VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

SWASTIK SRIVASTAVA (1BM21CS228)

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 **SWASTIK SRIVASTAVA (1BM21CS228),** 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.

SURYA Gold
Date 22/11/23 Page

Implement tic tac toe

```
import math
import copy

X = 'X'
O = 'O'
EMPTY = None
def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO = 0;
    countX = 0;
    for y in [0,1,2]:
        for x in board[y]:
            if x == "O":
                countO = countO+1;
            elif x == "X":
                countX = countX+1;
    if countO >= countX:
        return X
    elif countX > countO:
        return O

def actions(board):
    freeboxes = set();
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board[i][j] == EMPTY;
```

```
                freeboxes.add((i,j))
        return freeboxes

def result(board, action):
    i = action[0];
    j = action[1];
    if type(action) == list:
        action = (i,j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X;
        elif player(board) == O:
            board[i][j] = O
    return board

def winner(board):
    if (board[0][0] == board[0][1] == board[0][2] == X
or board[1][0] == board[1][1] == board[1][2] == X
or board[2][0] == board[2][1] == board[2][2] == X):
        return X;

    if (board[0][0] == board[0][1] == board[0][2] == O
or board[1][0] == board[1][1] == board[1][2] == O or
board[2][0] == board[2][1] == board[2][2] == O);
    return O;

for i in [0,1,2]:
    s2 = []
    for j in [0,1,2]:
        s2.append(board[j][i])
    if (s2[0] == s2[1] == s2[2]):
        return s2[0]
strike0 = []
```

```
for i in [0,1,2]:
    strikeD.append(board[i][i])
if (strikeD[0] == strikeD[1] == strikeD[2]):
    return strikeD[0]
if (board[0][2] == board[1][1] == board[2][0]):
    return board[0][2]
return None

def terminal(board):
    full = True
    for i in [0,1,2]:
        for j in board[i]:
            if j is None:
                full = False
    if full:
        return True
    if (winner(board) is not None):
        return True
    return False

def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:
        return -1
    else:
        return 0

def minmax_helper(board):
    isMaxTurn = True if Player(board) == X else
                False
    if terminal(board):
        return utility(board):
```

SURYA Gold

Date _____ Page _____

```
scores = []
for move in actions(board):
    result(board, move)

scores.append(minimax_helper(board))
    board[move[0]][move[1]] = EMPTY
return max(scores) if isMaxTurn else
min(scores)


def minimax(board):
    isMaxTurn = True if player(board) == X else
    False
    bestmove = None,
    if isMaxturn:
        bestscore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestscore):
                bestscore = score
                bestmove = move
        return bestmove
    else:
        bestscore = +math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score < bestscore):
                bestscore = score
                bestmove = move
        return bestmove
```

```
def print_board(board):
    for row in board:
        print(row);

game_board = initial_state()
print("Initial Board:")
print_board(game_board)

while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("\nEnter your move
                           (row, column): ")
        row, col = map(int, user_input.split(','))
        result(game_board, (row, col))
    else:
        print("\nAI is making a move...")
        move = minimax(copy.deepcopy(game_board))
        result(game_board, move)

print("\nCurrent Board:")
print_board(game_board)

if winner(game_board) is not None:
    print(f"\nThe winner is: {winner(game_board)
          }")
else:
    print("\nIt's a tie!")
```

OUTPUT

enter your move

X		

enter your move
AI making move

X		
O		

enter your move

X		
O	X	

AI move

X		
O	X	
	O	

enter your move

X		
O	X	
	O	X

Player wins

SURYA Gold
Date 22/11/23 Page

Vaccum cleaner Agent

```
def vaccum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter location of vaccum")
    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("vaccum is placed in location A")
        if status_input == '1':
            print("location A is Dirty")
            goal_state['A'] = '0'
            cost += 1
            print("cost for cleaning A" + str(cost))
            print("location A has been 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")
```

else:

if status_input == '0':
    print("Location A is already clean")

if status_input_complement == '1':
    print("Location B is dirty.")
    print("moving right to location B.")
    cost += 1;
    print("cost for moving right" + str(cost))
    goal_state['B'] = '0'
    cost += 1
    print("cost for suck" + 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)
printf("performance measurement" +
                    str(cost))
```

SURYA Gold

Date 27/11/23 Page

8- Puzzle program

