B.M.S. COLLEGE OF ENGINEERING
BENGALURU Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

(22CS5PCAIN)

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Vedang Jaithalya
1BM21CS239

Department of Computer Science and
Engineering B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
Mar-June 2021

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by **Vedang Jaithalya (1BM21CS239)** who is bonafide student of **B.M.S. Collage of Engineering**. during the 5th Semester June-September 2023.

Signature of the Faculty Incharge:

Sandhya A Kulkarni
Assistant professor
Department of CSE
BMSCE, Bangalore

DR. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bangalore

Table of Contents

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)
    count_O = 0
    count_X = 0
    for i in [0, 1, 2]:
        for j in board[i]:
            if j == 'O'
                count_O = count_O + 1
            elif j == 'X':
                count_X = count_X + 1
    if count_O >= count_X:
        return X
    elif count_X > count_O:
        return O

def actions(board):
    freefields = set()
    for i in [0, 1, 2]
```

```
    for j in [0, 1, 2]:
        if board[i][j] == empty:
            free boxes.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 print_board(board):
    for row in board:
        print(row)

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][0] = 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 0
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[0] == strikeD[1] == strikeD[2]:
    return strikeD[0]
if (board[0][2] == board[1][1] == board[2][0]:
    return board[0][2]
return none

game_board = initial_state()
print("initial board")
print_board(game_board)

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

    print("current board")
    print_board(game_board)
```

if winner(game_board) is not none
    Print("The winner is: {winner(game_board)}")
else:
    Print("It's a tie")

Output:

Initial board:

[none, none, none
  none, none, none
  none, none, none]

enter your move (row, column): 1, 1

current board:

[none none none
  none 'X'   none
  none none none]

It is making a move

current board:

[O none none
  none X none
  none none none]

enter your move: 1, 2

It is making a move

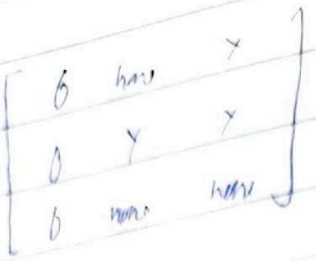
current board:

[O none none
  O X X
  none none none]

enter your move: 0, 3

It is making a move

current board



The winner is 'O'

8-puzzle

```
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_state)
    b = state.index(0)
    d = []
    if b not in [0,1,2]
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('l')
```

```
if 6not 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 moves_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

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

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

bfs(src, tarset)
```

output:

Step 1:
1 2 3
4 5 -1
6 7 8

Step 2:
1 2 3
4 5 9
6 7 -1

Step 3:
1 2 3
4 5 8
6 -1 7

Step 4:
1 2 3
4 5 8
-1 6 7

Step 5:
1 2 3
-4 5 8
4 6 7

Step 6:
1 2 3
5 -1 8
4 6 7

Step 7:
1 2 3
5 6 8
4 -1 7

Step 8:
1 2 3
5 6 8
4 7 -1

Step 9:
1 2 3
5 6 -1
4 7 8

Step 10:
1 2 3
5 -1 6
4 7 8

Step 11:
1 2 3
-1 5 6
4 7 8

Step 12:
1 2 3
4 5 6
-1 7 8

Step 13:
1 2 3
4 5 6
7 -1 8

Step 14:
1 2 3
4 5 6
7 8 -1

Vacuum Cleaner

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

    location_input = input("enter location ")
    Status_input = input("enter status of " + location_input)
    Status_input_complement = input("enter status of the
                                     room")
    print("Initial location condition" + str(goal_state))

    if location_input == 'A':
        print("Vacuum is placed in location A")
        if status_input == '1':
            print("location A is dirty")
            goal_state['A'] = 0
            cost = 1
        print("cost of cleaning A" + str(cost))
        print("location A has been cleaned")

    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 cleaning" + str(cost))
        print("location B has been cleaned")
    else:
```

print ("No action " + str(cost))
print ("location B is already clean")

if status_input == '0'
print ("location A is already Clean")
if Status_input_complement == '1':
print("location B is dirty")
print ("Moving right to location b")
cost += 1
Print ("cost to clean " + str(cost)
print ("location b 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 location A")
cost += 1
print ("cost to clean " + str(cost))
print ("location A has been cleaned")
else
print ("No action " + str(cost))
print ("location A is already clean")

Print ("Goal State: ")
print (Goal_State)
print ("Performance measurement" + str(cost))

Output:

Enter location of vacuum : a
Enter status of a : 1
enter status of other room : 1

initial location condition {'A' : '0', 'B' :
Vacuum is placed in location B
location B is dirty
cost of cleaning 1
location B has been cleaned
location A is dirty
moving left to A
cost for moving left 2
cost for sucking 3
location A has been cleaned

