G. Rakshitha IBM21CS258 5B

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	10/11/23	Basic python programs	1 - 3	
2	17/11/23	Tic-tac-toe game	4 - 9	
3	24/11/23	To implement 8X8 puzzle	10 - 11	
4	8/12/23	8X8 Iterative deepening A* search	12 - 15 16 - 19	
5	22/12/23	Vaccum cleaner problem.	20 - 23	
6	29/12/23	Knowledge base entailment knowledge base resolution.	24 - 29	
7	19/01/24	Unification in first order logic	30	
		Convert FOL to CNF	- 39	
		Forward reasoning		

Using else if ladder

1)
```
age = int(input("Enter your age : "))

if age < 0:
    print("Invalid age")
elif age == 18:
    print("You are minor")
elif age == 45:
    print("You are an adult")
else:
    print("You are an senior citizen")
```

Output

```
Enter your age : 45
You are an adult
```

2) Read the integers in the reverse order

```
num = int(input("Enter an integer to reverse"))
reverse = 0

while num > 0:
    digit = num % 10
    reverse = reverse * 10 + digit
    num = num // 10

print("Reverse integer : ", reverse)
```

Output

```
Enter an integer to reverse 12
Reversed integer 21
```

3] To print sequence

```
n = int(input("Enter the number."))

for i in range(1, n+1):
    for j in range(i):
        print(i)
```

Output:

```
Enter the number : 3
1
2
2
3
3
3
```

4]

```
n = int(input("Enter the number:"))

for i in range(1, n+1):
    for j in range(1, i+1):
        print(j)
```

Output

```
Enter the number : 3
1
1
2
1
2
3
```

5] Power of a number

```
base = int(input("Enter the base: "))
exponent = int(input("Enter the exponent: "))

result = base ** exponent

print(result) or print(f"the result is {result}")
```

output

```
Enter the base: 2
Enter the exponent: 2
the result is 4
```

6] switch case

```
while true:
    print("Menu:")
    print("1. opt 1")
    print("2. opt 2")
    print("3: opt 3")
    print("4.

lang = int(input("Lang"))
match lang:
    case "Java":
        print("1")
    case "PHP":
        print("2")
    case "Javascript":
        print("3")
    case _:
        print("default")
```

Output

```
Enter your choice: 1
1 opt 1
```

10/11/23

Lab program - 1

1 Implement Tic-Tac-Toe game

```
board = [' '
' for x in
range(10)]

    def insertLetter(letter, pos):
        board[pos] = letter

    def spaceIsFree(pos):
        return board[pos] == ' '

    def printBoard(board):
        print('   |   |')
        print(' ' + board[1] + ' | ' +
              board[2] + ' | ' + board[3])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[4] + ' | ' +
              board[5] + ' | ' + board[6])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[7] + ' | ' +
              board[8] + ' | ' + board[9])
        print('   |   |')

    def isWinner(bo, le):
        return (bo[7] == le and bo[8]
        == le and bo[9] == le)
        or (bo[4] == le and bo[5] == le
    and bo[6] == le) or
        (bo[1] == le and bo[2] == le and
```

```
bo[3]==le) or (bo[1]==le and
bo[4]==le and bo[7]==le) or
(
bo[2]==le and bo[5]==le and bo[8]
==le) or (
bo[3]==le and bo[6]==le and
bo[9]==le) or (
bo[1]==le and bo[5]==le and
bo[9]==le) or (
bo[3]==le and bo[5]==le and
bo[7]==le)

def playerMove():

run = True

while run:
    move = input('Please select a position
        to place an \'X\' (1-9):')
    try:
        move = int(move)
        if move>0 and move<10:
            if spaceIsFree(move):
                run = False
                insertLetter('X', move)
            else:
                print('Sorry this space is
                    occupied!')
        else:
            print('Please type a number
                within the range!')
    except:
        print('Please type a
            number!')
```

```
def compMove():
    possibleMoves = []
    enumerate(board, if letter ==
    and i != 0]
        move 0

    for let in ['O', 'X']:
        for i in possibleMoves:
            boardCopy = board[:]
            boardCopy[i] = let
            if isWinner(boardCopy, let):
                move = i
                return move

    cornersOpen = []
    for i in possibleMoves:
        if i in [1,3,7,9]:
            cornersOpen.append(i)

    if len(cornersOpen) > 0:
        move = selectRandom(cornersOpen)
        return move.

    if 5 in possibleMoves:
        move = 5
        return move

    edgesOpen = []
    for i in possibleMoves:
        if i in [2,4,6,8]:
            edgesOpen.append(i).
```

```
if len(edgesOpen) > 0:
    move = SelectRandom(edgesOpen)
return move.

def selectRandom(li):
    import random
    ln = len(li)
    r = random.randrange(0, ln)
    return li[r]

def isBoardFull(board):
    if board.count(' ') > 1:
        return False
    else:
        return True.
```