```
from collection import deque
class puzzle8:
    def __init__(self, size=3):
        self.size = size

    def display_state(self, state):
        for i in range(0, len(state), self.size):
            print(" ".join(state[i:i+self.size]))
        print()

    def get_neighbours(self, state):
        neighbours = []
        blank = state.index(' ')
        row, col = divmod(blank, self.size)
        for dr, dc in [(0,1), (1,0), (0,-1), (-1,0)]:
            nr, nc = row + dr, col + dc
            if 0 <= nr < self.size and 0 <= nc < self.size:
                new_state = state[:]
                new_blank = nr * self.size + nc
                new_state[blank], new_state[new_blank] = \
                new_state[new_blank], new_state[blank]
                neighbours.append(new_state)
        return neighbours

    def is_goal(self, state, target_state):
        return state == target_state

    def bfs(self, initial_state, target_state):
        return
```

```
queue = deque([(initial_state, [])])
visited = set()

while queue:
    current_state, path = queue.popleft()
    if self.is_goal_state(current_state, target_state):
        return path

    if tuple(current_state) not in visited:
        visited.add(tuple(current_state))
        for neighbor in neighbors:
            queue.append((neighbor, path + [neighbor]))
return None

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

puzzle = Puzzle8()
solution = puzzle.bfs(initial_state, goal_state)

if solution:
    print("Solution found:")
    for step, state in enumerate(solution):
        print(f"Step{step+1}:")
        puzzle.display_state(state)
        print()
else:
    print("No solution found.")
```

SURYA Gold

Date Page

OUTPUT of Vaccum cleaner

Enter location of Vaccum B

Enter status of B 1

Enter status of other room 1

Initial Location Condition {'A': 1, 'B': '1'}

Vaccum is placed in location B

Location B is dirty

Cost for CLEANING 1

Location B has been cleaned.

Location A is Dirty.

Moving LEFT to the location A.

Cost for moving LEFT 2

Cost for SUCK 3

Location A has been cleaned.

GOAL STATE:

{'A': '0', 'B': '0'}

Performance Measurement: 3

SURYA Gold
Date Page

WEEK-4

```
def iterative_deepening_search(src, target):
    depth_limit = 0;
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is 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)
    poss_moves_to_do = possible_moves(src, visited_states)

    for move in poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
            result = depth_limited_search(move, target,
                                          depth_limit-1, visited_
                                          states)
            if result is not None:
                return result
    return result 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]
    elif m == 'l':
        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
          {state[3]} {state[5]}\n {state[6]}
          {state[7]} {state[8]}\n")
```

SURYA Gold
Date Page

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

iterativeDeepeningSearch(src, target)

OUTPUT

0 2 3
1 4 5
6 7 8

1 2 3
6 4 5
0 7 8

1 2 3
6 4 5
0 7 8

1 2 3
6 4 5
7 0 8

1 2 3
4 0 5
6 7 8

1 2 3
4 0 5
6 7 8

0 2 3
1 4 5
6 7 8

1 0 3
4 2 5
6 7 8

2 0 3
1 4 5
6 7 8

1 2 3
4 7 5
6 0 8

1 2 3
6 4 5
0 7 8

1 2 3
4 5 0
6 7 8

1 2 3
4 5 0
6 7 8

Success

SURYA Gold
Date Page

8-Puzzle using Best first Search algorithm

```
import queue as Q
goal = [[1,2,3],[4,5,6],[7,8,0]]

def HeuristicValue(state):
def ~~isGoal(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 isGoal(state):
    return state == goal

def getCoordinates(current state):
    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) -> rut:
    visited = set()
    Pq = Q.PriorityQueue()
    Pq.put(HeuristicValue(state), 0, state))

    while not Pq.empty():
    -, moves, current state = Pq.get()
```

SURYA Gold

Date Page

```
if currentstate == goal:
    return moves

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

visited.add(tuple(map(tuple, currentstate)))
coordinates = getcoordinates(currentstate)
i, j = coordinates[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 currentstate]
    new_state[i][j], new_state[new_i][new_j] =
                    new_state[new_i][new_j], new_state[i][j]
    if tuple(map(tuple, new_state)) not in visited:
        pq.put((heuristicvalue(new_state), moves+1,
                    new_state))

return -1

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

SURYA Gold
Date Page

8-Puzzle using A* search algorithm

```
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(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 current state == goal:
    return moves

if tuple(map(tuple, currentState)) in visited:
    continue

visited.add(tuple(map(tuple, currentstate)))
coordinates = get coordinates(currentState)
i,j = coordinates[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 currentState]
    new_state[i][j], new_state[new_i][new_j] =
        new_state[new_i][new_j], new_state[i][j]
    if tuple(map(tuple, new_state)) not in visited:
        pq.put((Heuristic value(new_state) + moves,
                moves+1, new_state))
    if new_state == goal:
        print(new_state)
return -1

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

SURYA Gold
Date 27/12/23 Page 23

Create a KB using propositional logic & show that the given query entails the KB or not

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

def generate_truth_table():
    print("q | P | R | Expression(KB) | Query(R)")
    print("-- | - | - | - - - - - | - - - ")

    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
                print(f"{q} | {P} | {R} | {
                    expression_result}
                    | {query_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("In Query entails the knowledge.")
    else:
        print("In Query does not entail the knowle-
              -dge.")

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

IP Enter rule: (~q v ~p v r) ^ (~q ^ p) ^ q

Query = r

exp result

True | True | True | False

True | True | False | False

True | False | True | False

False | True | False | True

False | True | True | False

Logic entails query.

SURYA Gold
Date Page

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]}')
        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]}']
```

