VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence
(22CS5PCAIN)

Submitted by

STUTI UNIYAL (1BM21CS220)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Nov-2023 to Feb-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Artificial Intelligence**" carried out by **STUTI UNIYAL (1BM21CS220),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester Nov-2023 to Feb-2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Sneha S Bagalkot
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table of Contents

TIC - TAC - TOE

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

def spaceIsFree(position):
if board[position] == ' ':
    return True
else
    return False

def insertLetter(letter, position):
if spaceIsFree(position):
board[position] = letter
printBoard(board)
if(checkDraw()):
    print("Draw!")
    exit()
if checkForWin():
    if letter == 'X':
        print("Bot wins!")
        exit()
    else:
        print("Player wins!")
        exist()

    return

else:
    print("Can't insert there!")
    position = int(input("Please enter new position:"))
    insertLetter(letter, position)
    return
```

```
def checkforWin():
    if(board[1]==board[2] and board[1]==board[3]
       and board[1]!=' '):
        return True
    elif(board[4]==board[5] and board[4]==board[6]
       and board[4]!=' '):
        return True
    elif(board[7]==board[8] and board[7]==board[9]
    and board[7]!=' '):
        return True
    elif(board[1]==board[4] and board[1]==
        board[7] and board[1]!=' '):
        return True
    elif(board[2]==board[5] and board[2]==
            board[8]
        and board[2]!=' '):
        return True
    elif(board[3]==board[6] and board[3]==
            board[9]
        and board[3]!=' '):
        return True
    elif(board[1]==board[5] and board[1]==board[9] and
            board[1]!=' '):
        return True
    elif(board[7]==board[5] and board[7]==board[3]
        and board[7]!=' '):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if(board[key]==' '):
            return False
    return True
```

```
def checkDraw():
    for

def playerMove():
    position = int(input("Enter the position for
                         'O':"))
    insertLetter(player, position)

def compMove():
    bestScore = -800
    bestMove = 0
    for key in board.keys():
        if(board[key]==' '):
            board[key] = bot
            score = minimax(board, 0, False)
            board[key] = ' '
            if(score > bestScore):
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)
    return

def minimax(board, depth, isMaximizing):
    if(checkWhichMarkWon(bot)):
        return 1
    elif(checkWhichMarkWon(player)):
        return -1
    elif(checkDraw()):
        return 0

    if(isMaximizing):
        bestScore = -800
        for key in board.keys():
            if(board[key]==' '):
                board[key]=1
                if(score < bestScore):
                    bestScore = score
                return bestScore
```

```
else:
    bestScore = -800
    for key in board.keys():
        if(board[key] == ' '):
            board[key] = player
            score = minimax(board, depth+1, True)
            board[key] = ' '
            if(score < bestScore):
                bestScore = score
    return bestScore

board = {1: ' ', 2: ' ', 3: ' ', 4: ' ', 5: ' ', 6: ' ',
         7: ' ', 8: ' ', 9: ' '}

printBoard(board)
print("Computer goes first! Good luck.")
print("Positions are as follows:")
print("1, 2, 3")
print("4, 5, 6")
print("7, 8, 9")
print("\n")
player = 'O'
bot = 'X'

global firstComputerMove
firstComputerMove = True

while not checkForWin():
    compMove()
    playerMove()
```

OUTPUT

```
   |   |
---+---+---
   |   |
---+---+---
   |   |
```

computer goes first! Good luck.
positions are as follows:

```
1, 2, 3
4, 5, 6
7, 8, 9
```

```
X |   |
--+---+---
  |   |
--+---+---
  |   |
```

Enter the position for 'O': 7

```
X |   |
--+---+---
  |   |
--+---+---
O |   |
```

```
X | X |
--+---+---
  |   |
--+---+---
O |   |
```

Enter position for 'O': 3

```
X | X | O
--+---+---
  |   |
--+---+---
O |   |
```

```
X | X | O
--+---+---
  | X |
--+---+---
O |   |
```

Enter the position for 'O': 8

```
X | X | O
--+---+---
  | X |
--+---+---
O | O |
```

⇒

```
X | X | O
--+---+---
  | X |
--+---+---
O | O | X
```

BOT WINS!

Vaccume Cleaner

```
def vaccume_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of Vaccume")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter the status of other room")
    print("Initial location condition" + str(goal_state))
if location_input == 'A':
    print("Vaccume is placed in Location A")
    if status_input == '1':
        print("Location A is Dirty.")
        goal_state['A'] = '0'
        cost += 1
        print("Cost for CLEANING A" + str(cost))
        print("Location A has been cleared")
        if status_input_complement == '1':
            print("Location B is Dirty")
            print("Moving right to the location B")
            cost += 1
            print("COST for moving RIGHT" + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been cleaned.")
    else:
        print("no action" + str(cost))
        print("Location B is already clean.")
    if status_input == '0':
        print("Location A is already clean")
        if status_input_complement == '1':
            print("Location B is Dirty")
            print("Moving RIGHT to location B.")
            cost += 1
            print("Cost of moving RIGHT" + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been cleaned.")
else:
    print("Vaccume is placed in location B")
    if status_input == 1:
        print("Location B is Dirty")
        goal_state['B'] = '0'
        cost += 1
        print("COST of CLEANING" + str(cost))
        print("Location B has been cleaned.")
    if status_input_complement == '1':
        print("Location A is Dirty")
        print("Moving left to location A.")
        cost += 1
        print("COST for moving LEFT" + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("COST FOR SUCK" + str(cost))
        print("Location A has been cleaned.")
    else:
        print(cost)
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("moving LEFT to the location A")
            cost += 1
```