Alogrithm

1: Initialize the board.

```
board = [' ' for _ in range(10)]
```

2: Insert a letter ('X' or 'O') into the Board.

```
def insertLetter(letter, pos):
```

3: check if space on the Board is free:

```
def spaceIsFree(pos):
    return board[pos] == ' '
```

4: print the Board

def printBoard(board):

5: check for a winner.

def isWinner(bo, le):

6: player's move

def playerMove():
Ask the player to choose a position to place their 'X' on the board

7: Computer's move.

Implement the logic for the computers move
def compMove

Output

Player 'X', choose your position (1-9) : 5

```
  |   |
  | X |
  |   |
```

Computers turn

```
  |   |
O | X |
  |   |
```

player X, choose your position (1-9): 1

```
X |   |
O | X |
  |   |
```

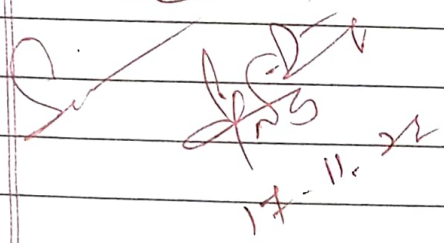
Computers turn

```
X | O |
O | X |
  |   | 1
```

Player 'X', choose your position (1-9) : 9

```
X | O |
O | X |
  |   | X
```

Congratulations ! You win !



Lab program - 2

To implement 8 X 8 puzzle

Algorithm

1	3	2
4	6	7
5	8	0

1	2	
4	5	
7	8	

Step 1: Start.

Step 2: Initialize puzzle.
Create an instance of puzzle.
The initial board is created randomly using two dimensional array

Step 3: Using loop we have to solve the puzzle.

Step 4: Display the current state of the board using print board function

Call the move method or function to move the number which is specified by the user Step swap the blank tile with

Step 5: Check for the valid move.
Ensure the move direction is valid based on the current position.

Step 6: check for puzzle solved.
is solved method is compare the current board state with the goal

Step 4: Once the loop exist the puzzle is solved or else the puzzle cannot be solved.

Step 5: End

Output

Input

1	2	3
4	5	6
7	8	0

1	2	3
4	5	0
6	7	8

The moves are ['down', 'left', 'left', 'up', 'right', 'down', 'right', 'up', 'left', 'left', 'down', 'right', 'right']

Import heapq

```
class puzzleNode:
    def __init__(self, state, parent = None,
                 move = None, cost = 0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost
        self.heuristic = self.calcuate_heuristic()

    def __lt__(self, other):
        return (self.cost + self.heuristic) <
        (other.cost + other.heuristic)
```

Lab program

3) 8-Puzzle Iterative deepening search algorithm.

