1 Tic Tack Toe Game WEEK-1 15/11/2023

```
board = [' ' for _ in range(10)]
def insertLetter(letter, pos):
    global board
    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]==
  or
 (bo[2]==le and bo[5]==le and bo[8]==
  or
 (bo[3]==le and bo[6]==le and bo[9]==
  or
 (bo[1]==le and bo[5]==le and bo[9]==
  or
 (bo[3]==le and bo[5]==le and bo[7]==
)

def playerMove():
    global board
    run = True
    while run:
        move = input('Please select a position to
                      place an 'X' (1-9):')

        try:
            move = int(move)
            if 1 <= move <= 9:
                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 ValueError:
            print('Please type a number
                   within the range!')
```

~~exce~~

```
def compMove():
    global board
    possibleMoves = [x for x, letter in enumerate
        (board) if letter == ' ' and x != 0]
    for let in ['O', 'X']:
        for i in possibleMoves:
            boardCopy = board[:]
            boardCopy[i] = let
            if isWinner(boardCopy, let):
                return i
    cornersOpen = [i for i in possibleMoves if i in
                   in [1,3,7,9]]
    if cornersOpen:
        return selectRandom(cornersOpen)
    if 5 in possibleMoves:
        return 5
    edgesOpen = [i for i in possibleMoves if i
                 in [2,4,6,8]]
    if edgesOpen:
        return selectRandom(cornersOpen)
    if 5 in possibleMoves:
        return none
    edgesOpen = [i for i in possibleMoves if i
                 in

    do
def selectRandom(li):
    ln = len(li)
    r = random.randrange(ln)
    return li[r]
def isBoardFull(board):
    return board.count(' ') <= 1
```

```
def main():
    global board
    print('Welcome to Tic Tac Toe!')
    printBoard(board)

    while not isBoardFull(board):
        if not isWinner(board, 'O'):
            playerMove()
            printBoard(board)
        else:
            print('Sorry, O's won this time!')
            break
    if not isWinner(board, 'X'):
        move = compMove()
        if move in None:
            print('Tie Game!')
        else:
            insertLetter('O', move)
            print('Computer placed an 'O'
                  in position', move, ':')
            printBoard(board)
    else:
        print('X's won this time! Good Job!')
        break
    if isBoardFull(board):
        print('Tie Game!')
    while True:
        answer = input('Do you want to play
                 again? (Y/N)')
        if answer.lower() == 'y' or answer.lower
                 == 'yes':
```

```
board = [' ' for _ in range(10)]
print('.....')
main()
else
    break;
main()
```

Output:

```
 |   |
 |   |
 |   |
```

Computer place an 'O' in position 3:

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

Please select a position to place an 'X' (1-9): 5

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

Computer placed an 'O' in position 9:

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

Please select a position to place on 'X' (1-9): 7

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

Computer placed an 'O' in position 4:

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

Please select a position to place an 'X' (1-9): 6

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

Tie Game!

```
print(.....)
main()
else
  break;
main()
```

output:

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

computer place an '0' in position 3:

```
X |  | 0
- - - - - -
  |  |
- - - - - -
  |  |
```

Please select a position to place an 'x' (1-9): 5

```
X |  | 0
- - - - - -
  | X |
- - - - - -
  |  |
```

computer placed an '0' in position 9:

```
X |  | 0
- - - - - -
  | X |
- - - - - -
  |  | 0
```

Please select a position to place an 'x' (1-9): 1

```
X |  | 0
- - - - - -
  | X | X
- - - - - -
  |  | 0
```

computer placed an '0' in position 4:

```
X |  | 0
- - - - - -
0 | X | X
- - - - - -
  |  | 0
```

Please select a position to place an 'x' (1-9): 0

```
X | 0 | 0
- - - - - -
0 | X | X
- - - - - -
```

2 Vacuum Cleaner Agent 22/11/2023

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input('Enter Location of Vacuum')
    status_input = input('Enter status of' + location_input)
    status_input_complement = input('Enter status of other room')
    print('Initial Location Condition' + str(goal_state))
    if location_input == 'A':
        print('Vacuum 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 bee cleaned')
        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 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(cost)
        print('Location B is already clean')
else
    print('Vacuum is placed in
                location B')
    if status_input == '1':
        print('Location B is Dirty')
        goal_state['B'] = '0'
        cost+ =1
        print('cost for cleaning' +
                        str(cost))
        print('Location B has been
                        cleaned')
```