```
        print("Cost for moving LEFT" + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("Cost for SUCK" + str(cost))
        print("Location A has been cleaned.")
    else:
        print("No action" + str(cost))
        print("Location A is already clean.")

print("GOAL STATE:")
print(goal_state)
print("Performance Measurement:" + str(cost))

vacuum_world()
```

OUTPUT:

```
Enter Location of Vacuum A
Enter status of A 1
Enter status of other room 1
Initial Location Condition {'A': '0', 'B': '0'}
Vacuume is placed in Location A
Location A is Dirty
Cost of cleaning A 1
Location A has been Cleaned
Location B is dirty
Moving right to the location B
Cost for moving RIGHT 2
Cost for SUCK 3
Location B has been Cleaned
GOAL STATE:
{'A': '0', 'B': '0'}
```

8PUZZLE PROBLEM USING BFS

```
import numpy as np
import pandas as pd
import os
def bfs(src, target)
    queue=[]
    queue.append(src)
    exp=[]
    while len(queue)>0:
        source = queue.pop(0)
        exp.append(source)
        print(source)
        if source == target:
            print("success")
            return
        poss_moves_to_do=[]
        poss_moves_to_do = possible_moves(source, exp)
        for move in poss_moves_to_do:
            if move not in exp and move not in queue:
                queue.append(move)
    def possible_moves(state, visited_states):
        b = state.index(-1)
        d=[]
        if b not in [-1,1,2]:
            d.append('u')
        if b not in [6,7,8]:
            d.append('d')
        if b not in [-1,3,6]:
            d.append('l')
        if b not in [2,5,8]:
            d.append('r')
        pos_moves_it_can=[]
        for i in d:
            pos_moves_it_can.append(gen(state,i,b))
        return [move_it_can for move_it_can in pos_moves_it_can
                if move_it_can not in visited_states]
    def gen(state,m,b):
        temp = state.copy()
        if m=='d':
            temp[b+3], temp[b] = temp[b], temp[b+3]
        if m=='u':
            temp[b-3], temp[b] = temp[b], temp[b-3]
        if m=='l':
            temp[b-1], temp[b] = temp[b], temp[b-1]
        if m=='r':
            temp[b+1], temp[b] = temp[b], temp[b+1]
        return temp
```

OUTPUT

src = [1,2,3,-1,4,5,6,7,8]

target = [1,2,3,4,5,-1,6,7,8]

bfs(src, target)

[1,2,3,-1,4,5,6,7,8]
[-1,2,3,1,4,5,6,7,8]
[1,2,3,6,4,5,-1,7,8]
[1,2,3,4,7,5,6,7,8]
[6,2,3,1,4,5,-1,7,8]
[8,2,3,1,4,5,6,7,-1]
[2,-1,3,1,4,5,6,7,8]
[1,2,3,6,4,5,7,-1,8]
[1,-1,3,4,2,5,6,7,8]
[1,2,3,4,7,5,6,-1,8]
[1,2,3,4,5,-1,6,7,8]

success

6/12/23

8 PUZZLE PROBLEM USING IDS

```
class PuzzleNode:
    def __init__(self, state, parent=None, action=None):
        self.state = state
        self.parent = parent
        self.action = action

    def get_path(self):
        path = []
        current = self
        while current:
            path.append((current.state, current.action))
            current = current.parent
        return path[::-1]

    def is_goal(state):
        goal_state = (1,2,3,6,4,5,0,7,8)
        return state == goal_state

    def get_neighbours(state):
        neighbours = []
        empty_index = state.index(0)
        row, col = divmod(empty_index, 3)

        for move in [(0,1), (1,0), (0,-1), (-1,0)]:
            new_row, new_col = row + move[0], col + move[1]
            if 0<=new_row<3 and 0<=new_col<3:
            neighbor_state = list(state)
            neighbor_index = new_row*3 + new_col
            neighbor_state[empty_index], neighbor_state[neighbor_index] =(
            neighbor_state[neighbor_index],
            neighbor_state[empty_index],
            )
            neighbors.append(tuple(neighbor_state))
        return neighbors
```

```
def depth_limited_search(node, goal_state,
                         depth_limit):
    if is_goal(node.state):
        return True
    elif depth_limit == 0:
        return False
    else:
        for neighbor_state in get_neighbors(node.state):
            child = PuzzleNode(neighbor_state, node)
            if depth_limited_search(child, goal_state,
                                    depth_limit-1):
                return True
        return False

if __name__ == "__main__":
    initial_state = (1, 2, 3, 0, 4, 5, 6, 7, 8)
    depth_limit = 1
    initial_node = PuzzleNode(initial_state)
    result = depth_limited_search(initial_node,
                (1, 2, 3, 6, 4, 5, 0, 7, 8), depth_limit)
    print(result)
```

OUTPUT:
True

6/12/23

8 PUZZLE PROBLEM USING GREEDY BEST FIRST SEARCH

```
class PuzzleNode:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.cost = 0
        self.heuristic = self.calculate_heuristic()

    def calculate_heuristic(self):
        goal_state = [[1,2,3],[4,5,6],[7,8,0]]
        heuristic = 0
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != 0:
                    goal_row, goal_col = divmod(self.state[i][j]
                                                -1, 3)
                    heuristic += abs(i-goal_row) + abs(j-
                                                goal_col)
        return heuristic