Algorithm

Step 1: Define puzzle node class. Each node represents a state of the puzzle

Step 2: Define helper functions

get_blank_position(state)
get_neighbors(node)
apply_action(state, action)
print_solution(solution)

Step 3: depth limited search function

depth_limited_search(node, goal_state, depth_limit)

Step 4: Iterative deepening search function

iterative_deepening_search(initial_goal_state)

Step 5: Main function.

code.

```
import copy

GOAL_STATE = [[1,2,3], [8,0,4], [7,6,5]]
MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT =
        'U', 'D', 'L', 'R'
MOVES = [MOVE_UP, MOVE_DOWN, MOVE_LEFT,
         MOVE_RIGHT]

def print_puzzle(state):
    for row in state:
        print(row)
    print()

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

def move(state, direction):
    i, j = find_blank(state)
    new_state = copy.deepcopy(state)

    if direction == MOVE_UP and i>0:
        new_state[i][j], new_state[i-1][j] =
            new_state[i-1][j], new_state[i][j]
    elsif direction == MOVE_DOWN and i<2:
        new_state[i][j], new_state[i+1][j] =
            new_state[i+1][j], new_state[i][j]

    return new_state
```

```
def is_goal(state):
    return state == GOAL_STATE

def depth_limited_search(state, depth, path):
    if depth == 0:
        return None

    if is_goal(state):
        return path

    for move_direction in MOVES:
        new_state = move(state, move_direction)
        if new_state not in path:
            new_path = path + [new_state]
            result = depth_limited_search
                (new_state, depth-1,
                    new_path)

    return None

def iterative_deepening_search(initial
        state):
    depth = 0
    while True:
        result = depth_limited_search(
                initial_state, depth, [initial_state])
        if result is not None:
            return result
        depth += 1

initial_state = [[1,2,3],[0,8,4],[7,6,5]]
```

```
if solution_path is not None:
    print("Solution found:")
else:
    print("No solution found")
```

Output

```
Initial State:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Solution found:

step 1:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

step 2:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

Lab program - 4

4) 8 Puzzle A* search algorithm

Algorithm

Step 1: Define Puzzle Node class

- Each node represents a state of the puzzle
- The state is a 3X3 grid

Step 2: Define Puzzlenode methods

calculate_heuristic(): calculate the heuristic value for the current state

__lt__(self, other)

Step 3: Define helper functions

get_blank_position(state)
Find the position of the blank(0)

get_neighbours(node)
Get valid neighbours for a given tile.

apply_action(state, action)
Apply the given action.

Step 4: A search algorithm *

Initialize the initial state as a puzzle node.

Initialize a set to store explored states

Step 5: Print solution

Trace back from goal state to initial state

Step 6: Main function
Define the initial state of puzzle
Call the A* algorithm with the initial state

Code

```
class
Node:
    def __init__(self, data, level, fval):
        """ Initialize the node with the data,
        level of the node with the data,
        level of the node and the calculated
        f value """ """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the give node
        by moving the blank space either
        in the four directions { up, down,
        left, right } """
        x, y = self.find(self.data, x, y,
                         i[0], i[1])

        if child is not None:
            child_node = Node(child, self.level+1)
```

```
children.append(child_node)
return children

def shuffle(self, puz, x1, y1, x2, y2):
    """ Move the blank space in the given
        direction """
    if x2 >= 0 and x2 < len(self.data)
    and y2 >= 0 and y2 < len(self.data):

        temp_puz = []
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][y1]
        temp_puz[x1][y1] = temp
        return temp_puz
    else
        return None

def copy(self, root)
    """ copy function to create a similar
        matrix of the given node """
    temp = []
    for i in root:
        t = []
        for j in i :
            t.append(j)
        temp.append(t)
    return temp
```

```
def process(self):
    "Accept Start and goal puzzle state"
    print("Enter the start state matrix")
    start = self.f(start, goal)
    while True:
        cur = self.open[0]
        print("  ")
        print(" | ")
        print(" | ")
        for i in cur.data:
            for j in i:
                print(j, end=" ")
            print("")

puz = Puzzle(3)
puz.process()
```

output

Enter the start state matrix

1 2 3
0 4 6
7 5 8

Enter the goal state matrix

1 2 3
4 5 6
7 8 0

↓

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

Lab program - 5

Vaccum cleaner problem

Algorithm

function REFLEX-VACUM-AGENT([location, status]) returns action

if status = Dirt then returns Suck
else if location = A then return Right
else if location = B then return left

Code