```
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(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()
```

Sample Output:

Enter Location of Vacuum A
Enter status of A 0
Enter status of other room 0
Initial Location Condition {'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean
No action 0
Location A is already clean
Goal state:
{'A': '0', 'B': '0'}
Performance Measurement: 0

WEEK-3

3 8-Puzzle Problem using BFS 29/11/2023

```
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('succes')
            return
        pos_moves_to_do = []
        pos_moves_to_do = possible_moves(source,ex
        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(0)
    d = []
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
```

queue =

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

```
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_ca
        in pos_moves_it_can if move_it_ca
    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
```

Sample Output:

```
src = [1,2,3,4,5,6,0,7,8]
Target = [1,2,3,4,5,6,7,8,0]
bfs(src, target)
[1,2,3,4,5,6,0,7,8]
[1,2,3,0,5,6,4,7,8]
[1,2,3,4,5,6,7,0,8]
[0,2,3,1,5,6,4,7,8]
[1,2,3,5,0,6,4,7,8]
[1,2,3,4,0,6,7,5,8]
```

[1,2,3,4,5,6,7,1;
success

Pall 6/12/23

4 8-Puzzle Problem using Iterative Deepening Search WEEK-4

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

```
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
```

Output:

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

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

depth = 1

iddfs(src, target, depth)

False

5 8-Puzzle Problem using Best First Search

```
import heapq
class Node:
    def __init__(self, state, level, heuristic):
        self.state = state
        self.level = level
        self.heuristic = heuristic
    def generate_child(node):
        x,y = find_blank(node.state)
        moves = [(x-1,y), (x+1,y), (x,y-1), (x,y+1)]
        children = []
        for move in moves:
            child_state = move_blank(node.state,
                          (x,y), move)
            if child_state is not None:
                h = calculate_heuristic(child_state
                child_node = Node(child_state,
                             node.level+1, h)
                children.append(child_node)
    return children
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j]==0:
                return i,j
def move_blank(state, src, dest):
    x1,y1 = src
    x2,y2 = dest
    if 0<=x2 and 0<=y2<3:
```

```
le Problem WEEK 3 using Best First Search

eapq

t_(self, state, level, heuristic):
state = state
level = level
heuristic = heuristic
rate_child(node):
= find_blank(node.state)
s = [(x-1,y), (x+1,y), (x,y-1), (x,y+1)]
ren = []

move in moves:
child_state = move_blank(node.state,
                (x,y), move)
if child_state is not None:
    h = calculate_heuristic(child_state)
    child_node = Node(child_state,
                node.level+1, h)
    children.append(child_node)
hildren
_blank(state):
in range(3):
 in range(3):
 state[i][j] == 0:
 return i,j
ank(state, src, dest):
rc
est
x2 and 0 <= y2 < 3:
    new_state = [row[:] for row in state]
    new_state[x1][y1], new_state[x2][y2] =
        new_state[x2][y2], new_state[x1][y1]
    return new_state
else:
    return None
def calculate_heuristic(state):
    goal_state = [[1,2,3], [4,5,6], [7,8,0]]
    h = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j]
                and state[i][j] != 0:
                h += 1
    return h
def best_first_search(initial_state):
    start_node = Node(initial_state, 0, calculate_
                heuristic(initial_state))
    open_list = [start_node]
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.state == [[1,2,3], [4,5,6],
                [7,8,0]]:
            return current_node
        closed_set.add(Tuple(map(Tuple, current_
                node.state)))
        for child in generate_child(current_node):
            if Tuple(map(Tuple, child.state)) not
                in closed_set:
                heapq.heappush(open_list, child)
return None
```

```
initial_state = [[1,2,3],[4,5,6],[7,8,0]]
solution_node = best_first_search(initial_state)
if solution_node:
    print("solution found in", solution_node.depth,
          "moves.")
    print("Path:")
    for row in solution_node.state:
        print(row)
else:
    print("No solution found.")
```

Output:

```
Solution found in 3 moves.
Path:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Time Complexity:

→ BFS using a priority queue for 8-puzzle problem with a simple misplaced tiles heuristic is generally $O(b^d)$, where:

- b is the average no. of paths.
- d is the depth or length of the path from the start to the solution in the maze.

