RN SHETTY TRUST®

RNS INSTITUTE OF TECHNOLOGY

Autonomous Institution, Affiliated to VTU, Recognized by GOK, Approved by AICTE
(NAAC 'A+ Grade' Accredited, NBA Accredited (UG - CSE, ECE, ISE, EIE and EEE)
Channasandra, Dr. Vishnuvardhan Road, Bengaluru - 560 098
Ph:(080)28611880,28611881 URL: www.rnsit.ac.in

DEPARTMENT OF CSE (AI&ML)

ARTIFICIAL INTELLIGENCE LAB MANUAL

(BAD402)

(As per Visvesvaraya Technological University Course type- IPCC)

Compiled by

DEPARTMENT OF CSE (AI&ML)

R N S Institute of Technology

Bengaluru-98

Name: _____

USN: _____

RN SHETTY TRUST®

RNS INSTITUTE OF TECHNOLOGY

Autonomous Institution, Affiliated to VTU, Recognized by GOK, Approved by AICTE
(NAAC 'A+ Grade' Accredited, NBA Accredited (UG - CSE, ECE, ISE, EIE and EEE)
Channasandra, Dr. Vishnuvardhan Road, Bengaluru - 560 098
Ph:(080)28611880,28611881 URL: www.rnsit.ac.in

Estd : 2001

DEPARTMENT OF CSE (AI&ML)

VISION OF THE DEPARTMENT

Empowering AI & ML Engineers to seamlessly integrate society and technology

MISSION OF THE DEPARTMENT

- To Inculcate, strong mathematical foundations as applied to AIML domain
- To Equip AIML graduates with skills to meet Industrial and Societal challenges.
- To Foster ethical values & engineering norms and standards in AIML graduates.

Disclaimer

Trademark

Estd : 2001

Edition: 2023- 24

Document Owner

The primary contact for questions regarding this document is:

Author(s):	1. Dr. Mallikarjun H M 2. Prof.Ashwini K 3. Prof. Rashmi B C
Department:	**CSE (AI&ML)**
Contact email ids :	mallikarjun.hm@rnsit.ac.in ashwini.k@rnsit.ac.in rashmi.bc@rnsit.ac.in

COURSE OUTCOMES

Course Outcomes: At the end of this course, students are able to:
CO1- Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO2- Compare various Searching and Inferencing Techniques.
CO3- Develop knowledge base sentences using propositional logic and first order logic.
CO4- Describe the concepts of quantifying uncertainty.
CO5- Use the concepts of Expert Systems to build applications.

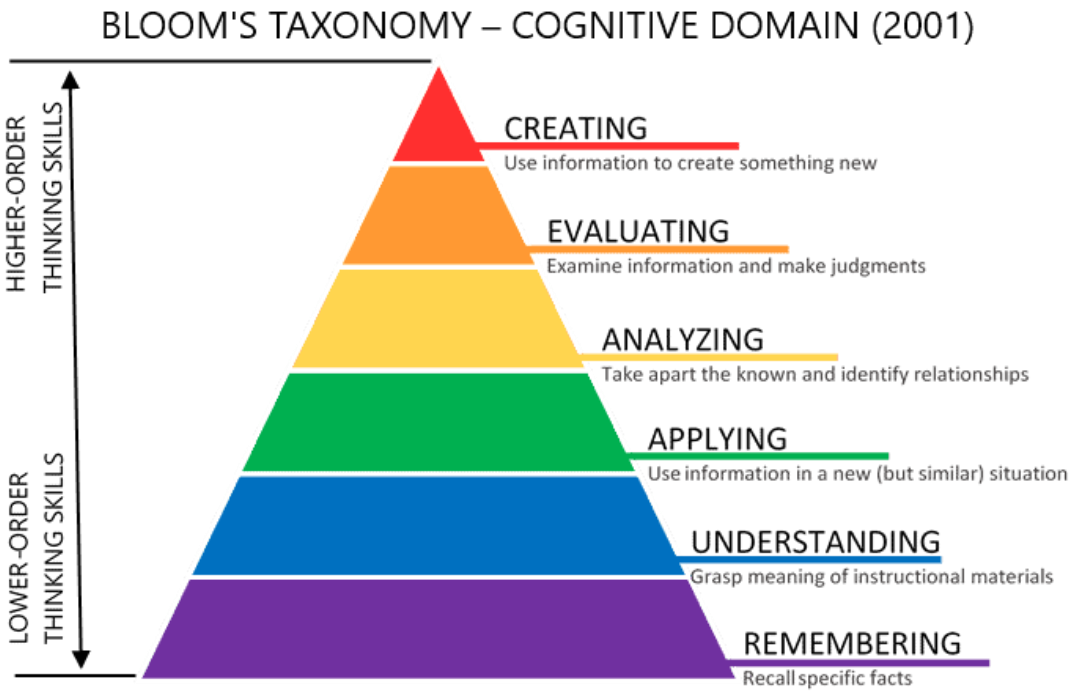
COs and POs Mapping of lab Component

COURSE OUTCOMES	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	2	2	2	2	-	2	2	-	-	-	-	2	-	-	-	-
CO2	2	3	3	3	2	2	2	-	1	-	-	3	-	-	-	-
CO3	2	3	3	3	2	2	2	-	1	-	-	3	-	-	-	-
CO4	2	3	3	3	2	2	2	-	1	-	-	3	-	-	-	-
CO5	2	3	3	3	2	2	2	-	1	-	-	3	-	-	-	-

