VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (22CS5PCAIN)

Submitted by

Varun S (1BM21CS238)

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 **Varun S (1BM21CS238),** 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 June-2023 to Sep-2023. 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.

Sandhya A Kulkarni
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Course Outcome

CO1	Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO2	Analyse Searching and Inferencing Techniques.
CO3	Design a reasoning system for a given requirement.
CO4	Conduct practical experiments for demonstrating agents, searching and inferencing.

INDEX

Name VARUN.S Sub. IBM21CS238

Std.: V sem Div. 'D' Roll No.

Telephone No. E-mail ID.

Blood Group. Birth Day.

Sr.No.	Title	~~Page No.~~ Date	Sign./Remarks
1.	8-puzzle	6/12	
2.	Tic-Tac-Toe	6/12	
3.	Vacuum Cleaner Agent	6/12	6/12
4.	8-puzzle using iterative depth first search	13/12	13/12
5.	8-puzzle using Best-first search and A* algorithm		
6.	Create knowledgebase using propositional logic and prove query entails knowledge base or not.	20/12	20/12
7.	Create knowledgebase using propositional logic and prove using query	27/12	27/12
8.	Implement unification in first order logic	10/01/2024	10/1/24
9.	Convert first Order Logic to CNF	17/01/2024	17/1/24
10.	Create a knowledgebase consisting of FOL statements and prove query using forward reasoning	24/01/2024	

PAGE NO :
DATE :

8-puzzle.

```
import heapq

class PuzzleState
    def __init__ (self, board, goal, moves=0):
        self.board = board
        self.goal = goal
        self.moves = moves
        self.priority = self.calculate_priority()

    def __it__ (self.other):
        return self.priority < other.priority

    def calculate_priority(self):
        return self.moves + self.manhattan distance

    def manhattan_distance(self)
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                if value != 0:
                    goal_row, goal_col =
                        divmode(self.goal.index[10],3)
                    distance += abs(i-goal_row)
                    abs(j-goal_col)
                    return dis
                return distance
```

```
def get_possible_moves(state):
    moves = []
    row, col = next((i,j) for i in range(3)
                          for j in range(3))
    if (state.board[i][j] == 0):
        fo
        for i,j in ((row-1, col), (row+1, col),
                    (row, col-1), (row, col+1)):
            if (0<= i <=3 and 0<= j <=3):
                new_board[row][col]
                m
                moves.append(puzzlestate(new_board[i]
                    = new_board[i][j])
    return moves

def solve_puzzle(initial, goal):
    initial_state = PuzzleState(initial, goal)
    heap = [initial_state]
    visited = set()
    while (heap):
        current_state = heapq.heappop(heap)
        if (current_state.board == goal):
            return current_state.moves
        visited.add(tuple(map(tuple,
                    current_state.board)))
        for m in get_possible_moves(restate)
            if tuple(map(tuple, moveboard)
                heap.heappush(heap, move)
            retu
    return 1
```

Output:

src: 2,0,3,1,8,4,7,6,5

target: 1,2,3,8,0,4,7,6,5

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

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

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

PAGE NO :
DATE :

Lab Program - 2
Tic-Tac-Toe

```
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-"*9)

def check_winner(board,players):
    for i in range(3):
        if all(board[i][j]==players for j in range
        or
        all(board[i][j]==players for j in range(3)
        return True
        if all(board[i][j]==players for ji in range(3)
        or
        all(board[i][2-i]==player for i in range(3)
        return True
    return False

def get_move():
    while True:
        try
            move=int(input("Enter your move(1-9):
            if 1<=move<=9:
                return move
            else
                print("invalid move")
        except ValueError
            print("Invalid")
```

```
def Tic-Tac-toe:
    board [[" " for _ in range(3)]
           for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        move = get_move()
        row, col = (move-1)//3, (move-1)%3
        if board[row][col] == " ":
            board[row][col] == current_player
        else:
            if check_winner(board, current_player):
                print_board(board)
                print("Tie")
            else
                print("Invalid move"):
```

Output

```
enter your move (1-9)
X |   |       X |   |       X |   |
  |   |       O | X |       O | X |
  |   |         |   |       O |   | X
                              player 1 wins

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

PAGE NO :
DATE :

Vacuum Cleaner agent

```
def vacuum_world():
    goal_state = {'A':'0', 'B':'0'}
    cost = 0
    location_input = input("Enter location")
    status_input = input("Enter status of" + location_input)
    status_input_complement = input("Enter status of other room")

    if location_input == 'A':
        print("Vacuum 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 cleaned.")

        if status_input_complement == '1':
            print("Loc B is dirty")
            print("Moving right to loc 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");
```