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

def get_neighbors(node):
    i, j = get_blank_position(node.state)
    neighbors = []
    moves = [(0,1),(0,-1),(1,0),(-1,0)]

    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in node.state]
            new_state[i][j], new_state[new_i][new_j] =
            new_state[new_i][new_j], new_state[i][j]
            neighbors.append(PuzzleNode(new_state, node))
    return neighbors
```

```
def greedy_best_first_search(initial_state):
    start_node = PuzzleNode(initial_state)
    open_set = [start_node]
    closed_set = set()

    while open_set:
        open_set.sort()
        current_node = open_set.pop(0)

        if current_node.state == [[1, 2, 3], [4, 5, 6],
                                  [7, 8, 0]]:

            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in get_neighbors(current_node):
            if tuple(map(tuple, neighbor.state)) not in closed_set:
                neighbor.cost = current_node.cost + 1
                open_set.append(neighbor)

    return None

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

solution = greedy_best_first_search(initial_state)
if solution:
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution exists.")
```

OUTPUT:

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

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

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

A* algorithm

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '_')
        val_list = [[x, y-1], [x, y+1], [x-1, y],
                    [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0],
                                 i[1])
            if child is not None:
                child_node = Node(child, self.level+1,
                                  0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
        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):
        Temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j

class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] !=
                    temp += 1
        return temp
```

```
def process(self):
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix\n")
    goal = self.accept()
    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
        print(" ")
        print(" | ")
        print(" | ")
        print(" \\|/ \n")
        for i in cur.data:
            for j in i:
                print(j, end = " ")
            print("")
        if(self.h(cur.data,goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]

        self.open.sort(key= lambda x: x.fval,
                       reverse = False)
puz = Puzzle(3)
puz.process()
```

KNOWLEDGE BASE - PROPORTIONAL LOGIC

```
def evaluate_first(premise, conclusion):
    models = [
        {'p': False, 'q': False, 'r': False},
        {'p': False, 'q': False, 'r': True},
        {'p': False, 'q': True, 'r': False},
        {'p': False, 'q': True, 'r': True},
        {'p': True, 'q': False, 'r': False},
        {'p': True, 'q': False, 'r': True},
        {'p': True, 'q': True, 'r': False},
        {'p': True, 'q': True, 'r': True}
    ]

    entails = True
    for model in models:
        if evaluate_expression(premise, model) != evaluate_
                                    expression(conclusion, model):
            entails = False
            break
    return entails

def evaluate_second(premise, conclusion):
    models = [
        {'p': p, 'q': q, 'r': r}
        for p in [True, False]
        for q in [True, False]
        for r in [True, False]
    ]
    entails = all(evaluate_expression(premise, model) for
    model in models if evaluate_expression(conclusion,
                                           model)
    and evaluate_expression(premise, model))
    return entails
def evaluate_expression(expression, model):
    if isinstance(expression, str):
        return model.get(expression)
```

```
elif isinstance(expression, tuple):
    op = expression[0]
    if op == 'not':
        return not evaluate_expression(expression[1], model)
    elif op == 'and':
        return evaluate_expression(expression[1], model) and evaluate_expression(expression[2], model)
    elif op == 'if':
        return (not evaluate_expression(expression[1], model)) or evaluate_expression(expression[2], model)
    elif op == 'or':
        return evaluate_expression(expression[1], model) or evaluate_expression(expression[2], model)

first_premise = ('and', ('or', ('not', 'p'), 'q'), ('or', ('not', 'r'), 'p'))
first_conclusion = ('and', 'p', 'r')
second_premise = ('and', ('or', ('not', 'q'), ('not', 'p'), 'r'), ('and', ('not', 'q'), 'p'), 'q')
second_conclusion = 'r'
result_first = evaluate_first(first_premise, first_conclusion)
result_second = evaluate_second(second_premise, second_conclusion)
if result_first:
    print("For the first input: The premise entails the conclusion.")
if result_second:
    print("for the second input: The premise entails the conclusion.")
else:
    print("for the second input: the premise doesn't entail the conclusion.")
```

OUTPUT:

For the first input: The premise doesn't entail the conclusion.

For the second input: The premise entails the conclusion

OUTPUT: (A* Algorithm)

Enter the start state matrix

1 2 3
4 5 6
0 7 8

Enter the goal state matrix

1 2 3
4 5 6
7 8 0

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

LAB-7

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t| Clause\t| Derivation\t|')
    print('-'*30)
    i = 1
    for step in steps:
        print(f'{i}.\t| {step}\t| {steps[step]}\t|')
        i += 1

def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]} V {t[0]}'
    return ''

def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_terms('~PVR')