6 8-Puzzle Problem using A* Algorithm

```
import heapq
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, x2, y2):
        if x2>=0 and x2< len(self.data) and
           y2>=0 and y2<len(self.data):
            Temp_puz = self.copy(puz)
            Temp = Temp_puz[x2][y2]
            Temp_puz[x][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 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, start_data, goal_data):
    start = Node(start_data, 0, 0)
    start.fval = self.f(start, goal_data)
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
        print("")
        for i in cur.data:
            for j in i:
                print(j, end = " ")
            print("")
        if self.h(cur.data, goal_data) == 0:
            break
        for i in curr.generate_child():
            i.fval = self.f(i, goal_data)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]
        self.open.sort(key = lambda x:
                       x.fval, reverse = False)
start_state = [['1','2','3'], ['_','4','6'
               ['7','5','8']]
goal_state = [['1','2','3'],['4','5','6'
              ['7','8','_']]
puz = Puzzle(3)
puz.process(start_state, goal_state)
```

Output:

```
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 _
```

Time Complexity

- Time complexity can be exponential in the worst case, usually denoted as $O(b^d)$
- b represents the average branching factor from a given state.
- d is the depth of the solution

7 Create KB using propositional logic and show if the given KB entails query

```
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) and
            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 entail

```
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)
```

((~q v ~p v r) ^ (~q ^ p) ^ q)

first_premise = ('and', ('or', '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_

```
result_second = evaluate_second(second_premise,
                second_conclusion)
if result_first:
    print("For the first input: The knowledge
          base entails the query.")
else:
    print("For the first input: The knowledge
          base doesnot entail the query.")
if result_second:
    print("For the second input: The knowledge
          base entails the query.")
else:
    print("For the second input: The knowledge
          base does not entail the query.")
```

Output:

For the first input: The knowledge base does not entail the query.

For the second input: The knowledge base entails the query.

8 Create a KB using propositional logic and prove the given query using 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
    split_terms('~PvR')
['~P', 'R']
def contradiction(goal, clause):
    contradictions = [f'{goal}v{negate(goal)}',
        f'{negate(goal)}v{goal}']
```

returns 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.
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]},
        return steps
    for clause in clauses:
        if clause not in temp and clause1 =
            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'
goal = 'R'
main(rules, goal)
```

Output:

Step	Clause	Derivation
1.	$R \vee \sim P$	Given
2.	$R \vee \sim Q$	Given
3.	$\sim R \vee P$	Given
4.	$\sim R \vee Q$	Given
5.	$\sim R$	Negated conclusion
6.	Resolved	$R \vee \sim P$ and $\sim R \vee P$ To $R \vee \sim R$, which is in Tum null.

A contradiction is found when $\sim R$ is assumed as True. Hence, R is True.

10/1/24.

9 Implement unification in first order logic

WEEK-8

```
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)==
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 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) != getInitial
                Predicate(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 initialSubstitution1 = []:
    Tail1 = apply(Tail1, initialSubstitut
    Tail2 = apply(Tail2, initialSubstitut
remainingSubstitution = unify(Tail1, Tail2)
if not remainingSubstitution:
    return False
initialSubstitution.extend(remaining
                Substitution)
return initialSubstitution
substitutions = unify(exp1, exp2)
```

print(substitutions)

Output:

exp1 = "knows(A, x)"
exp2 = "knows(y, y)"
Substitutions:
[('A', 'y'), ('y', 'x')]

exp1 = "like(A, y)"
exp2 = "like(K, g(x))"
Substitutions:
Fake

WEEK-11

Convert given first order logic statement into Conjunctive Normal Form (CNF)

```
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~]+\([A-Z,a-z,]+\)'
    return re.findall(expr, string)

def skolemization(statement):
    SKOLEM_CONSTATS = [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_CONSTATS.pop(
    return statement

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

```
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 = statement[:br] + new_statement
    if br>0 else new_statement
return skolemization(statement)
```

Output

```
print(skolemization(fol_to_cnf("∀x food(x)
⇒ likes(John,x)")))
```

~food(A) | likes(John, A)

```
print(skolemization(fol_to_cnf("∀x [∃y [
loves(x,y)]]")))
```

[loves(x, B(x))]

```
print(fol_to_cnf("[american(x) & weapon(y)
& sells(x,y,z) & hostile(z)] ⇒
criminal(x)"))
```

[~american(x) | ~weapon(y) | ~sells(x,y,z) |
~hostile(z)] | criminal(x)