Mapping of 'Graduate Attributes' (GAs) and 'Program Outcomes' (POs)

Graduate Attributes (GAs) (As per Washington Accord Accreditation)	Program Outcomes (POs) (As per NBA New Delhi)
Engineering Knowledge	Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems
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.
Design/Development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerations for the public health and safety and the cultural, societal and environmental consideration.
Conduct Investigation 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.
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.
The engineer and society	Apply reasoning informed by the contextual knowledge to assess society, health, safety, legal and cultural issues and the consequential responsibilities relevant to the professional engineering practice.
Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental context and demonstrate the knowledge of and need for sustainable development.
Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
Individual and team work	Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.
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.
Project management & finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones won work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
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.

REVISED BLOOMS TAXONOMY (RBT)



PROGRAM LIST

Program 1:

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

Source code:

```
class State:
 def __init__(self, jug1, jug2):
 # Initialize the state of the jugs
 self.jug1 = jug1 # Capacity of jug 1
 self.jug2 = jug2 # Capacity of jug 2
 def __eq__(self, other):
 # Define equality of states
 return self.jug1 == other.jug1 and self.jug2 == other.jug2
 def __hash__(self):
 # Define hash function for states
 return hash((self.jug1, self.jug2))
# Depth-first search algorithm to find a solution
def dfs(start, target, visited):
 # Check if a state is the goal state
 if start == target: # If the current state is the target state, return it
 return [start]
 visited.add(start) # Add the current state to visited states
 for next_state in get_successors(start): # For each possible next state
 if next_state not in visited: # If the next state has not been visited
 path = dfs(next_state, target, visited) # Recursively call DFS on the next state
 if path: # If a solution path is found
 return [start] + path # Return the current state and the solution path
 return None # If no solution path is found, return None
def get_successors(state):
 # Generate successor states from a given state
 successors = [] # List to store successor states
 # Fill jug1
 successors.append(State(jug1=3, jug2=state.jug2))
```

```
 # Fill jug2
 successors.append(State(jug1=state.jug1, jug2=4))
 # Empty jug1
 successors.append(State(jug1=0, jug2=state.jug2))
 # Empty jug2
 successors.append(State(jug1=state.jug1, jug2=0))
 # Pour jug1 to jug2
 pour_amount = min(state.jug1, 4 - state.jug2)
 successors.append(State(jug1=state.jug1 - pour_amount, jug2=state.jug2 +
pour_amount))
 # Pour jug2 to jug1
 pour_amount = min(3 - state.jug1, state.jug2)
 successors.append(State(jug1=state.jug1 + pour_amount, jug2=state.jug2 -
pour_amount))
 return successors
def print_solution(path):
 print("Solution:")
 for i, state in enumerate(path):
 print(f"Step {i}: Jug1={state.jug1}, Jug2={state.jug2}")
def main():
 start_state = State(jug1=0, jug2=0) # Initial state of the jugs
 target_state = State(jug1=2, jug2=0) # Target state we want to achieve
 visited = set() # Set to store visited states
 path = dfs(start_state, target_state, visited) # Find solution using DFS
 if path: # If solution path is found
 print_solution(path) # Print the solution path
 else: # If no solution path is found
 print("No solution found.")
if __name__ == "__main__":
 main()
```

Sample Output:

Solution:

Step 0: Jug1=0, Jug2=0

Step 1: Jug1=3, Jug2=0

Step 2: Jug1=3, Jug2=4

Step 3: Jug1=0, Jug2=4

Step 4: Jug1=3, Jug2=1

Step 5: Jug1=0, Jug2=1

Step 6: Jug1=1, Jug2=0

Step 7: Jug1=1, Jug2=4

Step 8: Jug1=3, Jug2=2

Step 9: Jug1=0, Jug2=2

Step 10: Jug1=2, Jug2=0

Program 2:

AIM: Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems us Python.