def contradiction(goal, clause):
    contradictions = [f'{goal} V {negate(goal)}',
                      f'{negate(goal)} V {goal}']
    return clause in contradictions or
           reverse(clause) in contradictions
```

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given'
    steps[negate(goal)] = 'Negated conclusion'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i+1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if
                          t != c]
                    t2 = [t for t in terms2 if t !=
                          negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]} V {gen[1]}']
                    else:
                        if contradiction(goal, f'{gen[0]}
                                         {gen[1]}'):
                            temp.append(f'{gen[0]} V {gen[1]}
                            steps[''] = f"Resolved {temp[i]}
and {temp[j]} to {temp[-1]}, which is in turn null.
```

```
\n A contradiction is found when {negate(goal)}
 is assumed as true. Hence {goal} is true.'
        return steps.
elif len(gen) == 1:
    clauses += [f'{gen[0]}']
else:
    if contradiction(goal, f'{terms1[0]} V
                                {terms2[0]}'):
        temp.append(f'{terms1[0]} V {terms2[0]}')
        steps[''] = f"Resolved {temp[i]} and
                {temp[j]} to {temp[-1]},
which is in turn null. \n A contradiction
is found when {negate(goal)} is
assumed as true. Hence, {goal} is true."
        return steps
for clause in clauses:
    if clause not in temp and clause != 
            reverse(clause) and
reverse(clause) not in temp:
    temp.append(clause)
    steps[clause] = f'Resolved from
{temp[i]} and {temp[j]},'
    j = (j+1) % n
i += 1
return steps
```

```
rules = 'RV~P RVQ ~RVP ~RVQ'
goal = 'R'
main(rules, goal)

rules = 'PVQ ~PVR ~QVR'
goal = 'R'
main(rules, goal)

rules = 'RVQ PVR ~PVR RVS RV~Q ~SV~Q'
main(rules, 'R')
```

OUTPUT :

Step	Clause	Derivation
1.	RV~P	Given
2.	RV~Q	Given
3.	~RVP	Given
4.	~RVQ	Given
5.	~R	negated conclusion
6.		Resolved RV~P and ~RVP to RV~R, which is in turn null

A contradiction is found when ~R is assumed as true.

Step	Clause	Derivation
1.	PVQ	Given
2.	~PVR	Given
3.	~QVR	Given
4.	~R	negated conclusion
5.	QVR	Resolved from PVQ and ~PVR
6.	PVR	Resolved from PVQ and ~QVR
7.	~P	Resolved from PVR and ~R
8.	~Q	

9. Q Resolved from ~R and QVR
10. P resolved from ~R and PVR
11 R resolved from QVR and ~Q
12 resolved R and ~R to RV~R, which is in turn true

27/12/23

LAB-8

UNIFICATION IN FIRST ORDER LOGIC

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\\(.), (?!.\))", expression)
    return expression
def getInitialPredicate(expression):
    return expression.split("(")[0]
def isConstant(char):
    return char.isupper() and len(char)==1
def isVariable(char):
    return char.islower() and len(char)==1
def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"
def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True
```

```
def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]
def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(
                                attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp2):
        return [(exp1, exp2)]
    if isConstant(exp2):
        return [(exp2, exp1)]
    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
        else:
            return [(exp2, exp1)]
    if isVariable(exp2):
        if checkOccurs(exp2, exp1):
            return False
```

```
        else:
            return [(exp1, exp2)]
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates do not match. Cannot be
                                    unified")
        return False
    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        return False
    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubsitution = unify(head1, head2)
    if not initialSubsitution:
        return False
    if attributeCount1 == 1:
        return initialSubsitution
    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)
    if initialSubsitution != []:
        tail1 = apply(tail1, initialSubsitution)
        tail2 = apply(tail2, initialSubsitution)
    remainingSubsitution = unify(tail1, tail2)
    if not remainingSubsitution:
        return False
    initialSubsitution.extend(remainingSubsitution)
    return initialSubsitution
```

```
exp1 = "knows(A,x)"
exp2 = "knows(y,y)"
substitutions = unify(exp1, exp2)
print("substitutions:")
print(substitutions)

exp1 = "knows(A,x)"
exp2 = "knows(y, mother(y))"
substitutions = unify(exp1, exp2)
print("substitutions:")
print(substitutions)
```

OUTPUT:

substitutions:

[('A', 'y'), ('y', 'x')]

substitutions:

[('A', 'y'), ('mother(y)', 'x')]

10/1/24

WEEK - 9

17/1/24

CONVERT FIRST ORDER LOGIC TO CNF

```
import re
def getattributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getpredicates(string):
    expr = '[a-zA]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in
                        range(ord('A'), ord('Z')+1)]
    matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if '' .join(attributes).islower():
                statement = statement.replace(
                    match[1], SKOLEM_CONSTANTS.pop(0))
    return statement
def fol_to_cnf(fol):
    statement = fol.replace("=>","-")
```

```
expr = r'\[([^\]]+)\]'
statements = re.findall(expr, statement)
for i,s in enumerate(statements):
    if '[' in s and ']' not in s:
        statements[i] += ']'

for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))

while '-' in statement:
    i = statement.index('-')
    br = statement.index('[') if
        '[' in statement else 0
    new_statement = '~' + statement[br:i]
        + '|' + statement[i+1:]
    statement = statement[:br] +
        new_statement if br > 0 else
        new_statement
return Skolemization(statement)
print(fol_to_cnf("bird(x) => ~fly(x)"))
print(fol_to_cnf("∃x[bird(x) => ~fly(x)]"))
```

OUTPUT:

~bird(x) | ~fly(x)
[~bird(A) | ~fly(A)]

17/1/24

WEEK 10

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

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and
        x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches
def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(
            expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].
            strip('()').split(',')
        return [predicate, params]
    def getResult(self):
        result self.result
    def getConstants(self):
        result [None if isVariable(c) else c for c in
            self.params]
```

```
def getVariables(self):
    return [v if isVariable(v) else None for v
            in self.params]

def substitute(self, constants):
    c = constants.copy()
    f = f"{self.predicate}({','.join([
        constants.pop(p) if isVariable(p) else
        p for p in self.params])})"
    return Fact(f)

class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].
                    split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVa-
                                          riables()):
                        if v:
                            constants[v] = fact.getConstants
                                           ()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.
                                expression)[0],
        str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[
                             key])
        expr = f'{predicate}({attributes})'
        return Fact(expr) if len(new_lhs) and
            all([f.getResult() for f in new_lhs]) else
            None

class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else: self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in
                     self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predi
                                    cate:
                print(f'\t{i}. {f}')
                i += 1
```

```
def display(self):
    print("All facts: ")
    for i, f in enumerate(set([f.expressi
        for f in self.facts])):
        print(f'\t{i+1}. {f}')

kb_ = KB()
kb_.tell('king(x) & greedy(x) => evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
kb_.display()
```