WEEK -10

Create a knowledge base 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(expr
        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 getSubstitute(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.predicat
                    for i, v in enumerate(val.
                        getVariables()):
                        if v:
                            constants[v] = fact.ge
```

```
                constants()[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])

class KB:
    def __init__(self):
        self.facts = self()
        self.implications = set()
    def query(self, e):
        facts = set([f.expression for f in
        i=1                         self.facts])
        print(f'Querying {e}:')
    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'{f}')
kb_ = KB()
kb_.tell('missile(x) => weapon(x)')
kb_.tell('missile(M1)')
kb_.tell('enemy(x, America) => hostile(x)')
```

```
kb_.tell('enemy(Nono, America)')
kb_.tell('owns(Nono, M1)')
kb_.tell('missile(x) & owns(Nono, x) ⇒
          sells(West, x, Nono)')
kb_.tell('american(x) & weapon(y) & sells(x,y,z)
          & hostile(z) ⇒ criminal(x)')
kb_.query('criminal(x)')
kb_.display()
```

Output:

Querying criminal(x):

1. criminal(West)

All facts:

1. sells(West, M1, Nono)
2. criminal(West)
3. hostile(Nono)
4. owns(Nono, M1)
5. enemy(Nono, America)
6. weapon(M1)
7. american(West)
8. missile(M1)

Pass
1/2/24.

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

ARTIFICIAL INTELLIGENCE
(22CS5PCAIN)

Submitted by

SHRAVANI SHEKAR (1BM21CS205)

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
November-2023 to Febrauary-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 Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Shravani Shekar (1BM21CS205) during the 5th Semester November-February 2024.

Signature of the Faculty Incharge:

Dr. Pallavai G B
Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Index Sheet

10	**Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.**	40

WEEK-2

1.Analyse and implement Tic-Tac-Toe game

```
import random

# Initialize the game board
board = [' ' for _ in range(10)]


def insertLetter(letter, pos):
    global board
    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():
    global board
    run = True
    while run:
        move = input('Please select a position to place an \'X\' (1-9): ')
```

```
        try:
            move = int(move)
            if 1 <= move <= 9:
                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 ValueError:
            print('Please type a number!')


def compMove():
    global board
    possibleMoves = [x for x, letter in enumerate(board) if letter == ' ' and x != 0]

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

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

    if 5 in possibleMoves:
        return 5

    edgesOpen = [i for i in possibleMoves if i in [2, 4, 6, 8]]
    if edgesOpen:
        return selectRandom(edgesOpen)

    return None  # Indicates a tie


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


def isBoardFull(board):
    return board.count(' ') <= 1


def main():
```

```
    global board
    print('Welcome to Tic Tac Toe!')
    printBoard(board)

    while not isBoardFull(board):
        if not isWinner(board, 'O'):
            playerMove()
            printBoard(board)
        else:
            print('Sorry, O\'s won this time!')
            break

        if not isWinner(board, 'X'):
            move = compMove()
            if move is None:
                print('Tie Game!')
            else:
                insertLetter('O', move)
                print('Computer placed an \'O\' in position', move, ':')
                printBoard(board)
        else:
            print('X\'s won this time! Good Job!')
            break

    if isBoardFull(board):
        print('Tie Game!')

    while True:
        answer = input('Do you want to play again? (Y/N)')
        if answer.lower() == 'y' or answer.lower() == 'yes':
            board = [' ' for _ in range(10)]
            print('-----------------------------------')
            main()
        else:
            break


# Run the game
main()
```

OUTPUT:

```
-----------
   |   |
-----------
   |   |
Computer placed an 'O' in position 3 :
 X |   | O
-----------
   |   |
-----------
   |   |
Please select a position to place an 'X' (1-9): 5
 X |   | O
-----------
   | X |
-----------
   |   |
Computer placed an 'O' in position 9 :
 X |   | O
-----------
   | X |
-----------
   |   | O
Please select a position to place an 'X' (1-9): 6
 X |   | O
-----------
   | X | X
-----------
   |   | O
Computer placed an 'O' in position 4 :
 X |   | O
-----------
 O | X | X
-----------
   |   | O
Please select a position to place an 'X' (1-9): 8
 X |   | O
-----------
 O | X | X
-----------
   | X | O
Computer placed an 'O' in position 2 :
 X | O | O
-----------
 O | X | X
-----------
   | X | O
Please select a position to place an 'X' (1-9): 7
 X | O | O
-----------
 O | X | X
-----------
 X | X | O
Tie Game!
```

2.Analyse and implement vacuum cleaner agent

```
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
```

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

WEEK-3

3.Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm

```
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 = 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(0)
```

```
    d = []

    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 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 = [1,0,3,4,2,6,7,5,8]
target = [1,2,3,4,5,6,7,8,0]
bfs(src, target)
```

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

```
[10] src=[2,0,3,1,8,4,7,6,5]
     target=[1,2,3,8,0,4,7,6,5]
     bfs(src, target)
```

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

WEEK-4

4.Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be soloved using this algorithm. Implement the same.