Source Code:

```
class State:
 # Define a class to represent the state of the missionaries and cannibals problem
 def __init__(self, missionaries, cannibals, boat):
 # Initialize the state with the number of missionaries, cannibals, and the boat's side
 self.missionaries = missionaries # Number of missionaries on the left bank
 self.cannibals = cannibals # Number of cannibals on the left bank
 self.boat = boat # Indicates the side of the boat (0 for left, 1 for right)
 def __eq__(self, other):
 # Define equality of states
 return self.missionaries == other.missionaries and self.cannibals == other.cannibals and
self.boat == other.boat
 def __hash__(self):
 # Define hash function for states
 return hash((self.missionaries, self.cannibals, self.boat))
def is_valid(state):
 # Check if the current state is valid (no more cannibals than missionaries on either side)
 if state.missionaries < state.cannibals and state.missionaries > 0:
 return False
 if 3 - state.missionaries < 3 - state.cannibals and 3 - state.missionaries > 0:
 return False
 return True
def get_successors(state):
 # Generate successor states from a given state
 successors = []
 if state.boat == 0: # If the boat is on the left side
 for i in range(1, 3): # Try moving 1 or 2 missionaries or cannibals
 for j in range(3):
 if is_valid(State(state.missionaries - i, state.cannibals - j, 1)):
```

```
successors.append(State(state.missionaries - i, state.cannibals - j, 1))
else: # If the boat is on the right side
for i in range(1, 3): # Try moving 1 or 2 missionaries or cannibals
for j in range(3):
if is_valid(State(state.missionaries + i, state.cannibals + j, 0)):
successors.append(State(state.missionaries + i, state.cannibals + j, 0))
return successors
def bfs(start, target):
# Perform Best-First Search to find a solution path
visited = set() # Set to store visited states
frontier = [] # List to store states to be explored
frontier.append([start]) # Add the start state to the frontier
while frontier:
path = frontier.pop(0) # Get the first path from the frontier
current_state = path[-1] # Get the current state from the path
if current_state == target: # If the current state is the target state, return the path
return path
visited.add(current_state) # Add the current state to visited states
for next_state in get_successors(current_state): # For each possible next state
if next_state not in visited: # If the next state has not been visited
new_path = list(path) # Create a new path by copying the current path
new_path.append(next_state) # Add the next state to the new path
frontier.append(new_path) # Add the new path to the frontier
return None # If no solution is found
def print_solution(path):
# Print the solution path
print("Solution:")
for i, state in enumerate(path):
print(f"Step {i}: Left Bank({state.missionaries}, {state.cannibals}), Right Bank({3 -
state.missionaries}, {3 - state.cannibals}), Boat: {'Left' if state.boat == 0 else 'Right'}")
def main():
# Main function to execute the program
start_state = State(3, 3, 0) # Initial state
```

```
target_state = State(0, 0, 1) # Target state
path = bfs(start_state, target_state) # Find solution using BFS
if path: # If solution path is found
print_solution(path) # Print the solution path
else: # If no solution path is found
print("No solution found.")
if __name__ == "__main__":
main()
```

Sample Output:

Solution:
Step 0: Left Bank(3, 3), Right Bank(0, 0), Boat: Left
Step 1: Left Bank(1, 1), Right Bank(2, 2), Boat: Right
Step 2: Left Bank(2, 2), Right Bank(1, 1), Boat: Left
Step 3: Left Bank(0, 0), Right Bank(3, 3), Boat: Right

Program 3:

AIM: Implement A* Search algorithm.

Source Code:

```
import heapq  # Importing the heapq module for heap queue algorithms.

def astar(start, goal, graph, heuristic):
    open_list = [(0, start)]  # Initialize open list with starting node and its f-score.
    closed_list = set()  # Initialize closed list to keep track of visited nodes.
    g_scores = {start: 0}  # Initialize g-scores with starting node having a cost of 0.
    parents = {start: None}  # Initialize parents with starting node having no parent.

    while open_list:  # Continue search until open list is not empty.
        f_score, current = heapq.heappop(open_list)  # Get node with lowest f-score from open list.

        if current == goal:  # Check if goal node is reached.
            path = []  # Initialize path to store the optimal path.
            while current:  # Reconstruct path by backtracking through parents.
                path.append(current)
                current = parents[current]
            return path[::-1], f_score  # Return the reversed path and its final f-score.

        closed_list.add(current)  # Add current node to closed list as it has been evaluated.

        for neighbor in graph[current]:  # Explore neighbors of the current node.
            if neighbor in closed_list:  # Ignore neighbors already evaluated.
                continue
            tentative_g_score = g_scores[current] + graph[current][neighbor]  # Calculate tentative g-score.
            if neighbor not in [n[1] for n in open_list]:  # Add neighbor to open list if not already in it.
                heapq.heappush(open_list, (tentative_g_score + heuristic(neighbor, goal), neighbor))
            elif tentative_g_score < g_scores[neighbor]:  # Update neighbor's g-score if new path is better.
                index = [n[1] for n in open_list].index(neighbor)  # Find index of neighbor in open list.
```

```
                open_list[index] = (tentative_g_score + heuristic(neighbor, goal), neighbor)  # Update neighbor' score in open list.

            parents[neighbor] = current  # Update parent of neighbor node.
            g_scores[neighbor] = tentative_g_score  # Update g-score of neighbor node.
    return None  # Return None if no path found.
# Example graph.
graph = {
    'A': {'B': 5, 'C': 10},
    'B': {'D': 15, 'E': 20},
    'C': {'F': 5},
    'D': {'G': 25},
    'E': {'G': 20},
    'F': {'G': 10},
    'G': {}
}
# Heuristic function.
def heuristic(node, goal):
    h_scores = {
        'A': 35,
        'B': 30,
        'C': 25,
        'D': 20,
        'E': 15,
        'F': 10,
        'G': 0
    }
    return h_scores[node]
# Find path from A to G.
path, dist = astar('A', 'G', graph, heuristic)
print("Path:", path, "Distance:", dist)
```