OUTPUT:

```
Querying evil(x):
        1. evil(John)

All facts:  king(John)
        1. king(Richard)
        2. evil(John)
        3. greedy(John)
```

Sneha
25/1/24

1.TIC-TAC-TOE

```
def printBoard(board):
print(board[1] + '|' + board[2] + '|' + board[3])
print('-+-+-')
print(board[4] + '|' + board[5] + '|' + board[6])
print('-+-+-')
print(board[7] + '|' + board[8] + '|' + board[9])
print("\n")
def spaceIsFree(position):
if board[position] == ' ':
return True
else:
return False
def insertLetter(letter, position):
if spaceIsFree(position):
board[position] = letter
printBoard(board)
if (checkDraw()):
print("Draw!")
exit()
if checkForWin():
if letter == 'X':
print("Bot wins!")
exit()
else:
print("Player wins!")
exit()
return
else:
```

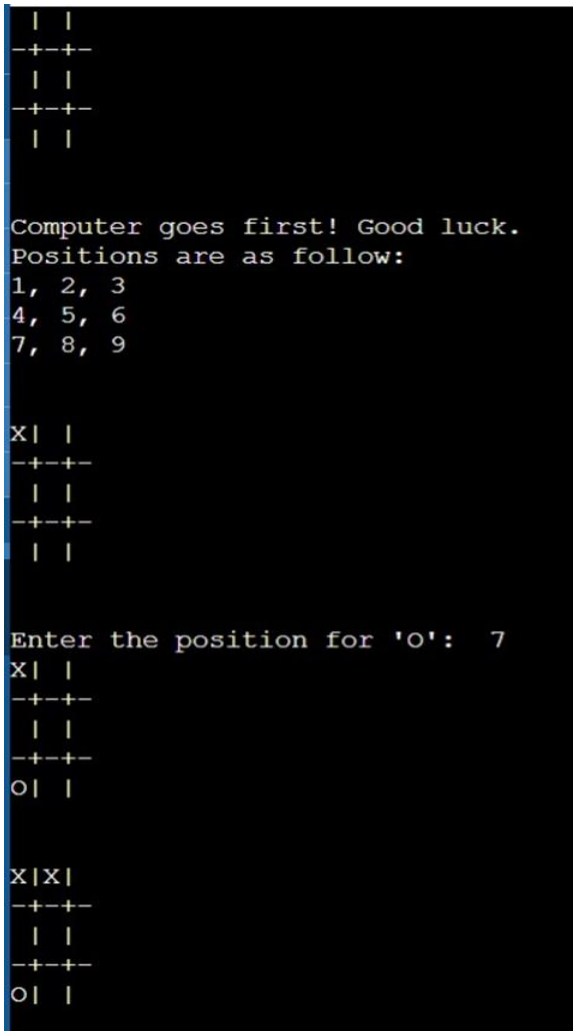
```
print("Can't insert there!")
position = int(input("Please enter new position: "))
insertLetter(letter, position)
return
def checkForWin():
if (board[1] == board[2] and board[1] == board[3] and board[1] != ' '):
return True
elif (board[4] == board[5] and board[4] == board[6] and board[4] != ' '):
return True
elif (board[7] == board[8] and board[7] == board[9] and board[7] != ' '):
return True
elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
return True
elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
return True
elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
return True
elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
return True
elif (board[7] == board[5] and board[7] == board[3] and board[7] != ' '):
return True
else:
return False
def checkWhichMarkWon(mark):
if board[1] == board[2] and board[1] == board[3] and board[1] == mark:
return True
elif (board[4] == board[5] and board[4] == board[6] and board[4] == mark):
return True
elif (board[7] == board[8] and board[7] == board[9] and board[7] == mark):
return True
```

```
elif (board[1] == board[4] and board[1] == board[7] and board[1] == mark):
return True
elif (board[2] == board[5] and board[2] == board[8] and board[2] == mark):
return True
elif (board[3] == board[6] and board[3] == board[9] and board[3] == mark):
return True
elif (board[1] == board[5] and board[1] == board[9] and board[1] == mark):
return True
elif (board[7] == board[5] and board[7] == board[3] and board[7] == mark):
return True
else:
return False
def checkDraw():
for key in board.keys():
if (board[key] == ' '):
return False
return True
def playerMove():
position = int(input("Enter the position for 'O': "))
insertLetter(player, position)
return
def compMove():
bestScore = -800
bestMove = 0
for key in board.keys():
if (board[key] == ' '):
board[key] = bot
score = minimax(board, 0, False)
board[key] = ' '
if (score > bestScore):
```

```
bestScore = score
bestMove = key
insertLetter(bot, bestMove)
return
def minimax(board, depth, isMaximizing):
if (checkWhichMarkWon(bot)):
return 1
elif (checkWhichMarkWon(player)):
return -1
elif (checkDraw()):
return 0
if (isMaximizing):
bestScore = -800
for key in board.keys():
if (board[key] == ' '):
board[key] = bot
score = minimax(board, depth + 1, False)
board[key] = ' '
if (score > bestScore):
bestScore = score
return bestScore
else:
bestScore = 800
for key in board.keys():
if (board[key] == ' '):
board[key] = player
score = minimax(board, depth + 1, True)
board[key] = ' '
if (score < bestScore):
bestScore = score
```

```
return bestScore
board = {1: ' ', 2: ' ', 3: ' ',
4: ' ', 5: ' ', 6: ' ',
7: ' ', 8: ' ', 9: ' '}
printBoard(board)
print("Computer goes first! Good luck.")
print("Positions are as follow:")
print("1, 2, 3 ")
print("4, 5, 6 ")
print("7, 8, 9 ")
print("\n")
player = 'O'
bot = 'X'
global firstComputerMove
firstComputerMove = True
while not checkForWin():
compMove()
playerMove()
```