```
import numpy as np
import pandas as pd


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]
    elif move == 'd':
        temp[blank + 3], temp[blank] = temp[blank], temp[blank + 3]
    elif move == 'r':
        temp[blank + 1], temp[blank] = temp[blank], temp[blank + 1]
    elif 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
```

OUTPUT:

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

depth = 1
iddfs(src, target, depth)
```

```
False
```

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

depth = 1
iddfs(src, target, depth)
```

```
False
```

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

depth = 1
iddfs(src, target, depth)
```

```
True
```

WEEK-5

5.Analyse best first search and A* algorithm. Implement 8 puzzle problem using both algorithms.

```
import heapq


class Node:
    def __init__(self, state, level, heuristic):
        self.state = state
        self.level = level
        self.heuristic = heuristic

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

def generate_child(node):
    x, y = find_blank(node.state)
    moves = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
    children = []

    for move in moves:
        child_state = move_blank(node.state, (x, y), move)
        if child_state is not None:
            h = calculate_heuristic(child_state)
            child_node = Node(child_state, node.level + 1, h)
            children.append(child_node)

    return children

def find_blank(state):
    for i in range(3):
        for j in range(3):
```

```
            if state[i][j] == 0:
                return i, j

def move_blank(state, src, dest):
    x1, y1 = src
    x2, y2 = dest

    if 0 <= x2 < 3 and 0 <= y2 < 3:
        new_state = [row[:] for row in state]
        new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
        return new_state
    else:
        return None

def calculate_heuristic(state):
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    h = 0

    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j] and state[i][j] != 0:
                h += 1

    return h

def best_first_search(initial_state):
    start_node = Node(initial_state, 0, calculate_heuristic(initial_state))
    open_list = [start_node]
    closed_set = set()
```

```
    while open_list:
        current_node = heapq.heappop(open_list)

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

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

        for child in generate_child(current_node):
            if tuple(map(tuple, child.state)) not in closed_set:
                heapq.heappush(open_list, child)

    return None

initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
solution_node = best_first_search(initial_state)

if solution_node:
    print("Solution found in", solution_node.level, "moves.")
    print("Path:")
    for row in solution_node.state:
        print(row)
else:
    print("No solution found.")
```

OUTPUT:

```
Solution found in 3 moves.
Path:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

A* Algorithm:

```
import heapq
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 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data):
            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 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, start_data, goal_data):
        start = Node(start_data, 0, 0)
        start.fval = self.f(start, goal_data)
        self.open.append(start)
        print("\n\n")

        while True:
            cur = self.open[0]
            print("")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")

            if self.h(cur.data, goal_data) == 0:
                break

            for i in cur.generate_child():
                i.fval = self.f(i, goal_data)
                self.open.append(i)
                self.closed.append(cur)

            del self.open[0]
            self.open.sort(key=lambda x: x.fval, reverse=False)

start_state = [['1', '2', '3'], ['_', '4', '6'], ['7', '5', '8']]
goal_state = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', '_']]
```

puz = Puzzle(3)

puz.process(start_state, goal_state)

OUTPUT:

```
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 _
```

WEEK-6

6.Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.

```
def evaluate_first(premise, conclusion):
    # Create all possible models for the variables p, q, and r
    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}
    ]

    # Check if the premise logically entails the conclusion
    entails = True  # Initially assume the premise entails the conclusion
    for model in models:
        if evaluate_expression(premise, model) != evaluate_expression(conclusion, model):
            entails = False  # If any model disagrees, premise does not entail conclusion
            break

    # Return the result
    return entails

# Define a function to evaluate logical expressions for the second input
def evaluate_second(premise, conclusion):
    # Generate all possible truth assignments to p, q, and r
    models = [
        {'p': p, 'q': q, 'r': r}
```

```
        for p in [True, False]
        for q in [True, False]
        for r in [True, False]
    ]

