AI_LAB
5D

INDEX

1BM21CS222

Name Subramanya.L Std Sec

Roll No. Subject School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	6/12/23	Tic-Tac-Toe		10/10
2.	6/12/23	Vacuum Cleaner		10/10
3.	6/12/23	8-Puzzle game		10/10
4.	6/12/23	Iterative deepning		10/10
5.	20/12/23	Breadth-first search		10/10
6.	20/12/23	A* algorithm to solve 8-puzzle		
7.	27/12/23	Show query entails knowledge or not.		
8.	27/12/23	Create knowledge base using propositional logic & prove query resolution.		
9.	10/1/24	Implement unification using 1st order logic		10/10
10.	17/1/24	FOL to CNF		10
11	24/1/24	Query FOL		10

Date ___/___/___
Page _____

TIC TACK TOE GAME

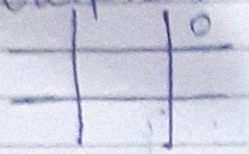
```
import random

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(' ' + board[4] + ' | ' + board[5] + ' | ' +
          board[6])
    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[1]==le and bo[5]==le and bo[9]==le
    (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 'X' ((1-9):")
        try:
            move = int(move)
            if 1<= move <=9:
                if spaceIsFree(move):
                    run = False
                    insertLetter('X', move)
                else:
                    print("Sorry")
            else:
                print("Please type a no.
                        within the range!")
        except ValueError:
            print("Please type a no:")

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

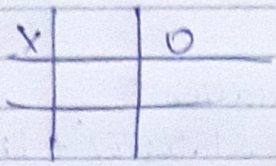
```
    for let in ['O', 'X']:
        for i in possibleMoves:
            boardCopy = board[:]
            boardCopy[i] = let
            if isWinner(boardCopy, let):
                return i

    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 in possibleMoves if
                 in [2,4,6,8]]
    if edgesOpen:
        return selectRandom(edgesOpen)
    return None

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");
            break

    if not isWinner(board, 'X'):
        move = compMove()
        if move == None:
            print("Tie")
        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")

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()
```

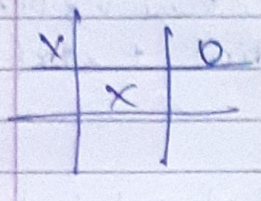
Output:



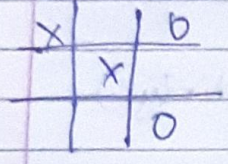
Computer placed an 'O' in position 3:



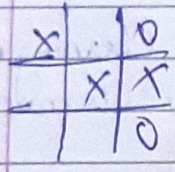
Please select a position to place an 'X' (1-9): 5



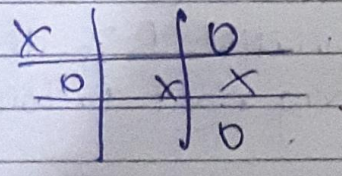
Computer placed an 'O' placed in 9:



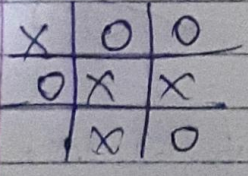
Please select a position to place an 'X' ((1-9) : 6



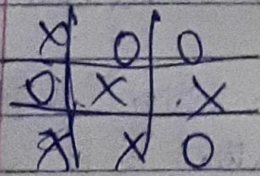
Computer placed an 'O' in position 4:



Please select a position to place an 'X' : 8



Comp Placed at position at 2



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

Tie Game!

VACCUM CLEANER AGENT

```
def vaccum-world():
    goal = {'A':'0', 'B':'0'}
    cost = 0

    Li = input("Enter location")
    Si = input("Enter status of" + Li)
    Sic = input("Enter status of other room")
    print("Initial Condition" + str(goal))

    if Li == 'A':
        print("Vacuum is placed in location A")
        if Si == '1':
            print("A is dirty")
            goal['A'] = '0'
            cost += 1
            print("Cost of Cleaning A" + str(cost))
        print("Location A is Cleaned")