SURYA Gold

Date Page

else: if

contradiction(goal, f'{gun[0]}v{gun[1]}'):

temp.append(f'{gun[0]}v{gun[1]}') steps[''].

f"Resolved {temp[i]} and {temp[j]} to

{temp[-1]}.

contradiction

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 = 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=>Q, P=>Q

~PvQ, Q=>R, ~QvR

goal = 'R'

main(rules, goal)

rules = 'PvQ PvR ~PvR RvS Rv~Q ~SvQ' #

(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)

main(rules, 'P')

SURYA Gold

Date Page

OUTPUT

step	clause	Derivation
1	R ∨ P	given
2	R ∨ ~Q	given
3	~R ∨ P	given
4	~P ∨ Q	given
5	~R	negation
		Resolved R ∨ ~R and ~R ∨ P to R ∨ P, which is in the form of null

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

Unification

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

    if func1 != func2:
        print("Expression 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 None

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

OUTPUT:

```
Enter the first expression : f(x,y)
Enter the 2nd expression : f(a,b)
The substitution are :
x|a
y|b

Unified expression 1: f(a,b)
Unified expression 2: f(a,b)
```

SURYA Gold
Date 17/1/24 Page

Convert given fol to 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-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, statements)
    for i, s in enumerate(statements)
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
```

SURYA Gold

Date Page

```
statement = statement.replace(s, fol_to_cnf(s))
while '-' in statements:
    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)]"))
```

o/p

~bird(x) | ~fly(x)

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

7/1/24

SURYA Gold

Date Page

Create a KB consisting of fol statements and prove the given query using forward reasoning

```
import re

def isVariable(x):
    return len(x) == 1 & x.islower() & 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 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.lhs.expression)
                            [0], str(getAttributes(self.
                            lhs.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.implication = 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)')
```

O/P

Querying evil(x):

1. evil(John)

2/12/24

SURYA Gold
Date Page

WEEK-4

```
def iterative_deepening_search(src, target):
    depth_limit = 0;
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is 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)
    poss_moves_to_do = possible_moves(src, visited_states)

    for move in poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
            result = depth_limited_search(move, target,
                                          depth_limit-1, visited_
                                          states)
            if result is not None:
                return result
    return result None

def possible_moves(state, visited_states):
    b = state.index(0)
```

SURYA Gold

Date _____ Page _____

```
scores = []
for move in actions(board):
    result(board, move)

scores.append(minimax_helper(board))
    board[move[0]][move[1]] = EMPTY
return max(scores) if isMaxTurn else
min(scores)


def minimax(board):
    isMaxTurn = True if player(board) == X else
    False
    bestmove = None,
    if isMaxturn:
        bestscore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestscore):
                bestscore = score
                bestmove = move
        return bestmove
    else:
        bestscore = +math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score < bestscore):
                bestscore = score
                bestmove = move
        return bestmove
```

SURYA Gold

Date Page

Create a KB consisting of fol statements and prove the given query using forward reasoning

```
import re

def isVariable(x):
    return len(x) == 1 & x.islower() & 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]
```

```
for i in [0,1,2]:
    strikeD.append(board[i][i])
if (strikeD[0] == strikeD[1] == strikeD[2]):
    return strikeD[0]
if (board[0][2] == board[1][1] == board[2][0]):
    return board[0][2]
return None

def terminal(board):
    full = True
    for i in [0,1,2]:
        for j in board[i]:
            if j is None:
                full = False
    if full:
        return True
    if (winner(board) is not None):
        return True
    return False

def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:
        return -1
    else:
        return 0

def minmax_helper(board):
    isMaxTurn = True if Player(board) == X else
                False
    if terminal(board):
        return utility(board):
```

else:

if status_input == '0':
    print("Location A is already clean")

if status_input_complement == '1':
    print("Location B is dirty.")
    print("moving right to location B.")
    cost += 1;
    print("cost for moving right" + str(cost))
    goal_state['B'] = '0'
    cost += 1
    print("cost for suck" + 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)

```
queue = deque([(initial_state, [])])
visited = set()

while queue:
    current_state, path = queue.popleft()
    if self.is_goal_state(current_state, target_state):
        return path

    if tuple(current_state) not in visited:
        visited.add(tuple(current_state))
        for neighbor in neighbors:
            queue.append((neighbor, path + [neighbor]))
return None

