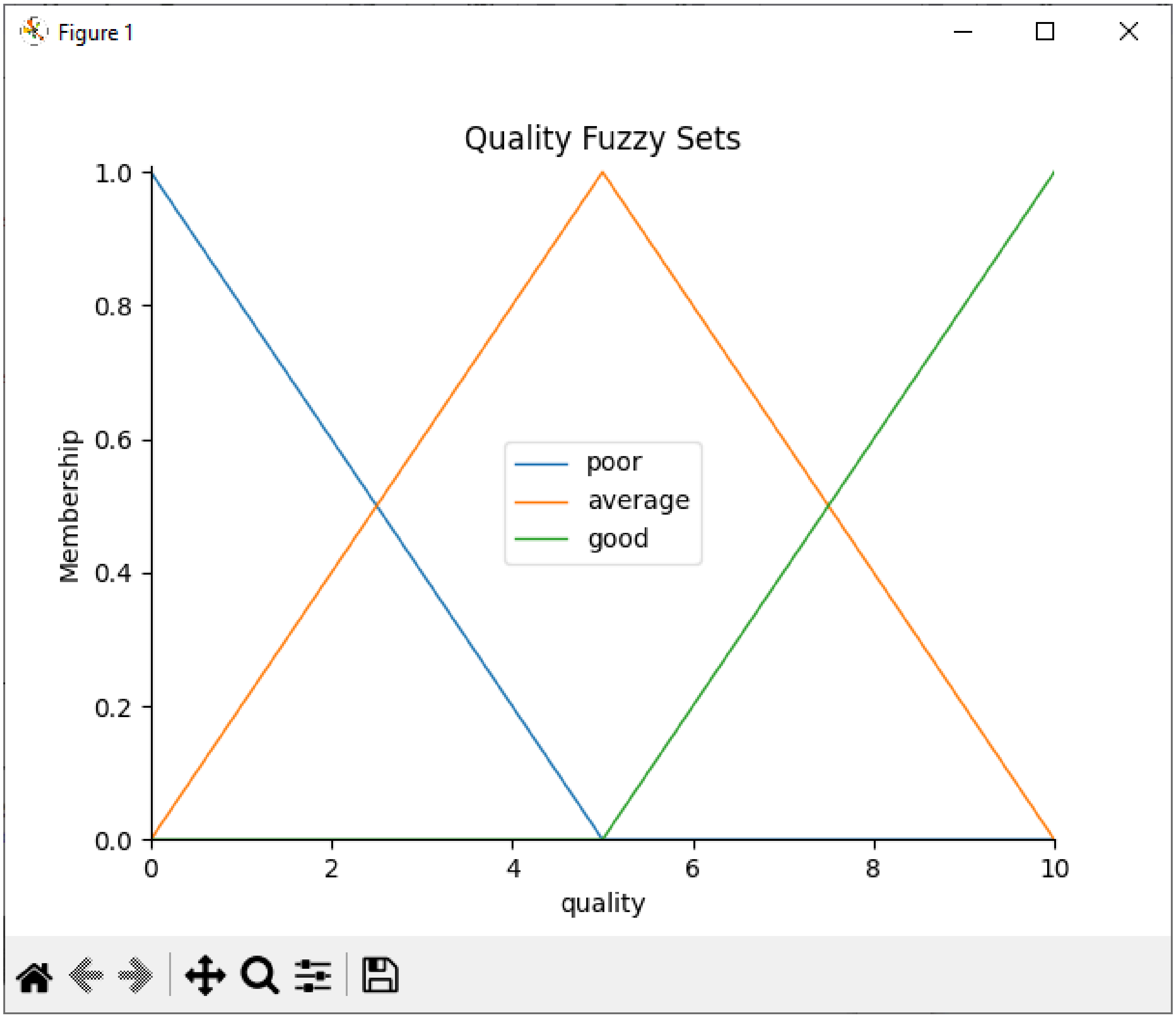
```
rule6 = ctrl.Rule(service['good'] & quality['good'],
tip['high'])

# Create control system
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4,
rule5, rule6])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

def compute_tip(service_input, quality_input):
    # Set input values
    tipping.input['quality'] = quality_input
    tipping.input['service'] = service_input

    # Compute tip
    tipping.compute()
    return tipping.output['tip']


# Example usage
service_input = 8  # Service rating (0-10)
quality_input = 7  # Quality rating (0-10)

tip_output = compute_tip(service_input, quality_input)
print(f"For a service input of {service_input} and quality
input of {quality_input}, the suggested tip is:
${tip_output:.2f}")

# Visualizing fuzzy sets
quality.view()
plt.title("Quality Fuzzy Sets")
plt.show()

service.view()
plt.title("Service Fuzzy Sets")
```

```
plt.show()

tip.view()
plt.title("Tip Fuzzy Sets")
plt.show()
```

Output:



```
>>>
======================== RESTART: D:/6223/AI/tipping.py ========================
For a service input of 8 and quality input of 7, the suggested tip is: $14.29
```



Code 2:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define the variables
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 100, 1), 'tip')
```

```
# Auto membership functions
quality.automf(3)
service.automf(3)

# Tip membership functions
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 30])
tip['medium'] = fuzz.trimf(tip.universe, [0, 30, 60])
tip['high'] = fuzz.trimf(tip.universe, [30, 60, 100])

# Define fuzzy rules
rules = [
    ctrl.Rule(quality['good'] & service['good'], tip['high']),
    ctrl.Rule(quality['good'] & service['average'],
tip['medium']),
    ctrl.Rule(quality['average'] & service['good'],
tip['medium']),
    ctrl.Rule(quality['average'] & service['average'],
tip['medium']),
    ctrl.Rule(quality['poor'] & service['poor'], tip['low']),
    ctrl.Rule(quality['poor'] & service['average'],
tip['low']),
    ctrl.Rule(quality['average'] & service['poor'],
tip['low']),
]

# Create control system
tip_ctrl = ctrl.ControlSystem(rules)
tip_simulation = ctrl.ControlSystemSimulation(tip_ctrl)

# Example input
tip_simulation.input['quality'] = 10  # Good
tip_simulation.input['service'] = 10   # Good

# Compute the tip
```

```
tip_simulation.compute()

# Output
print(f'Tip amount: {tip_simulation.output["tip"]}')
tip.view(sim=tip_simulation)
```

```
>>>
============================================================================ RESTART
: D:/6223/AI/tip2.py ===============================================================
============
Tip amount: 63.3204715969989
>>>
```