    if Sic == '1':
        print("Location B is dirty")
        print("Moving right to location B")
        cost += 1
        print("Cost for moving right" + str(cost))
        goal['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 s1 == '0':
    print("Location A is already clean")
    if s1c == '1':
        print("Location B is Dirty")
        print("Location moving right
              to location B")
        cost += 1
        print("COST for moving Right" + str(cost))
        print(cost)
        print("location B is already clean")
else:
    print("vacuum is placed in location B")
    if s1 == '1':
    print("Location B is dirty")
        gs['B'] = '0'
        cost += 1
    print("Location B has been cleaned")

if s1c == '1':
    print("location A is dirty")
    print("Moving left to location A")
    cost += 1
    print("Cost for moving left" + str(cost))
    gs['A'] = '0'
    cost += 1
    print("COST for SUCK" + str(cost))
    print("Location A has been cleaned")
else:
    print(cost)
    print("location B is already cleaned")
```

```
if sic == '1':
    print("Location A is dirty")
    print("Moving LEFT to location A")
    cost += 1
    print("Location B is already
                clean")

if sic == '1':
    print("Location A is dirty")
    print("Moving left to the location A")
    cost += 1
    print("cost for moving left" + str(cost))
    gs['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(gs)
print("Performance Measurement:" +
                str(cost))

vacuum_world()
```

Output:

Enter location of vacuum: A
Enter status of A = 1
Enter status of other room 1
Initial location condition {'A': '0', 'B': '0'}
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 location B
Cost for moving right 2
Cost for Suck 3
location B has been cleaned
Goal State:
{'A': '0', 'B': '0'}
Performance measurement: 3

Date __/__/__
Page _____

8- PUZZLE PROBLEM USING BFS

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

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

    exp = []

    while len(q) > 0:
        source = q.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 q:
            q.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 = [1,2,3,4,5,6,0,7,8]
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]
[1,2,3,4,5,6,7,8,0]

Success.

6 | 12 | 23

Date / /
Page

Iterative deeping search.

```
def bfs(src, target)
    q=[]
    q.append(src)

    exp=[]

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

        print(source)

        if source==target:
            print("Success")
            return
        poss_moves_to_do=list(possible_moves(source,exp))

        for move in poss_moves_to_do:

            if move not in exp and move not in queue:
                q.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
                if move_it_can not in visited]

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

    return temp

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

Best First Search:

objective = BFS also explores graph or tree, but it selects the most promising node based on heuristic func^c, not necessarily following strict level by level approach.

priority Queue = BFS uses priority queues to prioritize nodes based on their heuristic value.

Completeness = Not complete, because of infinite loop or heuristic func^c poorly designed.

Solving 8 puzzle problem using (BFS)

```
import queue as Q
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(current state):
      for i in range(len(goal)):
          for j in range( len(goal[i])):
              if current state[i][j] == 0:
                  return (i, j)
def invalid(i, j) -> bool:
    return 0 <= i < 3 and 0 <= j < 3
```

```
def BFS(state, goal) -> int:
    visited = set()
    pq = Q.priorityqueue()
    pq.put( (Heuristic Value (state), 0, state))
    while not pq.empty():
        _, moves, current state = pq.get()
        if Current.state == goal:
            return moves
        if tuple ( map( tuple, currentstate):
            Coordinates = get coordinates (Current state)
            i, j = coordinate [0], coordinates[1]
    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[:] for row in current]
            newstate[i][j], new_state [new_i][new_j]:
                newstate[new_i][new_j], new_state[i][j]
            if tuple(map(tuple, new_state)) not in visited
                pq.put ((Heuristic values (new_state), moves+1,
                                                    new state))
    return -1
State = [[1,4,3], [4,5,6], [7,0,8]]
moves = BFS (state, goal)
if moves == -1:
    Print ("No way")
else:
    print ("Reached in" + str(moves) + ")
```

O/P : Reached in 1 moves.

algo to solve 8 puzzle problem.

```
import queues as Q
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[0])):
            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.put((HeuristicValue(state), 0, state))

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

Prog 3

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

```
def evaluate_expression(q, p, r):
    expression_result = ((not q or not p or r)
                         and (not q and p) and r)
    return express_result

def generate_truth_table():
    print("q | p | r | expression (KB) | Query(s)")
    print("--- | --- | --- | ------- | -----")
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(q,
    query_result = r
    print(f"{q} | {p} | {r} | {expression_result}")

def query_entails_knowledge():
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(q, p, r)
                query_result = r
                if expression_result and not query_result:
                    return False
    return True
```