```
def clean(floor):
    i, j, row, col = 0, 0; len(floor),
        len(floor[0])
    for i in range(row):
        if i % 2 == 0:
            for j in range(col):
                if floor[i][j] == 1:
                print_floor(floor, i, j, "clean")
                floor[i][j] = 0
                print_floor(floor, i, j, "Right"
                    if j < col - 1 else "stay")
            else:
                for j in range(col - 1, -1, -1):
                    if floor[i][j] == 1:
                    print_floor(floor, i, j, "clean")
                    floor[i][j] = 0
                    print_floor(floor, i, j,
                        "Left" if j > 0 else "stay")
```

```
def print_floor(floor, vacuum_direction):
    """ A function to print the floor,
    (x,y) represent the current vacuum
    cleaner position """

    print(f"The floor matrix is a DeLise:")
    for x in range(len(floor)):
        for c in range(len(floor[x])):
            if x == vc and c == vc:
                print(f"> {floor[x][c]} <",
                      end = " ")
            else:
                print(f" {floor[x][c]} ",
                      end = " ")
        print(end = '\n')
    print(f"& direction {}:")
    print(end = '\n')

def main():
    floor = []
    m = int(input("Enter the number of rows:"))
    n = int(input("Enter the number of columns:"))
    print("Enter clean status for each cell
          (1 - dirty, 0 - clean)")
    for i in range(m):
        f = list(map(int, input().split(" ")))
        floor.append(f)
    print()
    clean(floor)

main()
```

Output for two rooms

Enter the no of rows : 1
Enter the no of columns : 2
Enter clean status for each cell
(1 - dirty, 0 - clean)

1 1

The floor matrix is as below :

>1< 1
clean

The floor matrix is as below :
>0< 1
Right

The floor matrix is as below :
0 >1<
clean

The floor matrix is as below :
0 >0<
Stay

Output for four rooms

Enter the no of rows : 2
Enter the no of columns : 2

Enter the no of rows (1 - dirty, 0 - clean)
1 0
1 0

The floor matrix is as below:

>1< 0
1 0

clean

The floor matrix is as below.

>0< 0
1 0

Right

The floor matrix is as below:

0 >0<
1 0

Stay

The floor matrix is as below:

0 0
1 >0<

Down

The floor matrix is as below:

0 0
>1< 0

clean

The floor matrix is as below:

0 0
>0< 0

Stay.

29/12/23

Lab program - 6

Knowledge base entailment

Algorithm

Step 1 : Tokenigation:

Tokenize both 'sentence 1' and 'sentence 2' into individual words, considering punctuation and whitespaces.

Step 2: Normalization:

Convert all words to lowercase to make the comparision case-insensitive.

Step 3: comparision:

Initialize a boolean variable,
• entailment Holds to true.

Step : End:

End of algorithm.

Code.

```
from sympy.logic.boolalg import Implies,
                                NOT
from sympy.abc import p, q
```

```
class PropositionalKnowledge:
    def __init__(self):
        self.knowledge_base = set()

    def add_statement(self, statement):
        return query.simplify() in self.knowledge_base

Kb = PropositionKnowledgeBase()

Kb.add_statement(Implies(p, q))
Kb.add_statement(Not(q))

query1 = p
queryR = Not(p)

result1 = Kb.check_entailment(query1)
result R = Kb.check_entailment(query)

print(f"Does {query1} logically follow from the knowledge base? {result1}")

print(f"Does {query2} logically follow from the knowledge base? {result2}")
```

Output

Does 'p' logically follow from the knowledge base? True.

Does '~p' logically follow from the knowledge base? False.

Lab program - 7

Knowledge base resolution

Algorithm

Step 1: Initialize Knowledge Base:

- Create a class KnowledgeBase to represent the knowledge base.
- Populate the knowledge base with a set of predefined questions and answers.

Step 2: User interaction loop

- Enter a loop to interact with the user.
- Prompt the user to ask a question or type exit to end the program.

Step 3: User input

- Receive user input for the question.

Step 4: Knowledge Resolution

- Use the resolve_query method in the "KnowledgeBase" class to find the answer to the user's question.
- If an answer is found, display the answer.

Step: Exit condition

If the user types exit, the loop and end the program.

Code