    # Check if the conclusion holds in every model where the premise is true
    entails = all(evaluate_expression(premise, model) for model in models if
evaluate_expression(conclusion, model) and evaluate_expression(premise, model))

    # Return the result
    return entails

# Define a function to evaluate logical expressions based on the given expression and model
def evaluate_expression(expression, model):
    # Evaluate the logical expression recursively using the given model
    if isinstance(expression, str):
        # Base case: if it's a single variable or literal
        return model.get(expression)
    elif isinstance(expression, tuple):
        op = expression[0]  # Get the operator
        if op == 'not':
            return not evaluate_expression(expression[1], model)  # Negation
        elif op == 'and':
            return evaluate_expression(expression[1], model) and
evaluate_expression(expression[2], model)  # Conjunction
        elif op == 'if':
            return (not evaluate_expression(expression[1], model)) or
evaluate_expression(expression[2], model)  # Implication
        elif op == 'or':
            return evaluate_expression(expression[1], model) or
evaluate_expression(expression[2], model)  # Disjunction (OR)
```

```
# Premise and conclusion for the first and second inputs
first_premise = ('and', ('or', 'p', 'q'), ('or', ('not', 'r'), 'p'))  # (p OR q) AND (NOT r OR p)
first_conclusion = ('and', 'p', 'r')  # p AND r

second_premise = ('and', ('or', ('not', 'q'), ('not', 'p'), 'r'), ('and', ('not', 'q'), 'p'), 'q')
second_conclusion = 'r'

# Evaluate both inputs using the merged function
result_first = evaluate_first(first_premise, first_conclusion)
result_second = evaluate_second(second_premise, second_conclusion)

# Print the results for both inputs
if result_first:
    print("For the first input: The knowledge base entails the query.")
else:
    print("For the first input: The knowledge base does not entail the query.")

if result_second:
    print("For the second input: The knowledge base entails the query.")
else:
    print("For the second input: The knowledge base does not entail the query.")
```

OUTPUT:

```
>>>
============ RESTART: C:/Users/Admin/Desktop/1BM21CS205 AI LAB/p5.py ===========
For the first input: The knowledge base does not entail the query.
For the second input: The knowledge base entails the query.
>>>
```

WEEK-7

7.Create a knowledgebase using prepositional logic and prove the given query using 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]}']
```

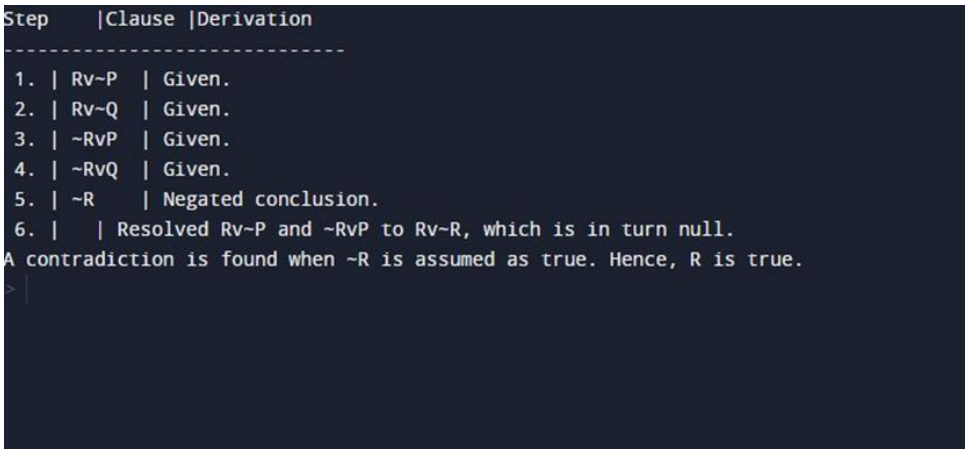
```
                else:
                    if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                        temp.append(f'{terms1[0]}v{terms2[0]}')
                        steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                        \nA contradiction 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
```

OUTPUT

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 ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)

main(rules, 'R')

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

WEEK-8

8.Implement 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(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
```

OUTPUT

exp1 = "knows(A,x)"

exp2= "knows(y,Y)"

exp1 = "like(A,y)"

exp2 = "like(K,g(x))"

WEEK-9

9.Convert given first order logic statement into Conjunctive Normal Form (CNF).

```
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
import re

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

OUTPUT:

WEEK-10

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):
        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}')
```

OUTPUT