OUTPUT


| |
-+-+-
| |
-+-+-
| |
Computer goes first! Good luck.
Positions are as follow:
1, 2, 3
4, 5, 6
7, 8, 9
X| |
-+-+-
| |
-+-+-
| |
Enter the position for 'O': 7
X| |
-+-+-
| |
-+-+-
O| |
X|X|
-+-+-
| |
-+-+-
O| |

```
Enter the position for 'O':  3
X|X|O
-+-+-
 | |
-+-+-
O| |


X|X|O
-+-+-
 |X|
-+-+-
O| |


Enter the position for 'O':  8
X|X|O
-+-+-
 |X|
-+-+-
O|O|


X|X|O
-+-+-
 |X|
-+-+-
O|O|X


Bot wins!
```

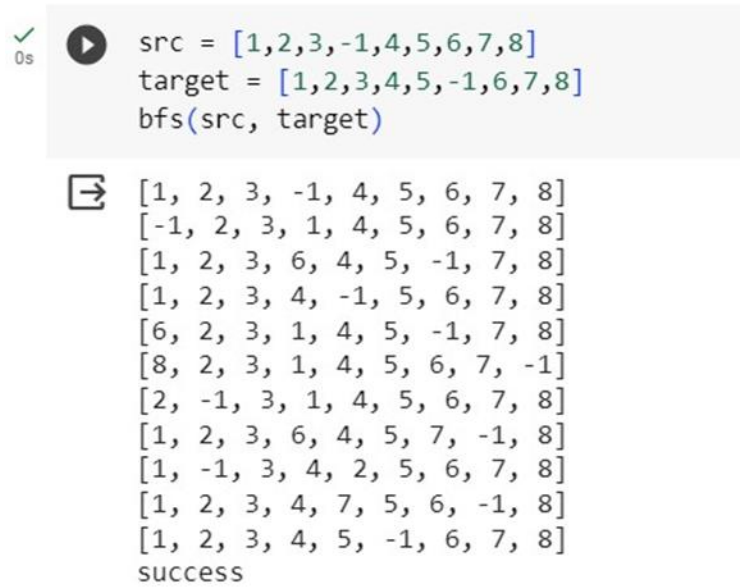
2. 8 Puzzle
(bfs)

```
import numpy as np
import pandas as pd
import os

def bfs(src,target):
    queue = []
    queue.append(src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print(source)

        if source==target:
            print("success")
            return

        poss_moves_to_do = []
        poss_moves_to_do = possible_moves(source,exp)

        for move in poss_moves_to_do:

            if move not in exp and move not in queue:
                queue.append(move)
```

```
def possible_moves(state,visited_states):
    #index of empty spot
    b = state.index(-1)

    #directions array
    d = []
    #Add all the possible directions

    if b not in [-1,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [-1,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')



    # If direction is possible then add state to move
    pos_moves_it_can = []


    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the
given directions


    for i in d:
        pos_moves_it_can.append(gen(state,i,b))


    return [move_it_can for move_it_can in pos_moves_it_can if
move_it_can not in visited_states]


def gen(state, m, b):
```

```
    temp = state.copy()

    if m=='d':
        temp[b+3],temp[b] = temp[b],temp[b+3]

    if m=='u':
        temp[b-3],temp[b] = temp[b],temp[b-3]

    if m=='l':
        temp[b-1],temp[b] = temp[b],temp[b-1]

    if m=='r':
        temp[b+1],temp[b] = temp[b],temp[b+1]


    # return new state with tested move to later check if "src ==
target"
    return temp
```