```
def resolution(knowledge_base, query):

    clauses = knowledge_base + [frozenset({~query})]

    while True:
        new_clauses = set()

        for i in range(len(clauses)):
            for j in range(i+1, len(clauses)):
                resolvents = resolve(clauses[i],
                                     clauses[j])
                if frozenset() in resolvents:
                    new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):
            return False

        clauses.update(new_clauses)

def resolve(ci, cj):
    resolvents = set()

    for li in ci:
        for lj in cj:
            if li == ~lj or ~li == lj:
                resolvent = (ci - {li}) | (cj - {lj})
                resolvents.add(frozenset(resolvent))
    return resolvents
```

```
knowledge_base = []

while True:
    clause = input("Enter a clause for the knowledge base (or 'done' to finish): ").strip()

    if clause.lower() == 'done':
        break
    knowledge_base.append(frozenset(eval(clause)))

query = eval(input("Enter the query: "))

result = resolution(knowledge_base, query)

print(f"Is {query} entailed by the knowledge base? {result}")
```

Output

```
Enter a clause for the knowledge base (or 'done' to finish): {p, ~q}
Enter a clause for the knowledge base (or 'done' to finish): {~p, q}
Enter a clause for the knowledge base (or 'done' to finish): done
{q, ~q}
```

Enter a clause for the knowledge base
(or 'done' to finish): done

Enter the query: p

Is p resolved by the knowledge base?
False

Date 19/1/24
Page

Lab program - 8

Unification in first order logic

Algorithm

Step 1: Initialize: Start with an empty substitution "theta = {}"

Step 2: Comparision of atoms: compare the two atoms ["P", "x", "y"] and ["P", "A", "B"]
The first elements "P" are same
The second elements "x" and "A" are different, so we need to unify them

Step 3: Unify variables: unify the variable "x" with the term "A"
Update "theta" with this substitution
theta = {"x": "A"}

Step 4: Recursive unification: Now, recursively unify the remaining elements:
- For the third elements "y" and "B", unify the variable "y" with the term "B". Update theta theta = {"x": "A", "y": "B"}

Step 5: Result. The final substitution theta is {"x": "A", "y": "B"}
the two expressions ["P", "x", "y"] and ["P", "A", "B"] can be unified

Code

```
class UnificationError(Exception):
    pass

def print_step(step, atom1, atom2, theta):
    print(f"\n step {step}:")
    print(f" Unifying {atom1} and
                {atom2}")
    print(f" Current Substitution Theta:
            {theta}")

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    else x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def unify(atom1, atom2, theta = None, step=1):
    if theta is None:
        theta = {}

print_step(step, atom1, atom2, theta)

    if atom1 == atom2:
        return theta
    elif isinstance(atom1, str) and
        atom1.isalpha():
        return unify_var(atom1, atom2,
                         theta)
```

```
if len(atom1) != len(atom2):
    raise UnificationError("Lists have different lengths")
for a1, a2 in zip(atom1, atom2):
    theta = unify(a1, a2, theta, step)
return theta
else:
raise UnificationError("Cannot unify")