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

puzzle = Puzzle8()
solution = puzzle.bfs(initial_state, goal_state)

if solution:
    print("Solution found:")
    for step, state in enumerate(solution):
        print(f"Step{step+1}:")
        puzzle.display_state(state)
        print()
else:
    print("No solution found.")
```

```
                freeboxes.add((i,j))
        return freeboxes

def result(board, action):
    i = action[0];
    j = action[1];
    if type(action) == list:
        action = (i,j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X;
        elif player(board) == O:
            board[i][j] = O
    return board

def winner(board):
    if (board[0][0] == board[0][1] == board[0][2] == X
or board[1][0] == board[1][1] == board[1][2] == X
or board[2][0] == board[2][1] == board[2][2] == X):
        return X;

    if (board[0][0] == board[0][1] == board[0][2] == O
or board[1][0] == board[1][1] == board[1][2] == O or
board[2][0] == board[2][1] == board[2][2] == O);
    return O;

for i in [0,1,2]:
    s2 = []
    for j in [0,1,2]:
        s2.append(board[j][i])
    if (s2[0] == s2[1] == s2[2]):
        return s2[0]
strike0 = []
```

SURYA Gold
Date Page

8-Puzzle using Best first Search algorithm

```
import queue as Q
goal = [[1,2,3],[4,5,6],[7,8,0]]

def HeuristicValue(state):
def ~~isGoal(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 isGoal(state):
    return state == goal

def getCoordinates(current state):
    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) -> rut:
    visited = set()
    Pq = Q.PriorityQueue()
    Pq.put(HeuristicValue(state), 0, state))

    while not Pq.empty():
    -, moves, current state = Pq.get()
```

SURYA Gold
Date 22/11/23 Page

Implement tic tac toe

```
import math
import copy

X = 'X'
O = 'O'
EMPTY = None
def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO = 0;
    countX = 0;
    for y in [0,1,2]:
        for x in board[y]:
            if x == "O":
                countO = countO+1;
            elif x == "X":
                countX = countX+1;
    if countO >= countX:
        return X
    elif countX > countO:
        return O

def actions(board):
    freeboxes = set();
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board[i][j] == EMPTY;
```

SURYA Gold

Date Page

OUTPUT of Vaccum cleaner

Enter location of Vaccum B

Enter status of B 1

Enter status of other room 1

Initial Location Condition {'A': 1, 'B': '1'}

Vaccum is placed in location B

Location B is dirty

Cost for CLEANING 1

Location B has been cleaned.

Location A is Dirty.

Moving LEFT to the location A.

Cost for moving LEFT 2

Cost for SUCK 3

Location A has been cleaned.

GOAL STATE:

{'A': '0', 'B': '0'}

Performance Measurement: 3

SURYA Gold
Date Page

8-Puzzle using A* search algorithm

```
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(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()
```

SURYA Gold

Date Page

```
statement = statement.replace(s, fol_to_cnf(s))
while '-' in statements:
    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)]"))
```

o/p

~bird(x) | ~fly(x)

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

7/1/24

SURYA Gold
Date Page

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]}')
        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]}']
```

```
        if v:
            constants[v] = fact.getConstants()[i]

        new_lhs.append(fact)
    predicate, attributes = getPredicates(self.lhs.expression)
                            [0], str(getAttributes(self.
                            lhs.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.implication = 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])
```

SURYA Gold
Date 17/1/24 Page

Convert given fol to 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-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, statements)
    for i, s in enumerate(statements)
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
```

```
def print_board(board):
    for row in board:
        print(row);

game_board = initial_state()
print("Initial Board:")
print_board(game_board)

while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("\nEnter your move
                           (row, column): ")
        row, col = map(int, user_input.split(','))
        result(game_board, (row, col))
    else:
        print("\nAI is making a move...")
        move = minimax(copy.deepcopy(game_board))
        result(game_board, move)

print("\nCurrent Board:")
print_board(game_board)

if winner(game_board) is not None:
    print(f"\nThe winner is: {winner(game_board)
          }")
else:
    print("\nIt's a tie!")
```

OUTPUT

enter your move

X		

enter your move
AI making move

X		
O		

enter your move

X		
O	X	

AI move

X		
O	X	
	O	

enter your move

X		
O	X	
	O	X

Player wins

```
def getResult(self):
    return self.result

def getConstants(self):
    return [None if isVariable(c) else 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()):
```

SURYA Gold
Date 27/12/23 Page 23

Create a KB using propositional logic & show that the given query entails the KB or not

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

def generate_truth_table():
    print("q | P | R | Expression(KB) | Query(R)")
    print("-- | - | - | - - - - - | - - - ")

    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
                print(f"{q} | {P} | {R} | {
                    expression_result}
                    | {query_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
```

```
    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]
    elif m == 'l':
        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
          {state[3]} {state[5]}\n {state[6]}
          {state[7]} {state[8]}\n")
```