OUTPUT



0s

```
src = [2,-1,3,1,8,4,7,6,5]
target = [1,2,3,8,-1,4,7,6,5]
bfs(src, target)
```

```
[2, -1, 3, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, -1, 4, 7, 6, 5]
[-1, 2, 3, 1, 8, 4, 7, 6, 5]
[2, 3, -1, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, 6, 4, 7, -1, 5]
[2, 8, 3, -1, 1, 4, 7, 6, 5]
[2, 8, 3, 1, 4, -1, 7, 6, 5]
[7, 2, 3, 1, 8, 4, -1, 6, 5]
[1, 2, 3, -1, 8, 4, 7, 6, 5]
[5, 2, 3, 1, 8, 4, 7, 6, -1]
[2, 3, 4, 1, 8, -1, 7, 6, 5]
[2, 8, 3, 1, 6, 4, -1, 7, 5]
[2, 8, 3, 1, 6, 4, 7, 5, -1]
[-1, 8, 3, 2, 1, 4, 7, 6, 5]
[2, 8, 3, 7, 1, 4, -1, 6, 5]
[2, 8, -1, 1, 4, 3, 7, 6, 5]
[2, 8, 3, 1, 4, 5, 7, 6, -1]
[7, 2, 3, -1, 8, 4, 1, 6, 5]
[7, 2, 3, 1, 8, 4, 6, -1, 5]
[1, 2, 3, 7, 8, 4, -1, 6, 5]
[1, 2, 3, 8, -1, 4, 7, 6, 5]
success
```

3. Implement Iterative deepening search algorithm

```
def dfs(src,target,limit,visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False
    visited_states.append(src)
    moves = possible_moves(src,visited_states)
    for move in moves:
        if dfs(move, target, limit-1, visited_states):
            return True
    return False

def possible_moves(state,visited_states):
    b = state.index(-1)
    d = []
    if b not in [0,1,2]:
        d += 'u'
    if b not in [6,7,8]:
        d += 'd'
    if b not in [2,5,8]:
        d += 'r'
    if b not in [0,3,6]:
        d += 'l'
    pos_moves = []
    for move in d:
        pos_moves.append(gen(state,move,b))
    return [move for move in pos_moves if move not in visited_states]
```

```
def gen(state, move, blank):
    temp = state.copy()
    if move == 'u':
        temp[blank-3], temp[blank] = temp[blank], temp[blank-3]
    if move == 'd':
        temp[blank+3], temp[blank] = temp[blank], temp[blank+3]
    if move == 'r':
        temp[blank+1], temp[blank] = temp[blank], temp[blank+1]
    if move == 'l':
        temp[blank-1], temp[blank] = temp[blank], temp[blank-1]
    return temp

def iddfs(src,target,depth):
     for i in range(depth):
        visited_states = []
        if dfs(src,target,i+1,visited_states):
            return True
     return False

src = []
target=[]
n = int(input("Enter number of elements : "))
print("Enter source elements")
for i in range(0, n):
    ele = int(input())
    src.append(ele)
print("Enter target elements")
for i in range(0, n):
    ele = int(input())
```

```
    target.append(ele)
depth=8
iddfs(src, target,depth)
```

OUTPUT

```
Enter number of elements : 9
Enter source elements
1
2
3
-1
4
5
6
7
8
Enter target elements
1
2
3
4
5
-1
6
7
8
True
```

4. 8 Puzzle A* Search Algorithm

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

    def generate_child(self):
        # Generate hild nodes from the given node by moving the blank space
        # either in the four direction {up,down,left,right}
        x, y = self.find(self.data, '_')
        # val_list contains position values for moving the blank space in either of
        # the 4 direction [up,down,left,right] respectively.
        val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level + 1, 0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
```

```
    # Move the blank space in the given direction and if the position value are out
    # of limits the return None
    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 find(self, puz, x):
    # Specifically used to find the position of the blank space
    for i in range(0, len(self.data)):
        for j in range(0, len(self.data)):
            if puz[i][j] == x:
                return i, j
```

```
class Puzzle:
    def __init__(self, size):
        # Initialize the puzzle size by the the specified size,open and closed lists to empty
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        # Accepts the puzzle from the user
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        # Heuristic function to calculate Heuristic value f(x) = h(x) + g(x)
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        # Calculates the difference between the given puzzles
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

    def process(self):
        # Accept Start and Goal Puzzle state
```

```
        print("enter the start state matrix \n")
        start = self.accept()
        print("enter the goal state matrix \n")
        goal = self.accept()
        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        # put the start node in the open list
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("=================================================\n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
            # if the difference between current and goal node is 0 we have reached the goal node
            if (self.h(cur.data, goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]
            # sort the open list based on f value
            self.open.sort(key=lambda x: x.fval, reverse=False)
puz = Puzzle(3)
puz.process()
```

OUTPUT

```
enter the start state matrix

1 2 3
_ 4 6
7 5 8
enter the goal state matrix

1 2 3
4 5 6
7 8 _


==================================================

1 2 3
_ 4 6
7 5 8
==================================================

1 2 3
4 _ 6
7 5 8
==================================================

1 2 3
4 5 6
7 _ 8
==================================================

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

5.Vaccum cleaner

```
def vacuum_world():
    # initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
  goal_state = {'A': '0', 'B': '0'}
  cost = 0

  location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
  status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
  status_input_complement = input("Enter status of other room")
  print("Initial Location Condition" + str(goal_state))

  if location_input == 'A':
    # Location A is Dirty.
    print("Vacuum is placed in Location A")
    if status_input == '1':
```

```
    print("Location A is Dirty.")
    # suck the dirt  and mark it as clean
    goal_state['A'] = '0'
    cost += 1                    #cost for suck
    print("Cost for CLEANING A " + str(cost))
    print("Location A has been Cleaned.")

    if status_input_complement == '1':
        # if B is Dirty
        print("Location B is Dirty.")
        print("Moving right to the Location B. ")
        cost += 1                    #cost for moving right
        print("COST for moving RIGHT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['B'] = '0'
        cost += 1                    #cost for suck
        print("COST for SUCK " + str(cost))
        print("Location B has been Cleaned. ")
    else:
        print("No action" + str(cost))
        # suck and mark clean
        print("Location B is already clean.")

if status_input == '0':
    print("Location A is already clean ")
    if status_input_complement == '1':# if B is Dirty
        print("Location B is Dirty.")
        print("Moving RIGHT to the Location B. ")
        cost += 1                    #cost for moving right
        print("COST for moving RIGHT " + str(cost))
```

```
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                    #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
        goal_state['B'] = '0'
        cost += 1  # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
```

```
            cost += 1  # cost for suck
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        # suck and mark clean
        print("Location B is already clean.")

        if status_input_complement == '1':  # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")

# done cleaning
print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))
```

vacuum_world()