Sample Output:

Path: ['A', 'C', 'F', 'G'] Distance: 25

Program 4:

AIM: Implement AO* Search algorithm.

Source Code:

```
import heapq
def ao_star(start, goal, graph, heuristic, epsilon):
    open_list = [(0, start)]  # Initialize open list with starting node and its f-score.
    closed_list = set()  # Initialize closed list to keep track of visited nodes.
    g_scores = {start: 0}  # Initialize g-scores with starting node having a cost of 0.
    parents = {start: None}  # Initialize parents with starting node having no parent.
    while open_list:  # Continue search until open list is not empty.
        f_score, current = heapq.heappop(open_list)  # Get node with lowest f-score from open list
        if current == goal:  # Check if goal node is reached.
            path = []  # Initialize path to store the optimal path.
            while current: # Reconstruct path by backtracking through parents.
                path.append(current)
                current = parents[current]
            return path[::-1], f_score  # Return the reversed path and its final f-score.
        closed_list.add(current)  # Add current node to closed list as it has been evaluated.
        for neighbor in graph[current]:  # Explore neighbors of the current node.
            if neighbor in closed_list:  # Ignore neighbors already evaluated.
                continue
            tentative_g_score = g_scores[current] + graph[current][neighbor]  # Calculate tentative g-score.
            if neighbor not in [n[1] for n in open_list]:  # Add neighbor to open list if not already in it.
                heapq.heappush(open_list, (tentative_g_score + heuristic(neighbor, goal), neighbor))
            elif tentative_g_score < g_scores[neighbor]:  # Update neighbor's g-score if new path is better.
                index = [n[1] for n in open_list].index(neighbor)  # Find index of neighbor in open list.
                open_list[index] = (tentative_g_score + heuristic(neighbor, goal), neighbor)  # Update neighbor' score in open list.
            parents[neighbor] = current  # Update parent of neighbor node.
            g_scores[neighbor] = tentative_g_score  # Update g-score of neighbor node.
    return None  # Return None if no path found.
```

```
# Example graph.
graph = {
    'A': {'B': 5, 'C': 10},
    'B': {'D': 15, 'E': 20},
    'C': {'F': 5},
    'D': {'G': 25},
    'E': {'G': 20},
    'F': {'G': 10},
    'G': {}
}

# Heuristic function.
def heuristic(node, goal):
    h_scores = {
        'A': 35,
        'B': 30,
        'C': 25,
        'D': 20,
        'E': 15,
        'F': 10,
        'G': 0
    }
    return h_scores[node]
# Find path from A to G with AO*.
path, dist = ao_star('A', 'G', graph, heuristic, epsilon=1)
print("Path:", path, "Distance:", dist)
```

Sample Output:

Path: ['A', 'C', 'F', 'G'] Distance: 25

Program 5:

AIM: Solve 8-Queens Problem with suitable assumptions.

Source Code:

```
def is_safe(board, row, col):
 # Check if there is a queen in the same column
 for i in range(row):
 if board[i] == col:
 return False

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

 # Check upper right diagonal
 for i, j in zip(range(row, -1, -1), range(col, 8)):
 if board[i] == j:
 return False

 return True
def solve_queens(board, row):
 # Base case: If all queens are placed, return True
 if row >= 8:
 return True

 # Check placing the queen in each column of the current row
 for col in range(8):
 if is_safe(board, row, col):
 # Place the queen
 board[row] = col

 # Recur to place the next queen
```

```
 if solve_queens(board, row + 1):
 return True

 # Backtrack if placing the queen doesn't lead to a solution
 board[row] = -1

 # Return False if no safe placement is found in this row
 return False
def print_board(board):
 # Print the chessboard with queens placed
 for i in range(8):
 for j in range(8):
 if board[i] == j:
 print("Q", end=" ")# Print queen
 else:
 print(".", end=" ")# Print empty cell
 print()
def main():
 # Initialize the board with all cells as empty (-1)
 board = [-1] * 8

 # Solve the 8 Queens problem starting from the first row
 if solve_queens(board, 0):
 print("Solution:")
 print_board(board)
 else:
 print("No solution exists.")# If no solution is found
if __name__ == "__main__":
 main()
```

Output:

Solution:

Q . . . . . .

. . . . Q . . .

. . . . . Q

. . . . . Q . .

. . Q . . . . .