Unification

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

    if func1 != func2:
        print("Expression 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 None

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

OUTPUT:

```
Enter the first expression : f(x,y)
Enter the 2nd expression : f(a,b)
The substitution are :
x|a
y|b

Unified expression 1: f(a,b)
Unified expression 2: f(a,b)
```

SURYA Gold
Date 22/11/23 Page

Vaccum cleaner Agent

```
def vaccum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter location of vaccum")
    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("vaccum is placed in location A")
        if status_input == '1':
            print("location A is Dirty")
            goal_state['A'] = '0'
            cost += 1
            print("cost for cleaning A" + str(cost))
            print("location A has been 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 current state == goal:
    return moves

if tuple(map(tuple, currentState)) in visited:
    continue

visited.add(tuple(map(tuple, currentstate)))
coordinates = get coordinates(currentState)
i,j = coordinates[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 currentState]
    new_state[i][j], new_state[new_i][new_j] =
        new_state[new_i][new_j], new_state[i][j]
    if tuple(map(tuple, new_state)) not in visited:
        pq.put((Heuristic value(new_state) + moves,
                moves+1, new_state))
    if new_state == goal:
        print(new_state)
return -1

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

SURYA Gold

Date 27/11/23 Page

8- Puzzle program

```
from collection import deque
class puzzle8:
    def __init__(self, size=3):
        self.size = size

    def display_state(self, state):
        for i in range(0, len(state), self.size):
            print(" ".join(state[i:i+self.size]))
        print()

    def get_neighbours(self, state):
        neighbours = []
        blank = state.index(' ')
        row, col = divmod(blank, self.size)
        for dr, dc in [(0,1), (1,0), (0,-1), (-1,0)]:
            nr, nc = row + dr, col + dc
            if 0 <= nr < self.size and 0 <= nc < self.size:
                new_state = state[:]
                new_blank = nr * self.size + nc
                new_state[blank], new_state[new_blank] = \
                new_state[new_blank], new_state[blank]
                neighbours.append(new_state)
        return neighbours

    def is_goal(self, state, target_state):
        return state == target_state

    def bfs(self, initial_state, target_state):
        return
```

SURYA Gold

Date Page

OUTPUT

step	clause	Derivation
1	R ∨ P	given
2	R ∨ ~Q	given
3	~R ∨ P	given
4	~P ∨ Q	given
5	~R	negation
		Resolved R ∨ ~R and ~R ∨ P to R ∨ P, which is in the form of null

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

```
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)
printf("performance measurement" +
                    str(cost))
```

```
        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)')
```

O/P

Querying evil(x):

1. evil(John)

2/12/24

1. Implement Tic –Tac –Toe Game.

```
import math
import copy

X = "X"
O = "O"
EMPTY = None

def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO = 0
    countX = 0
    for y in [0, 1, 2]:
        for x in board[y]:
            if x == "O":
                countO = countO + 1
            elif x == "X":
                countX = countX + 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O


def actions(board):
```

```
    freeboxes = set()
    for i in [0, 1, 2]:
        for j in [0, 1, 2]:
            if board[i][j] == EMPTY:
                freeboxes.add((i, j))
    return freeboxes


def result(board, action):
    i = action[0]
    j = action[1]
    if type(action) == list:
        action = (i, j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X
        elif player(board) == O:
            board[i][j] = O
    return board


def winner(board):
    if (board[0][0] == board[0][1] == board[0][2] == X or board[1][0] == board[1][1] ==
board[1][2] == X or board[2][0] == board[2][1] == board[2][2] == X):
        return X
    if (board[0][0] == board[0][1] == board[0][2] == O or board[1][0] == board[1][1] ==
board[1][2] == O or board[2][0] == board[2][1] == board[2][2] == O):
        return O
    for i in [0, 1, 2]:
        s2 = []
        for j in [0, 1, 2]:
```

```
            s2.append(board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]
    strikeD = []
    for i in [0, 1, 2]:
        strikeD.append(board[i][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        return strikeD[0]
    if (board[0][2] == board[1][1] == board[2][0]):
        return board[0][2]
    return None


def terminal(board):
    Full = True
    for i in [0, 1, 2]:
        for j in board[i]:
            if j is None:
                Full = False
    if Full:
        return True
    if (winner(board) is not None):
        return True
    return False


def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:
```

```
        return -1
    else:
        return 0


def minimax_helper(board):
    isMaxTurn = True if player(board) == X else False
    if terminal(board):
        return utility(board)

    scores = []
    for move in actions(board):
        result(board, move)
        scores.append(minimax_helper(board))
        board[move[0]][move[1]] = EMPTY
    return max(scores) if isMaxTurn else min(scores)


def minimax(board):
    isMaxTurn = True if player(board) == X else False
    bestMove = None
    if isMaxTurn:
        bestScore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move
```