OUTPUT

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

```
vacuum_world()
```

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
No action 0
0
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0
```

```
Enter Location of VacuumB
Enter status of B0
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

6. Knowledge Base Entailment

```
def tell(kb, rule):
    kb.append(rule)


combinations = [(True, True, True), (True, True, False),
                (True, False, True), (True, False, False),
                (False, True, True), (False, True, False),
                (False, False, True), (False, False, False)]
```

```
def ask(kb, q):
    for c in combinations:
        s = all(rule(c) for rule in kb)
        f = q(c)
        print(s, f)
        if s != f and s != False:
            return 'Does not entail'
    return 'Entails'



kb = []


# Get user input for Rule 1
rule_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): ")
r1 = eval(rule_str)
tell(kb, r1)


# Get user input for Rule 2
#rule_str = input("Enter Rule 2 as a lambda function (e.g., lambda x: (x[0] or x[1]) and x[2]): ")
#r2 = eval(rule_str)
#tell(kb, r2)


# Get user input for Query
query_str = input("Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): ")
q = eval(query_str)
```

Ask KB Query

result = ask(kb, q)

print(result)

OUTPUT

```
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x:  (x[0] or x[1]) and ( not x[2] or x[0])
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x:  (x[0] and x[2])
True True
True False
Does not entail
```

7. Knowledge Base Resolution

import re

def main(rules, goal):

```
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1

def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''

def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']
    return clause in contradictions or reverse(clause) in contradictions

def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
```

```
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
```

```
                if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                    temp.append(f'{terms1[0]}v{terms2[0]}')
                    steps['] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                    \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."
                    return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps


rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
goal = 'R'
main(rules, goal)
 rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
main(rules, goal)
```

OUTPUT

```
Step    |Clause |Derivation
------------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.

Step    |Clause |Derivation
------------------------------
 1.     | PvQ   | Given.
 2.     | ~PvR  | Given.
 3.     | ~QvR  | Given.
 4.     | ~R    | Negated conclusion.
 5.     | QvR   | Resolved from PvQ and ~PvR.
 6.     | PvR   | Resolved from PvQ and ~QvR.
 7.     | ~P    | Resolved from ~PvR and ~R.
 8.     | ~Q    | Resolved from ~QvR and ~R.
 9.     | Q     | Resolved from ~R and QvR.
 10.    | P     | Resolved from ~R and PvR.
 11.    | R     | Resolved from QvR and ~Q.
 12.    |       | Resolved R and ~R to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

8. Unification

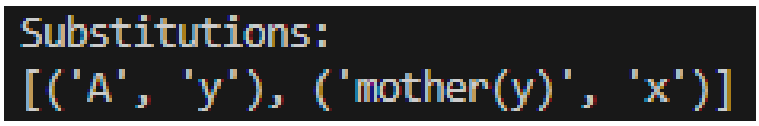
```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"
```

```
def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True

def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]

def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression

def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp1):
        return [(exp1, exp2)]
```

```
if isConstant(exp2):
    return [(exp2, exp1)]
if isVariable(exp1):
    if checkOccurs(exp1, exp2):
        return False
    else:
        return [(exp2, exp1)]
if isVariable(exp2):
    if checkOccurs(exp2, exp1):
        return False
    else:
        return [(exp1, exp2)]
if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Cannot be unified")
    return False
attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False
head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)
initialSubstitution = unify(head1, head2)
if not initialSubstitution:
    return False
if attributeCount1 == 1:
    return initialSubstitution
tail1 = getRemainingPart(exp1)
tail2 = getRemainingPart(exp2)
if initialSubstitution != []:
    tail1 = apply(tail1, initialSubstitution)
```

```
        tail2 = apply(tail2, initialSubstitution)
    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return False
    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution


exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

OUTPUT

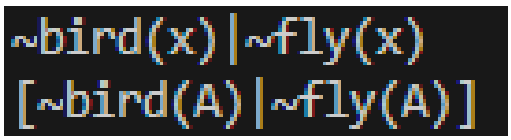


9. FOL to CNF

```
import re
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
     matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
    return statement

def fol_to_cnf(fol):
    statement = fol.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
```

```
    print(statements)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    return Skolemization(statement)


print(fol_to_cnf("bird(x)=>~fly(x)"))
print(fol_to_cnf("∃x[bird(x)=>~fly(x)]"))
```

OUTPUT



10. Forward Reasoning

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
```

```
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in
self.params])})"
        return Fact(f)

class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
```

```
            if val.predicate == fact.predicate:
                for i, v in enumerate(val.getVariables()):
                    if v:
                        constants[v] = fact.getConstants()[i]
                new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None


class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()


    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)


    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
```

```
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1


    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')
kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

OUTPUT