. . . . . Q .

. Q . . . . . .

. . . Q . . . .

Program 6:

AIM: Implementation of TSP using heuristic approach.

Source Code:

```
import sys # Importing the sys module to access system-specific parameters and functions
def tsp(graph, start=0):
 # Traveling Salesman Problem solver using the Nearest Neighbor heuristic
 num_nodes = len(graph) # Number of nodes in the graph
 visited = [False] * num_nodes # Initialize all nodes as unvisited
 path = [start] # Start the path from the given start node
 visited[start] = True # Mark the start node as visited
 # Continue until all nodes are visited
 while len(path) < num_nodes:
 current_node = path[-1] # Get the current node
 nearest_neighbor = None
 min_distance = sys.maxsize # Initialize minimum distance to a very large value
 # Find the nearest unvisited neighbor
 for neighbor in range(num_nodes):
 if not visited[neighbor] and graph[current_node][neighbor] < min_distance:
 nearest_neighbor = neighbor
 min_distance = graph[current_node][neighbor]
 # Mark the nearest neighbor as visited and add it to the path
 visited[nearest_neighbor] = True
 path.append(nearest_neighbor)
 # Complete the cycle by returning to the start node
 path.append(start)
 return path
# Example graph representing distances between cities (0-based index)
graph = [
 [0, 10, 15, 20],
 [10, 0, 35, 25],
 [15, 35, 0, 30],
 [20, 25, 30, 0]
]
```

```
# Find the optimal path using the nearest neighbor heuristic
optimal_path = tsp(graph)
# Print the optimal path
print("Optimal Path:", optimal_path)
# Calculate the total distance of the optimal path
# by summing up the distances between consecutive nodes
total_distance = sum(graph[optimal_path[i - 1]][optimal_path[i]] for i in range(1,
len(optimal_path)))
print("Total Distance:", total_distance)
```

Sample Output:

```
Optimal Path: [0, 1, 3, 2, 0]
Total Distance: 80
```

Program 7:

AIM: Implementation of the problem-solving strategies: either using Forward Chaining or Backward Chainii

Source Code:

```
def backward_chaining(KB, query):
 # Function to perform backward chaining inference
 agenda = [query] # Initialize the agenda with the query
 inferred = set() # Initialize set of inferred statements
 while agenda:
 q = agenda.pop() # Get the next query from the agenda
 if q in inferred:
 continue # Skip if the query has already been inferred
 if q not in KB:
 print(f"Query '{q}' is not in knowledge base.")
 return False # If query is not in the knowledge base, return False
 # Add the query's prerequisites to the agenda
 for p in KB[q]:
 agenda.append(p)
 # Add the query to the inferred set
 inferred.add(q)
 # If the query is inferred, return True
 if query in inferred:
 return True
 else:
 return False
# Example knowledge base represented as a dictionary
# Each statement has a list of prerequisites
KB = {
 'p': ['q', 'r'],
 'q': ['s'],
 'r': [],
 's': []
}
```

```
# Example query
query = 'p'
# Perform backward chaining to determine if the query can be inferred from the knowledge
base
result = backward_chaining(KB, query)
# Print the result
if result:
 print(f"Query '{query}' can be inferred from the knowledge base.")
else:
 print(f"Query '{query}' cannot be inferred from the knowledge base.")
```

Sample Output:

Query 'p' can be inferred from the knowledge base.

Program 8:

AIM: Implement resolution principle on FOPL related problems.

Source Code:

```
def negation(sentence):
 """
 Return the negation of a sentence by adding a negation symbol '~' to it.
 """
 return "~" + sentence
def resolve(c1, c2):
 """
 Resolve two clauses by eliminating complementary literals.
 """
 resolved = []
 for literal in c1:
 if negation(literal) in c2:
 # Eliminate complementary literals
 resolved.append(c1.replace(literal, "").replace(negation(literal), ""))
 return resolved
def resolution(KB, query):
 """
 Perform resolution inference to determine if the query can be inferred from the
knowledge base.
 """
 KB.append(negation(query)) # Add the negation of the query to the knowledge base
 new = set(KB) # Initialize set of new clauses with the knowledge base
 while True:
 old = set(new) # Update old clauses with the current set of new clauses
 for ci in old:
 for cj in old:
 if ci != cj:
 # Resolve every pair of clauses
 resolvents = resolve(ci, cj)
```

```
 if "" in resolvents:
 # If an empty clause is obtained, the query can be inferred
 return True
 new.update(resolvents) # Update set of new clauses with resolvents
 if new == old:
 # If no new clauses are added, stop
 return False
# Example knowledge base represented as a list of clauses
KB = [
 "P(x) | Q(x)",
 "~P(A)",
 "Q(A)"
]
# Example query
query = "Q(A)"
# Perform resolution inference to determine if the query can be inferred from the knowledge
base
result = resolution(KB, query)
# Print the result
if result:
 print(f"The query '{query}' can be inferred from the knowledge base.")
else:
 print(f"The query '{query}' cannot be inferred from the knowledge base.")
```

Sample Output:

The query 'Q(A)' cannot be inferred from the knowledge base.