```
        return bestMove
    else:
        bestScore = +math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score < bestScore):
                bestScore = score
                bestMove = move
        return bestMove


def print_board(board):
    for row in board:
        print(row)


# Example usage:
game_board = initial_state()
print("Initial Board:")
print_board(game_board)

while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("\nEnter your move (row, column): ")
        row, col = map(int, user_input.split(','))
        result(game_board, (row, col))
    else:
        print("\nAI is making a move...")
```

```
        move = minimax(copy.deepcopy(game_board))
        result(game_board, move)

    print("\nCurrent Board:")
    print_board(game_board)

# Determine the winner
if winner(game_board) is not None:
    print(f"\nThe winner is: {winner(game_board)}")
else:
    print("\nIt's a tie!")
```

```
Initial Board:
[None, None, None]
[None, None, None]
[None, None, None]

Enter your move (row, column): 1,2

Current Board:
[None, None, None]
[None, None, 'X']
[None, None, None]

AI is making a move...

Current Board:
[None, None, None]
[None, 'O', 'X']
[None, None, None]

Enter your move (row, column): 0,0

Current Board:
['X', None, None]
[None, 'O', 'X']
[None, None, None]

AI is making a move...

Current Board:
['X', 'O', None]
[None, 'O', 'X']
[None, None, None]

Enter your move (row, column): 2,1
```

```
Current Board:
['X', 'O', None]
[None, 'O', 'X']
[None, 'X', None]

AI is making a move...

Current Board:
['X', 'O', None]
[None, 'O', 'X']
['O', 'X', None]

Enter your move (row, column): 1,0

Current Board:
['X', 'O', None]
['X', 'O', 'X']
['O', 'X', None]

AI is making a move...

Current Board:
['X', 'O', 'O']
['X', 'O', 'X']
['O', 'X', None]

The winner is: O
```

OUTPUT:

2. Solve 8 puzzle problems.

```
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(0)
```

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

    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')



    # 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

print("Example 1")
src= [2,0,3,1,8,4,7,6,5]
target=[1,2,3,8,0,4,7,6,5]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)

print("\nExample 2")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)
```

OUTPUT:

```
Example 1
Source:  [2, 0, 3, 1, 8, 4, 7, 6, 5]
Goal State:  [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]
[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

Example 2
Source:  [1, 2, 3, 0, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 4, 5, 0, 6, 7, 8]
[1, 2, 3, 0, 4, 5, 6, 7, 8]
[0, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 0, 7, 8]
[1, 2, 3, 4, 0, 5, 6, 7, 8]
[2, 0, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, 0, 8]
[1, 0, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, 0, 8]
[1, 2, 3, 4, 5, 0, 6, 7, 8]
Success
```

3. Implement Iterative deepening search algorithm.

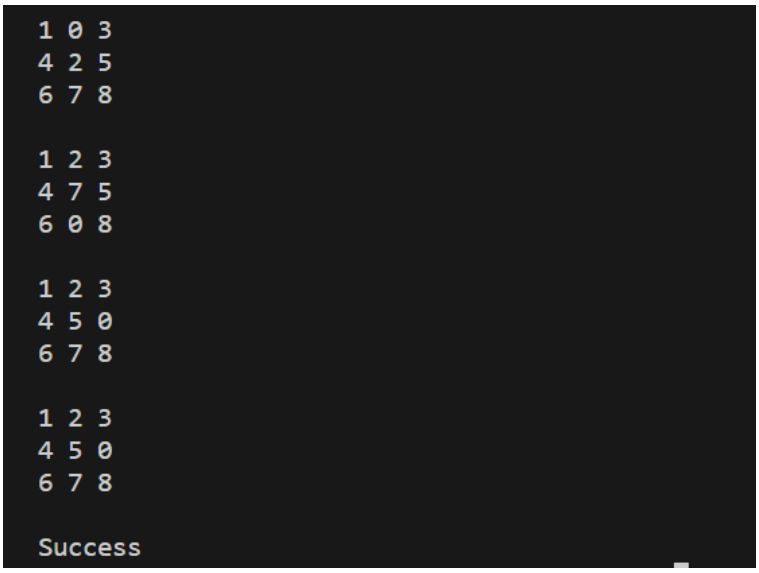
```
def iterative_deepening_search(src, target):
    depth_limit = 0
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is not None:
            print("Success")
            return
        depth_limit += 1
        if depth_limit > 30:  # Set a reasonable depth limit to avoid an infinite loop
            print("Solution not found within depth 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)
    poss_moves_to_do = possible_moves(src, visited_states)

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