```
if status_input == '0':
    print("Loc A is already cleaned")
    if status_input_complement == '1':
        print("Loc B is Dirty")
        print("Moving right to location B")
        cost += 1
        goal_state['B'] = '0'
        cost += 1
        print("cost for suck " + str(cost))
        print("Loc B has been cleaned")
    else
        print("No action" + str(cost))
        print(cost)
        print("Location B is already cleaned")
else
    print("Vacuum is placed in loc B")
    if status_input == '1'
        print("Location B is dirty")
        goal_state['B'] = '0'
        cost += 1
        print("cost for cleaning" + str(cost))
        print("Loc B has been cleaned")
        is
        if status_input_complement == '1'
            print("Loc A is dirty")
            print("Moving Left to LOCA")
            cost += 1
            print("cost for moving LEFT" +
                  str(cost))
            print("Loc A has been cleaned")
```

```
else:
    print(cost)
    print("Loc B is already cleaned")
    if status_input-complement=='1':
        print("Location A is dirty")
        print("Moving left to Loc A")
        cost+=1
        print("cost for moving left"+str(cost))

        goal_state['A']='0'
        cost+=1
        print("cost="+str(cost)+" Loc A 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 loc of vacuum A
Enter state of A: 1
Enter state of other room: 0
Initial {'A':'0','B':'1'}
Loc A is dirty
cost for cleaning A: 1
Loc B is already clean
Goal state:
{'A':'0','B':'1'}
performance measurement=1
```

Week-4

8 puzzle problem with iterative deepening Search

```
def iterative_deepening_search(src, target):
    depth_limit = 0;
    while True:
        result = depth_limited_search(src, target, depth_limit)
        if result not None:
            print("success");
            return
        depth_limit += 1
        if depth_limit > 30:
            print("Solution not found within limit")
            return

def depth_limited_search(src, target, depth_limit, visited_states):
    if src == target
        print_state(src)
        return src
    if depth_limit == 0;
        return None
    visited_states.append(src)
    pos
    poss_moves_to_do = possible_moves(src, visited_states)

    for moves in poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
            result = depth_limited_search
                (move, target, depth_limit -1,
                visited_state)
```

```
            if result is not None:
                return result
        return None

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]
    elif m == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    elif m == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    return temp
```

```
def print_state(state):
    print(f"{state[0]} {state[1]} {state[2]}\n"
          f"{state[3]} {state[4]} {state[5]}\n"
          f"{state[6]} {state[7]} {state[8]}\n")
```

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

iterative_deepening_search(src, target)
```

Output:

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

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

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

Success
Solution found
```

PAGE NO :
DATE :

Week-5

8 puzzle using best first search.

```
input queue as Q
import goal = [[1,2,3], [4,5,6], [7,8,0]]
def isgoal(state):
    return state == goal.

def heuristicvalue(state):
    cnt = 0
    for i in range(len(goal)):
        for j in range(len(goal[i])):
            if goal[i][j] != state[i][j]:
                cnt += 1
    return cnt

def getCoordinates(currentState):
    for i in range(len(goal)):
        for j in range(len(goal[i])):
            if currentState[i][j] == 0:
                return (i, j)

def isValid(i, j) -> bool:
    return 0<= i <3 and 0<= j <3

def BFS(state, goal) -> int:
    visited = set()
    pq = Q.priorityqueue()
    pq.put([Heuristicvalue(state), 0, state])
    while not pq.empty():
        moves, current_state = pq.get()
```

```
if currentState == goal:
    return moves
if tuple(map(tuple, currentstate)):
    i, j = coordinates[0], coordinates[1]
return moves.

if tuple(map(tuple, currentstate)) in visited:
    continue.

for dx, dy in [[0,1],(0,1),(1,0),[-1,0]]:
    new-i, new-j = i+dx, j+dy
    if isvalid(new-i, new-j):
        new-state [row[i] for row in
                   currentstate)
    new_state[i][j], new_state[new-i][new-j]
                =
    new state[new-i][new-j], new-state[i][j]

State = [1,2,3], [4,5,6], [7,8]
State = [1,2,3], [4,5,6], [7,0,8]
moves = BFS (State, goal)
if moves == -1:
    print("No way to reach given state")
else
    print("Reached in " + str(moves)+"moves")
```

Output:

Reached in 1 moves.

PAGE NO :
DATE :

A* algorithm