Program 9:

AIM: Implement Tic-Tac-Toe game using Python.

Source Code:

```
def print_board(board):
 """
 Print the Tic Tac Toe board.
 """
 for row in board:
 print(" | ".join(row))# Join elements of each row with "|"
 print("-" * 9)# Print horizontal line after each row
def check_winner(board):
 """
 Check if there is a winner or if the game is a draw.
 """
 # Check rows and columns
 for i in range(3):
 if board[i][0] == board[i][1] == board[i][2] != " ":# Check row-wise
 return board[i][0]
 if board[0][i] == board[1][i] == board[2][i] != " ":# Check column-wise
 return board[0][i]
 # Check diagonals
 if board[0][0] == board[1][1] == board[2][2] != " ":# Check diagonal from top-left to
bottom-right
 return board[0][0]
 if board[0][2] == board[1][1] == board[2][0] != " ":# Check diagonal from top-right to
bottom-left
 return board[0][2]# Return the winning player
 # Check for draw
 if all(board[i][j] != " " for i in range(3) for j in range(3)):# If all cells are filled
 return "draw"# Return "draw" indicating a draw
 return None# Return None if there is no winner or draw
def tic_tac_toe():
```

```
"""
Main function to run the Tic Tac Toe game.
"""
board = [[" " for _ in range(3)] for _ in range(3)] # Initialize the board with empty spaces
current_player = "X" # X starts the game
while True:
print_board(board) # Print the board
# Get the move from the current player
row = int(input(f"Player {current_player}, enter row (0-2): "))# Get row input
col = int(input(f"Player {current_player}, enter column (0-2): "))# Get column input
# Check if the move is valid
if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":# If row and column are
valid and cell is empty
board[row][col] = current_player # Make the move
winner = check_winner(board) # Check for winner or draw
if winner:# If there's a winner or draw
print_board(board)# Print the final board
if winner == "draw":# If it's a draw
print("It's a draw!")# Print draw message
else:
print(f"Player {winner} wins!")# Print winning player message
break
current_player = "O" if current_player == "X" else "X" # Switch player
else:
print("Invalid move! Try again.")# Print message for invalid move
# Run the Tic Tac Toe game
tic_tac_toe()
```

Sample Output:

```
| |
---------
 | |
---------
 | |
---------
Player X, enter row (0-2): 1
Player X, enter column (0-2): 2
 | |
---------
 | | X
---------
 | |
---------
Player O, enter row (0-2): 1
Player O, enter column (0-2): 1
 | |
---------
 | O | X
---------
 | |
---------
Player X, enter row (0-2): 2
Player X, enter column (0-2): 2
 | |
---------
 | O | X
---------
 | | X
---------
Player O, enter row (0-2): 1
Player O, enter column (0-2): 1
```

Invalid move! Try again.

| |

| O | X

| | X

Player O, enter row (0-2): 2

Player O, enter column (0-2): 2

Invalid move! Try again.

| |

| O | X

| | X

Player O, enter row (0-2): 1

Player O, enter column (0-2): 2

Invalid move! Try again.

| |

| O | X

| | X

Player O, enter row (0-2): 1

Player O, enter column (0-2): 0

| |

O | O | X

| | X

Player X, enter row (0-2): 0

Player X, enter column (0-2): 2

 | | X

O | O | X

 | | X

Player X wins!

Program 10:

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

Source Code:

```
import wikipedia# Import the Wikipedia library
def search_and_provide_info(query):
 try:
 # Use the Wikipedia library to search for the query
 results = wikipedia.search(query)

 if not results:
 return "No relevant information found."# Return if no results are found

 # Select the first result and fetch its summary
 page_summary = wikipedia.summary(results[0], sentences=2)
 return page_summary
 except wikipedia.exceptions.DisambiguationError as e:
 # If there are multiple results, provide suggestions
 return f"Multiple results found. Try to be more specific: {', '.join(e.options[:5])}"
 except wikipedia.exceptions.PageError:
 return "No information found for the given query."# Return if no page exists for the
query
 except Exception as e:
 return f"An error occurred: {e}"# Return if any other exception occurs
def main():
 while True:
 # Take user input for the search query
 query = input("Enter your search query (or 'quit' to exit): ")

 if query.lower() == 'quit':
 print("Exiting...")# Exit message
 break# Break out of the loop

 # Call the function to search and provide information
```

```
 result = search_and_provide_info(query)

 # Print the result
 print(result)
if __name__ == "__main__":
 main()
```

Sample Output:

Enter your search query (or 'quit' to exit): Artificial Intelligence
Artificial intelligence (AI), in its broadest sense, is intelligence exhibited by machines, particularly computer systems. It is a field of research in computer science that develops and studies methods and software which enable machines to perceive their environment and uses learning and intelligence to take actions that maximize their chances of achieving defined goals.
Enter your search query (or 'quit' to exit): rashmi
Rashmi (also Rashami, Rashmika) (Sanskrit: रश्मि) is a Hindu, Sanskrit unisexual name in India. Rashmi means a single 'ray of light', a very popular name in Hindu.
Enter your search query (or 'quit' to exit): r n s institute of technology
Ramaiah Institute of Technology (RIT), formerly known as M.S. Ramaiah Institute of Technology (MSRIT), is an autonomous private engineering institute located in Bengaluru in the Indian state of Karnataka. Established in 1962, the college is affiliated to Visvesvaraya Technological University.