```
            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 == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    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{state[3]} {state[4]} {state[5]}\n{state[6]}
{state[7]} {state[8]}\n")

print("Example 1")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
iterative_deepening_search(src, target)
```

OUTPUT:

```
Example 1
Source:  [1, 2, 3, 0, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 4, 5, 0, 6, 7, 8]
0 2 3
1 4 5
6 7 8

1 2 3
6 4 5
0 7 8

1 2 3
4 0 5
6 7 8

0 2 3
1 4 5
6 7 8

2 0 3
1 4 5
6 7 8

1 2 3
6 4 5
0 7 8

1 2 3
6 4 5
7 0 8

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



4. Implement A* search algorithm.

```
def print_grid(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
        f"""
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]} {state[8]}
        """
    )


def h(state, target):
    #Manhattan distance
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3
```

```
        dist += abs(x1-x2) + abs(y1-y2)
    return dist


def astar(src, target):
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        moves = []
        for state in states:
            visited_states.add(tuple(state))
            print_grid(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(state, visited_states) if move not in
moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i] for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")


def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if 9 > b - 3 >= 0:
        d += 'u'
    if 9 > b + 3 >= 0:
        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 tuple(move) not in visited_states]

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

#Test 1
print("Example 1")
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 2
print("Example 2")
```

```
src = [1,2,3,-1,4,5,6,7,8]
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 3
print("Example 3")
src = [1,2,3,7,4,5,6,-1,8]
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)
```

OUTPUT:

```
Example 1
Source:  [1, 2, 3, -1, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 4, 5, -1, 6, 7, 8]

1 2 3
  4 5
6 7 8


1 2 3
4   5
6 7 8


1 2 3
4 5
6 7 8

Success
Example 2
Source:  [1, 2, 3, -1, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 6, 4, 5, -1, 7, 8]

1 2 3
  4 5
6 7 8


1 2 3
6 4 5
  7 8

Success
```

```
Example 3
Source:  [1, 2, 3, 7, 4, 5, 6, -1, 8]
Goal State:  [1, 2, 3, 6, 4, 5, -1, 7, 8]

1 2 3
7 4 5
6   8


1 2 3
7 4 5
  6 8


1 2 3
  4 5
7 6 8


  2 3
1 4 5
7 6 8


1 2 3
4   5
7 6 8


1 2 3
4 6 5
7   8
```

```
1 2 3
  6 5
4 7 8


1 2 3
6   5
4 7 8


1 2 3
6 7 5
4   8


1 2 3
6 7 5
  4 8


1 2 3
  7 5
6 4 8


  2 3
1 7 5
6 4 8


1 2 3
7   5
6 4 8
```

```
7 1 3
4 6 5
  2 8


7 1 3
4 6 5
2   8


7 1 3
4   5
2 6 8


7 1 3
4 6 5
2 8


7 1 3
  4 5
2 6 8


7 1 3
2 4 5
  6 8

Fail
```

5. Implement vacuum cleaner agent.

```
def clean(floor, row, col):
    i, j, m, n = row, col, len(floor), len(floor[0])
    goRight = goDown = True
    cleaned = [not any(f) for f in floor]
    while not all(cleaned):
        while any(floor[i]):
            print_floor(floor, i, j)
            if floor[i][j]:
                floor[i][j] = 0
                print_floor(floor, i, j)
            if not any(floor[i]):
                cleaned[i] = True
                break
            if j == n - 1:
                j -= 1
                goRight = False
            elif j == 0:
                j += 1
                goRight = True
            else:
                j += 1 if goRight else -1
        if all(cleaned):
            break
        if i == m - 1:
            i -= 1
            goDown = False
        elif i == 0:
```

```
            i += 1
            goDown = True
        else:
            i += 1 if goDown else -1
        if cleaned[i]:
            print_floor(floor, i, j)

def print_floor(floor, row, col): # row, col represent the current vacuum cleaner position
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r == row and c == col:
                print(f" >{floor[r][c]}< ", end = '')
            else:
                print(f"  {floor[r][c]}  ", end = '')
        print(end = '\n')
    print(end = '\n')

# Test 1
floor = [[1, 0, 0, 0],
         [0, 1, 0, 1],
         [1, 0, 1, 1]]

print("Room Condition: ")
for row in floor:
    print(row)
print("\n")
clean(floor, 1, 2)
```