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

Performance measurement : 3

IDS - 8puzzle

```
def IDS(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 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_state)

            if result is not None:
```

```
return result
return none

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')

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

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
```

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

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

IDS (src, target)

output:

0 2 3
1 4 5
6 7 8

1 0 3
4 2 5
6 7 8

1 2 3
6 4 5
0 7 8

1 2 3
4 7 5
6 0 8

1 2 3
4 0 5
6 7 8

1 2 3
4 5 6
6 7 8

2 0 3
1 4 5
6 7 8

1 2 3
4 5 0
6 7 8

1 2 3
6 4 5
7 0 8

Success

1 2 3
4 0 5
6 7 8

13/12

BEST FIRST SEARCH

```
Import queue as Q

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

def isGoal(State):
    return State == goal

def HeuristicValue(State):
    ctr = 0
    for i in range(len(goal)):
        for j in range(len(goal[i])):
            if goal[i][j] != State[i][j]:
                ctr += 1
    return ctr

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

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

def BFS(State, goal) -> int:
    visited = Set()
    pq = Q.PriorityQueue()
    pq.put(HeuristicValue(State), 0, State))
```

while not pq.empty():
    moves, current_state = pq.get()
    if current_state == goal:
        return moves

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

visited.add(tuple(map(tuple, current_state)))

8 puzzle using A*

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

```
def isGoal(state):
    return state == goal

def HeuristicValue(state):
    in = 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 get getcoordinate(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

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

visited.add(tuple(map(tuple, current_state)))
coordinates = get_coordinates(current_state)
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 is_valid(new_i, new_j):
        new_state = [row[:] for row in current_state]
        if (tuple(map(tuple, new_state)) not in visited:
            pq.put((calculate_cost(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")
else:
    Print("Reached in " + str(moves) + " moves")
```

Output :

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

Reached in 14 moves

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

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

def generate_truth_table():
    print("expression (KB) | Query")
    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 = p and r
                print(f"{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 = p and r
                if expression_result and not query_result:
                    return False
    return True

def main():
    generate_truth_table()
    if query_entails_knowledge():
```

```
        print ("Query entails knowledge")
    else:
        print ("Query does not entail knowledge")

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

Output

```
True | True
True | False
False | False
True | False
True | True
True | False
False | False
False | False
```

27/12

6. 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('Step | Clause | Derivation')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f'{i}. \t | {steps[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('~P∨R')
['~P', 'R']

def contradiction(goal, clause):
    contradictions = [f'{goal}∨{negate(goal)}',
                      f'{negate(goal)}∨{goal}']
    return clause in contradictions or reversed(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
    clause = []
    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]):
                        clause += [f'{gen[0]}∨{gen[1]}']
```

~~{goal} is true~~

else:

if (contradiction(goal, f'{gen[0]} ∨ {gen[1]}')):

temp.append(f'{gen[0]} ∨ {gen[1]}')

return steps

elif len(gen)==1:

Clause += [f'{gen[0]}']

else:

if contradiction(goal, f'{terms1[0]} ∨ {terms2[0]}'):

temp.append(f'{terms1[0]} ∨ {terms2[0]}')

Steps['1'] = f'Resolved {temp[0]} and {temp[1]} to {temp[-1]}'

return Steps

rules = 'R∨~P R∨~Q ~R∨P ~R∨Q'

goal = 'R'

main(rules, goal)

rules = 'P ∨Q P∨Q ~P∨R R∨S R∨~Q ~S∨~Q'

main(rules, 'R')

output:

Steps	Clause	derivations
1	R∨~P	Given
2	R∨~Q	given
3	~R∨P	Given
4	~R∨Q	Given
5	~R	Negated conclusion
6.		resolved R∨~P and ~R∨P to R∨R which is in turn null.

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

Steps	Clause	Derivation
1.	$P \vee Q$	Given
2.	$P \vee R$	Given
3.	$\sim P \vee R$	Given
4.	$P \vee S$	Given
5.	$P \vee \sim Q$	Given
6.	$\sim S \vee \sim R$	Given
7.	$\sim R$	negated conclusion
8.	$Q \vee R$	Resolved from $P \vee Q$ and $\sim P \vee R$
9.	$P \vee \sim S$	resolved from $P \vee Q$ and $\sim S \vee \sim R$
10.	P	resolved from $P \vee R$ and $\sim R$
11.	$\sim P$	resolved from $\sim P \vee R$ and $\sim R$
12.	$R \vee \sim S$	resolved from $\sim P \vee Q$ and $P \vee \sim S$
13.	R	resolved from $\sim P \vee R$ and P
14.	S	resolved from $R \vee S$ and $\sim R$
15.	$R \vee \sim Q$	resolved from $R \vee \sim S$ and $\sim R$
16.	Q	resolved from $\sim R$ and $Q \vee R$
17.	$\sim S$	resolved from $\sim R$ and $R \vee \sim S$
18.		resolve $\sim R$ and R to $\sim P \vee R$,

which is in turn null. A contradiction is found when $\sim R$ is assumed is true, hence, R is true

10/11/24

Unification

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

    if func1 != func2:
        Print("expression cannot be unified")
        return none
    args1 = args1.rstrip(')').split(',')
    args2 = args2.rstrip(')').split(',')

    Substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1.is lower() and a2.lower() 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("expression cannot be unified. Incompatible
                    arguments.")
            return none
```

```
if __name__ == "__main__":

    expr1 = input ("Enter first expression: ")
    expr2 = input ("Enter second expression: ")

    Substitution = unify(expr1, expr2)

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

    expr1_result = apply_substitution(expr1, Substitution)
    expr2_result = apply_substitution(expr2, Substitution)
```