Program 11:

AIM: Implement any Game and demonstrate the Game playing strategies.

Source Code:

```
import random # Import the random module
def rock_paper_scissors(): # Define the function for the game
 choices = ['rock', 'paper', 'scissors'] # Define the choices that can be made
 computer_choice = random.choice(choices) # The computer makes a random choice
 # The user is asked to make a choice
 user_choice = input("Enter your choice (rock, paper, scissors): ")
 # If the user's choice is not valid, they are asked again
 while user_choice not in choices:
 user_choice = input("Invalid choice. Enter your choice (rock, paper, scissors): ")
 # The choices of both the user and the computer are printed
 print(f"\nUser choice: {user_choice}")
 print(f"Computer choice: {computer_choice}")
 # Determine the winner
 if user_choice == computer_choice:
 return "\nIt's a tie!"
 if (user_choice == 'rock' and computer_choice == 'scissors') or \
 (user_choice == 'scissors' and computer_choice == 'paper') or \
 (user_choice == 'paper' and computer_choice == 'rock'):
 return "\nUser wins!"
 else:
 return "\nComputer wins!"
print(rock_paper_scissors()) # Call the function to start the game
```

Sample Output:

```
Enter your choice (rock, paper, scissors): rock
User choice: rock
Computer choice: rock
It's a tie!
```

Viva Questions

1. What is Depth First Search (DFS)?

DFS is a graph traversal algorithm that starts at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking.

2. How is DFS applied in the Water Jug Problem?

In the Water Jug Problem, DFS is used to explore all possible states by filling, emptying, or transferring water between jugs until the goal state is reached. The algorithm explores deeper into possible actions before backtracking.

3. What are the limitations of DFS in solving problems like the Water Jug Problem?

DFS can be inefficient as it may explore unpromising paths deeply, leading to high computational costs and memory usage. It may also get stuck in infinite loops without proper checks for repeated states.

4. What is the Best First Search algorithm?

Best First Search is an informed search algorithm that selects the most promising node based on a heuristic function to guide the search towards the goal state.

5. How does the Best First Search work in the Missionaries and Cannibals problem?

In the Missionaries and Cannibals problem, the Best First Search algorithm selects states based on a heuristic such as the number of missionaries and cannibals on the correct side of the river. It explores states that minimize this heuristic value, steering the search towards the solution.

6. What are the drawbacks of Best First Search?

Best First Search can suffer from getting stuck in local optima if the heuristic function doesn't perfectly align with the goal. It also doesn’t guarantee the optimal solution unless the heuristic is admissible.

7. What is the A* Search algorithm?

A* Search is a pathfinding and graph traversal algorithm that finds the shortest path from a start node to a goal node. It uses a heuristic to estimate the cost of reaching the goal and combines it with the actual cost from the start node.

8. How does A* Search work?

A* maintains a priority queue of nodes to explore, where each node has a cost function f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n). g(n)g(n)g(n) is the cost to reach the node, and h(n)h(n)h(n) is the estimated cost to the goal. The algorithm selects the node with the lowest f(n)f(n) f(n) and continues the search until it reaches the goal.

9. What makes A* better than other search algorithms?

A* is optimal and complete if the heuristic function is admissible (never overestimates the cost). It efficiently balances between exploration and exploitation by using both actual cost and heuristic cost.

10. What is the AO* Search algorithm?

AO* is a search algorithm that is used for solving problems with AND/OR graphs, where the nodes represent decisions and the edges represent possible actions. It is used for solving problems that involve making decisions under uncertainty.

11. How is AO* different from A* Search?

Unlike A*, which works on regular graphs, AO* works on AND/OR graphs. AND nodes represent a set of actions that must all be done, while OR nodes represent actions where one of many options may be chosen.

12. In what type of problems is AO* particularly useful?

AO* is useful in problems that have a hierarchical structure, such as planning problems, decision-making problems, and problems that involve multiple objectives or sub-goals.

13. What is the 8-Queens Problem?

The 8-Queens Problem involves placing 8 queens on an 8x8 chessboard such that no two queens can attack each other, meaning no two queens can be on the same row, column, or diagonal.

14. How is the problem solved?

The problem is typically solved using backtracking algorithms. The solution involves placing queens one by one in each row, ensuring that no two queens threaten each other. If a conflict arises, the algorithm backtracks and tries a different configuration.