OUTPUT:

```
Room Condition:
[1, 0, 0, 0]
[0, 1, 0, 1]
[1, 0, 1, 1]


  1    0    0    0
  0    1   >0<   1
  1    0    1    1

  1    0    0    0
  0    1    0   >1<
  1    0    1    1

  1    0    0    0
  0    1    0   >0<
  1    0    1    1

  1    0    0    0
  0    1   >0<   0
  1    0    1    1

  1    0    0    0
  0   >1<   0    0
  1    0    1    1

  1    0    0    0
  0   >0<   0    0
  1    0    1    1

  1    0    0    0
  0    0    0    0
  1   >0<   1    1
```

```
 1    0    0    0
 0    0    0    0
>1<   0    1    1

 1    0    0    0
 0    0    0    0
>0<   0    1    1

 1    0    0    0
 0    0    0    0
 0   >0<   1    1

 1    0    0    0
 0    0    0    0
 0    0   >1<   1

 1    0    0    0
 0    0    0    0
 0    0   >0<   1

 1    0    0    0
 0    0    0    0
 0    0    0   >1<

 1    0    0    0
 0    0    0    0
 0    0    0   >0<

 1    0    0    0
 0    0    0   >0<
 0    0    0    0

 1    0    0   >0<
 0    0    0    0
 0    0    0    0
```

```
 1    0   >0<   0
 0    0    0    0
 0    0    0    0

 1   >0<   0    0
 0    0    0    0
 0    0    0    0

>1<   0    0    0
 0    0    0    0
 0    0    0    0

>0<   0    0    0
 0    0    0    0
 0    0    0    0
```

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

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

def generate_truth_table():
    print(" p | q | r | Expression (KB) | Query (p^r)")
    print("---|---|---|-----------------|------------")

    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r

                print(f" {p} | {q} | {r} | {expression_result}              | {query_result}")

def query_entails_knowledge():
    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r

                if expression_result and not query_result:
                    return False

    return True
```

```
def main():
    generate_truth_table()

    if query_entails_knowledge():
        print("\nQuery entails the knowledge.")
    else:
        print("\nQuery does not entail the knowledge.")

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

OUTPUT:

```
KB: (p or q) and (not r or p)

  p  |  q  |  r  | Expression (KB) | Query (p^r)
-----|-----|-----|-----------------|------------
 True | True | True | True           | True
 True | True | False | True          | False
 True | False | True | True          | True
 True | False | False | True         | False
 False | True | True | False         | False
 False | True | False | True         | False
 False | False | True | False        | False
 False | False | False | False       | False

Query does not entail the knowledge.
```

7. Create a knowledge base 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


split_terms('~PvR')
def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']
    return clause in contradictions or reverse(clause) in contradictions
def resolve(rules, goal):
```

```
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(goal,f'{gen[0]}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'
print('Rules: ',rules)
print("Goal: ",goal)
main(rules, goal)

rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
print('Rules: ',rules)
print("Goal: ",goal)
main(rules, goal)

rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
goal = 'R'
print('Rules: ',rules)
```

print("Goal: ",goal)

main(rules, goal)

OUTPUT:

```
Example 1
Rules:  Rv~P Rv~Q ~RvP ~RvQ
Goal:  R

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.

Example 2
Rules:  PvQ ~PvR ~QvR
Goal:  R

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.
```

```
Example 3
Rules:  PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
Goal:  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.
```

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

print("\nExample 1")
exp1 = "knows(f(x),y)"
exp2 = "knows(J,John)"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)

print("\nExample 2")
exp1 = "knows(John,x)"
```

```
exp2 = "knows(y,mother(y))"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)


print("\nExample 3")
exp1 = "Student(x)"
exp2 = "Teacher(Rose)"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

OUTPUT:

```
Example 1
Expression 1:  knows(f(x),y)
Expression 2:  knows(J,John)
Substitutions:
[('J', 'f(x)'), ('John', 'y')]

Example 2
Expression 1:  knows(John,x)
Expression 2:  knows(y,mother(y))
Substitutions:
[('John', 'y'), ('mother(y)', 'x')]

Example 3
Expression 1:  Student(x)
Expression 2:  Teacher(Rose)
Predicates do not match. Cannot be unified
Substitutions:
False
```

9. Convert a 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)


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


print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

OUTPUT:

```
Example 1
FOL: bird(x)=>~fly(x)
CNF:  ~bird(x)|~fly(x)

Example 2
FOL: ∃x[bird(x)=>~fly(x)]
CNF:  [~bird(A)|~fly(A)]

Example 3
FOL: animal(y)<=>loves(x,y)
CNF:  ~animal(y)<|loves(x,y)

Example 4
FOL: ∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]
CNF:  ∀x~[∀y[~animal(y)|loves(x,y)]]|[[loves(A,x)]]

Example 5
FOL: [american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)
CNF:  ~[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]|criminal(x)
```

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(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('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
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()

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:

```
Example 1
Querying criminal(x):
        1. criminal(West)
All facts:
        1. american(West)
        2. enemy(Nono,America)
        3. hostile(Nono)
        4. sells(West,M1,Nono)
        5. owns(Nono,M1)
        6. missile(M1)
        7. weapon(M1)
        8. criminal(West)

Example 2
Querying evil(x):
        1. evil(John)
```