try:
    theta = unify(["P", "x", "y"],
                  ["P", "A", "B"])
    print("\n Unification successful, theta
except UnificationError as e
    print("\n Unification failed:", e)
```

Output

Step 1 : Unifying ['P', 'x', 'y'] and ['P', 'A', 'B']
current substitution Theta : {}

Step 2 : Unifying P and P
Current substitution Theta : {}

Step 3 : Unifying x and A
Current substitution Theta : {x/A}

Step 4 : Unifying y and B
current substitution Theta : {'x': 'A', 'y': 'B'}

Unification successful
substitution theta : {'x': 'A', 'y': 'B'}

Lab program - 9

Convert a first order logic to conjunctive normal form

Algorithm

Step 1: Eliminate implications
Replace implications with their equivalent forms using the rule $p \Rightarrow q \equiv \neg p \vee q$

Step 2: Move negations inwards
Apply DE Morgan's law to move negation inwards.
For example
$\neg(p \wedge q) \equiv \neg p \vee \neg q$
$\neg(p \vee q) \equiv \neg p \wedge \neg q$

Step 3: Distribute disjunctions over conjunctions
Distribute disjunctions ($\vee$) over conjunctions ($\wedge$) using the distributive property.
For example $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

Step 4: Convert to CNF form
Ensure that formula is in CNF by applying any additional simplifications if needed

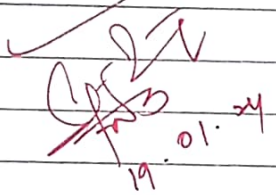
Code

```
from sympy import symbols, And, Or, Implies, Not, to_cnf

def eliminate_implications(formula):
    return formula.subs(Implies(p, q), Or(Not(p), q))

def move_negations_inwards(formula):
    return formula.simplify()

def distribute_disjunctions_over_conjunctions(formula):
    return formula.simplify()

def fol_to_cnf(fol_formula):
    formula_step1 = eliminate_implications(fol_formula)

    formula_step2 = move_negations_inwards(formula_step1)

    formula_step3 = distribute_disjunctions_over_conjunctions(formula_step2)

    cnf_formula = to_cnf(formula_step3)

    return cnf_formula
```

```
user_input = input ("Enter a first order
                    logic formula:")
p, q, r = symbols ('p q r')
fol_formula = eval (user_input, {'p': p,
                    'q': q, 'r': r})

cnf_formula = fol_to_cnf (fol_formula)
print ("\nFOL Formula:", fol_formula)
print ("CNF Formula:", cnf_formula)
```

Output

```
Enter a first order logic: p & (~q | r)
FOL formula: And(p, Or(Not(q), r))
CNF Formula: Or(And(p, r), And
                (p, ~q))
∨(∧(p, r) ∧(p, ~q))
```



Lab program - 10

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm

1. Initialize Knowledge Base (KB)
   Start with an empty knowledge base.

2. Add FOL statements to KB
   Add relevant FOL statements to the knowledge base. These statements represent facts and rules about the domain.

3. Define forward reasoning rules
   Specify the rules for forward reasoning.

4. Initalize working memory
   Create a working memory to store the current state of the knowledge base during the reasoning process.

5. Ask Query
   Formulate the query you want to prove.

6. Forward reasoning loop:

Repeat the following steps until the query

a. Iterate through each rule.

b. Apply rules whose conditions match the current state

c. Add the conclusions of the applied

7. Check query in working memory
Verify if query is now proved or can't be inferred

8. output result

If the query is proven output "Query is true" otherwise output Query is false.

Code

```
class KnowledgeBase:
    def __init__(self):
        self.rules = []

    def add_rule(self, conditions, conclusion):
        self.rules.append({"conditions":
            set(conditions), "conclusion":
            conclusion})
```

```
def forward_reasoning(self, query):
    working_memory = set()
    unchanged = False

    while not unchanged:
        unchanged = True
        for rule in self.rules:
            if rule['conditions'].issubset
            (working_memory) and rule
            ['conclusion'] not in working
                    memory.
                working_memory.add(rule['conclusion'])
                Unchanged = False

    return query in working_memory

kb = KnowledgeBase()
kb.add_rule(["P", "Q"], "R")
kb.add_rule(["R", "S"], "T")
kb.add_rule(["T", "U"], "V")

query = "V"

result = kb.forward_reasoning(query)

if result:
    print("Query is true")
else:
    print("Query is false")
```

Output

Enter the query.

(["P", "Q"], "R")
(["R", "S"], "T")
(["T", "U"], "V")

Query is false.

Enter the query

(["P", "Q"], "R")
(["Q", "R",], "P")
(["R", "P",], "Q")

Query is true.

19.01.24