```
import queue as Q
goal = [[1,2,3], [4,5,6], [7,8,0]]
def isgoal(state):
    return state == goal

def Heuristic Value(state):
    cnt=0
    for i in range((len(goal)):
        for j in range (len(goal[i])):
            if goal[i][j] != state[i][j]
                cnt += 1
    return cnt

def getCoordinates(currentstate):
    for i in range (len(goal)):
        for j in range (len(goal[i])):
            if currentstate[i][j] == 0:
                return (i,j)
def isValid(i,j) → bool:
    return 0 <= i<3 and 0 <= j<3

def A_star(state, goal) → int:
    visited = set()
    pq = Q.priorityQueue()
    pq.print((Heuristic value (state), 0, state))

while not pq.empty():
    _, moves, currentstate = pq.get()
        if currentstate == goal
            return moves
```

M4 PRO

```
for dx, dy in [(0,1) (0,-1) (1,0) (-1,0)]
    new-i, new-j = i+dx, j+dy
    if isValid (new-i, new-j):
        new_state = [row[i] for row in
                     currentstate]

new-state[i][j], new-state[new-i][new-j]
= new-state[new-i][new-j], new-state[i][j]

if tuple(map(tup, new_state)) not in visited:
    pq.print((Heuristicvalues(new-state)
              + moves, moves+1,
              new-state))

    if new-state == goal:
        print(new-state)
return -1

states = [[1,2,3], [4,0,5], [6,7,8]]
moves = A-star (state, goal)
    if moves == -1;
        print("No way to reach")
    else:
        print("Reach in "+ str(moves)+ "moves")
```

Output:

Reach in 1 moves

13/12/23

20/12/2023

Week-6

Code: Create knowledgebase using propositional logic and show query entails knowledge base

```
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 = True
            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,mod
    for model in models if
    if evaluate_expression (conclusion, model)
    and evaluate_expression (premise, model)
    return entails

def evaluate_expression (expression, mode)):
    if is instance (expression, str):
        return model.get (expression, str)
    elif is instance (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[2], 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', 'p', 'q'),
                        ('or', ('not', 'r'), 'p'))

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

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
          knowledge base entails query")

else
    print("For first input: Knowledge base
          does not entails query")
```

Output:

For first input: Knowledge base does not entail query

For second input: Knowledge base entails query.

PAGE NO :
DATE :

27-12-2023

Week-7

Create knowledgebase using propositional logic and prove the given query using propositional logic

```
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}']
```

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]}')
                        :
```

Continued.

```
        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{terms[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
    j = (j+1) % n
i += 1
return steps
```

```
rules = 'RV~P RV~Q ~RVP ~RV~Q'
goal = 'R'
main(rules, goal)
rules = 'PVQ&PV~R~QVR'
goal = 'R'
main(rules, goal)
rules = 'PVQ PV~R~PVR RVS RV~Q~SV~Q'
main(rules, 'R')
```

Output:

Step	Clause	Derivation
1.	R∨¬P	Given.
2.	R∨¬Q	Given.
3.	¬R∨P	Given.
4.	¬R∨Q	Given.
5.	¬R	Negated conclusion
6.		Resolved R∨¬P and R∨¬R to R∨¬R, which is in turn null

A contradiction is found when ¬R is assumed as true. Hence, R is true.

Step	Clause	Derivation
1.	P∨Q	Given.
2.	¬P∨R	Given.
3.	¬Q∨R	Given.
4.	¬R	Negated conclusion
5.	Q∨R	Res. from P∨Q and ¬P∨R
6.	P∨R	Res. from P∨Q and ¬Q∨R
7.	¬P	Res. from ¬P∨R and ¬R
8.	¬Q	Res. from ¬Q∨R and ¬R
9.	Q	Res. from ¬R and Q∨R
10.	P	Res. from ¬R and P∨R
11.	R	Res. from Q∨R and ¬Q
12.		Res. from R and ¬R to R∨¬R, which is in turn null

A contradiction is found when ¬R is assumed as true, Hence R is true.

PAGE NO:
DATE: 10/01/2024

Week-8

Implement unification in first order logic.

Code:

```
def unify(expr1, expr2):
    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    if func1 != func2:
        print("Expressions cannot be unified
              different functions")
        return None

    args1 = args1.rstrip(')').split(',')
    args2 = args2.rstrip(')').split(',')
    substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower and a1!=a2
            substitution[a1] = a2
        elif
            substitution [a1] = a2
        elif
            substitution [a2] = a1
        elif
            print("Incompatible arguments")
            return None
    return substitution

def apply_substitution (expr, substitution):
    for key, value in substitution.items()
        expr = expr.replace (key, value)
    return expr
```

```
if __name__ == "__main__":
    expr1 = input("Enter first expression")
    expr2 = input("Enter second expression")

    substitution = unify(expr1, expr2)

    if substitution:
        print("The substitutions are:")
        for key, value in substitution.items():
            print(f'{key} / {value}')

        expr1_result = apply_substitution(expr1, substitution)
        expr2_result = apply_substitution(expr2, substitution)

        print(f'unified expression 1: {expr1_result}')
        print(f'unified expression 2: {expr2_result}')
```