```
def main():
    generate_truth_table()
    if query_entails_knowledge():
        print("\n Query entails the knowledge")
    else:
        print("\n Query does not entails
                the knowledge")

if __name__ == "__main__":
    main()
```

O/p :

```
Enter rule : (p v q) ^ (~r v p)
Enter the Query : p ^ r
***** Truth Table Reference *****
kb  alpha
True  True
.
True  False
```

The knowledge Base does not entail query.

Q. Create a knowledge base using propositional logic and prove the given query using resolution.

Code:

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

P(a, f(x))

```
def contradiction(goal, clauses):
    contradictions = [f'{goal} v {negate(goal)}',
                      f'{negate(goal)} v {goal}']
    return clauses in contradictions or reversed(clauses)
        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]}'
        return steps
    elif len(gen) == 1:
        clauses += [f'{gen[0]}']
    else:
        if contradiction(goal, f'{temp1[0]}v{temp2[0]}'):
            temp.append(f'{temp1[0]}v{temp2[0]}')
            steps[' '] = f'Resolved {temp[i]} and {temp[j]}
                                        to {temp[-1]}'
            return steps
        elif len(gen) == 1:
            clauses += [f'{gen[0]}']

    for clause in clauses:
        if clause not in temp and clause[::-1] 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 = i+1
return steps

rules = 'RV~P R~QS ~RVP ~RVQ'
goal = 'R'
main(rules, goal)
rules = 'PVQ ~PVR ~QVR'
Goal = 'R'
rules = 'PVQ PVR ~PVR RVS RV~Q ~SVQ'
main (rules, 'R')
```

O/P:

R v ~P R v ~Q ~P v P ~R v Q

Enter the query: R

Step	clause	Derivation
1.	R v ~P	Given
2.	R v ~Q	Given
3.	~R v P	Given
4.	~P v Q	Given
5.	~R	Negated Conclusion
6.		Resolved R v ~P and ~R v P to R v ~R, which is in turn null.

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

27/12/23

Q. Implement unification in first order logic

```
def unify(exp1, exp2):
    func1, args1 = exp1.split('(', 1)
    func2, args2 = exp2.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 a1.islower() and not a2.islower():
            substitution[a1] = a2
        elif not a1.islower() and a2.islower():
            substitution[a2] = a1
        elif a1 != a2:
            print("Expressions cannot be unified. Incompatible arguments.")
            return None
    return substitution
```

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

if __name__ == "__main__":
    exp1 = input("Enter 1st exp")
    exp2 = input("Enter 2nd exp")

    substitution = unify(exp1, exp2)

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

        exp1_result = apply_substitution(exp1, substitution)
        exp2_result = apply_substitution(exp2, substitution)

        print(f"Unified exp1 is {exp1_result}")
        print(f"Unified exp2 is {exp2_result}")
```

O/P =

Enter 1st expression: p(x, f(y))
Enter 2nd expression: p(a, f(g(z)))

The substitutions are:
x with a
f(y) with f(g(z))

Unified expression 1 = p(a, f(g(z)))
Unified expression 2 = p(a, f(g(z)))

10/1/24

First order logic to CNF conversion.

```
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())
    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 statement:
    statement = statement.replace(s, fol
while '-' in statement:
    i = statement.find('-')
    br = statement.rfind('[', 0, i) if '[' in
         statement else 0
    new_statement = '~' + statement[br:i] +
                    '|' + statement[i+1:]
    statement = statement[:br] + new_stat
                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)"))
```

o/p = ~bird(x) | ~fly(x)

11/1/24

Query FOL

```
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~]+)\([A-Z,]+\)'
    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].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.getVa
                        if v:
                            constants[v] = fact.ge
                    new_lhs.append(fact)
```

```
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key,
                                constants[key])
        exp = f'{predicate}({attributes})'
        return Fact(exp) 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.expres
        for f in self.facts])):
        print(f"{i}: {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)')
```

O/P = Querying evil(x):
1. evil(John)

Smita
23/4/24