15. What is the time complexity of the backtracking solution for the 8-Queens Problem?

The time complexity is O(N!)O(N!)O(N!), where NNN is the number of queens, since the algorithm tries placing each queen in each row and backtracks when a conflict is found.

16. What is the Traveling Salesman Problem (TSP)?

TSP is a classic optimization problem where a salesman must visit a set of cities exactly once and return to the origin city, minimizing the total distance traveled.

17. How can heuristics help in solving TSP?

Heuristics like the Nearest Neighbor or Greedy Approach can provide an approximate solution to the TSP. These heuristics help to quickly find a solution without guaranteeing the optimal path, but they are computationally efficient.

18. What is the time complexity of the Nearest Neighbor heuristic for TSP?

The time complexity of the Nearest Neighbor heuristic is O(N2)O(N^2)O(N2), where NNN is the number of cities, as it involves visiting each city once and choosing the nearest city at each step.

19. What is Forward Chaining?

Forward Chaining is a data-driven inference technique where the system starts from known facts and applies rules to infer new facts until a goal is reached.

20. What is Backward Chaining?

Backward Chaining is a goal-driven inference technique where the system starts from a goal and works backward through the rules to see if it can reach known facts that support the goal.

21. When would you prefer Forward Chaining over Backward Chaining?

Forward Chaining is preferred when you have a large set of facts and you need to generate as many inferences as possible, while Backward Chaining is more efficient when you're trying to prove specific goals.

22. What is the Resolution Principle?

The Resolution Principle is a rule of inference used in automated reasoning. It involves deriving conclusions from logical sentences by resolving contradictions between two clauses. This is often used in the context of First-Order Predicate Logic (FOPL).

23. How is the Resolution Principle used in FOPL?

In FOPL, the Resolution Principle is applied to convert logical sentences into a normal form (like conjunctive normal form), and then it systematically resolves pairs of clauses to deduce new information or contradictions.

24. How does the Tic-Tac-Toe game work?

The game is played on a 3x3 grid, where two players (X and O) take turns to place their marks in the cells. The goal is to get three of the same mark in a row, column, or diagonal.

25. How do you implement the AI for the Tic-Tac-Toe game?

The AI can be implemented using strategies like the Minimax algorithm, which evaluates possible moves and chooses the optimal one based on maximizing the player's chances of winning while minimizing the opponent's chances.

26. What is the purpose of the bot in this task?

The bot serves to retrieve and display relevant information based on user input in the search box. It could use techniques such as natural language processing (NLP) or keyword matching to understand and respond.

27. What technologies are used to build such a bot?

The bot can be built using Python libraries like NLTK, spaCy, or transformers for NLP tasks, and frameworks like Flask or Django for web-based applications.

28. What are game-playing strategies?

Game-playing strategies involve algorithms or approaches to make decisions during a game. For example, in Tic-Tac-Toe, the Minimax algorithm is used to evaluate moves and make optimal decisions.

29. How can you ensure that the strategy leads to an optimal solution?

By using algorithms like Minimax (for two-player, zero-sum games), which evaluates all possible game states and selects the best move for the player, or by implementing Monte Carlo Tree Search for more complex games.

Additional Programs

1. BFS Program

```
from collections import deque
def bfs(graph, start, goal):
    queue = deque([(start, [start])])  # Store path along with node
    visited = {start}
    while queue: #This loop will run as long as there are nodes in the queue.
        node, path = queue.popleft()
        if node == goal:
            return path
        for neighbor in graph.get(node, []):
            # retrieves the neighbors of the node if it exists in the graph. If it doesn't exist (i.e., no neighbors), it returns an empty list.
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
    return None  # No path found
# Define the graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'G'],
    'F': ['C'],
    'G': ['E']
}
start_node = 'A'
goal_node = 'G'
path = bfs(graph, start_node, goal_node)
if path:
    print(f"Path from {start_node} to {goal_node}: {path}")
else:
```

```
    print(f"No path found.")
start_node = 'A'
goal_node = 'D'
path = bfs(graph, start_node, goal_node)
if path:
    print(f"Path from {start_node} to {goal_node}: {path}")
else:
    print(f"No path found.")
```

2. Iterative DFS Program

```
def dfs_iterative(graph, start, goal):
    stack = [(start, [start])]  # Stack of (node, path) tuples
    visited = set()
    while stack:
        node, path = stack.pop()
        if node not in visited:
            visited.add(node)
            if node == goal:
                return path
            for neighbor in reversed(graph.get(node, [])): # Reverse for typical DFS order
                if neighbor not in visited:
                    stack.append((neighbor, path + [neighbor]))
    return None
start_node = 'A'
goal_node = 'G'
path = dfs_iterative(graph, start_node, goal_node)
if path:
    print(f"DFS (Iterative) Path from {start_node} to {goal_node}: {path}")
else:
    print(f"No path found.")
```