Output:

```
Enter the first expression : knows(John, x)
Enter the second expression : knows(y, mother(y))
The substitutions are:
    y / John
    x / mother(y)
Unified expression 1 : knows(John, mother(y))
Unified expression 2 : knows(John, mother(John))
```

Scanned with CamScanner

PAGE NO :
DATE :

Week-9

First Order Logic to CNF:

Code:

```
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-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)
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)]
```

8/12/24

Week-10

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

#Code:

```
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-zA-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 [v if 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 =~~
    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.fact.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.tell('king(sushanth)')
kb.tell('greedy(sushanth)')
kb.query('evil(x)')
kb.display()
```

P.T.O.

Output:

Querying evil(x):

1. evil (sushanth)
2. evil (John)

All facts:

1. evil (sushanth)
2. greedy(sushanth)
3. King (John)
4. evil (John)
5. king (Richard)
6. king (sushanth)
7. greedy (John

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:
```

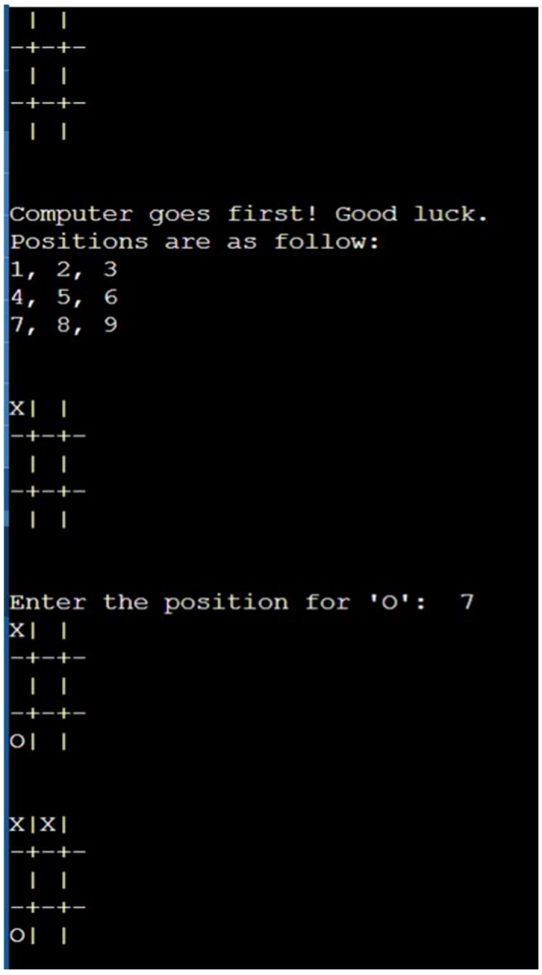
```
print("Can't insert there!")
position = int(input("Please enter new position: "))
insertLetter(letter, position)
return
def checkForWin():
if (board[1] == board[2] and board[1] == board[3] and board[1] != ' '):
return True
elif (board[4] == board[5] and board[4] == board[6] and board[4] != ' '):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

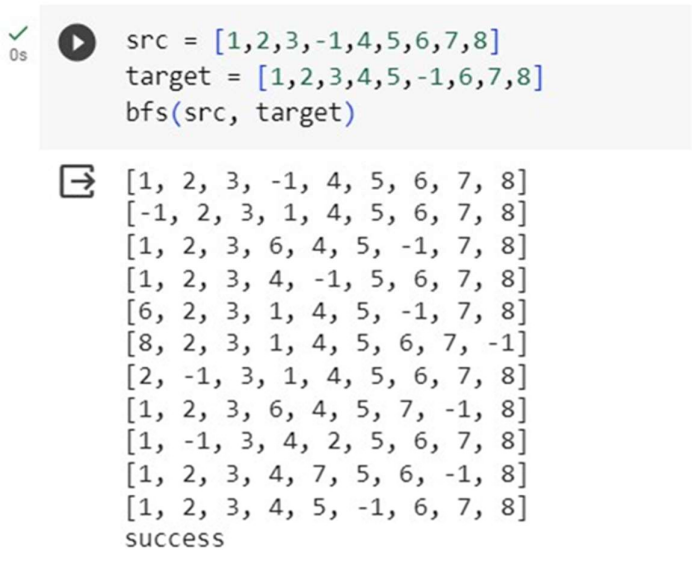
2. 8 Puzzle
(bfs)

```
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



```
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, -1, 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
```

```
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

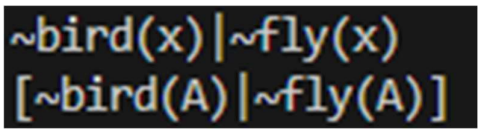
```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

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