output :

First expression f(A, y) knows(a, b)
Second expression f(g, x) knows(x, g)

Substitution are :

g / A
b / y

unified expression 1 : f(A, y)
unified expression 2 : f(A, y)

Q. Convert given first order logic to 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-zA-Z~]+\([A-Za-z,]+\)'
    return re.findall(expr, String)

def Skolemization(Statement):
    SkolemConstant = [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],
                    SkolemConstant.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 = 'v' + statement[br:i] + '(' +
    statement = statement[:br] + statement[i+1:]
    statement = statement[:br] + new_statement
    if br > 0
    else new statement
return standardization(statement)
```

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

OUTPUT:

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

24/1

Q. Create a knowledgebase consisting of first order logic 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([c.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 [c].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 = get_predicates(self.rhs.expressions[0])
str(get_attributes(self.rhs.expressions)[0])
for key in constants:
    if constant[key]:
        attributes = attributes.replace(key, constant[key])
expr = f'{predicate}({attributes})'
return fact(expr) if len(new_lhs) and all([f.get_result()
    for f in new_lhs]) else None


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

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

def query(self, e):
    facts = set([f.expression for f in self.facts])
    i = 1
    print(f'Querying {e}:')
    for f in facts:
        if fact(f).predicate == fact(e).predicate:
```

```
        print(f'\t{i}.{f}')
        i += 1

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

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

Output:

```
query: evil(x):
    1: evil(John)
```

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]
    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!")

Output:

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

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

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

AI is making a move...

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

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

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

AI is making a move...

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

Enter your move (row, column): 0,2

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

AI is making a move...

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

The winner is: O
```

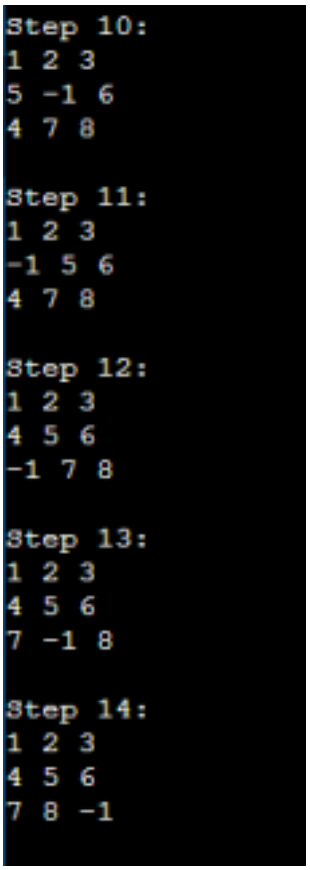
8-PUZZLE

```
from collections import deque

# Class representing the 8 Puzzle problem
class Puzzle8:
    def __init__(self, size=3):
        self.size = size

    def display_state(self, state):
        for i in range(self.size):
            for j in range(self.size):
                                print(state[i * self.size + j], end=" ")
            print()

    def get_blank_index(self, state):
        return state.index(-1)

    def get_neighbors(self, state):
        neighbors = []
                blank_index = self.get_blank_index(state)
        row, col = divmod(blank_index, self.size)

                            moves = [(0, 1), (1, 0), (0, -1), (-1, 0)] # right, down, left, up

        for move in moves:
                                new_row, new_col = row + move[0], col + move[1]

            if 0 <= new_row < self.size and 0 <= new_col < self.size: new_state =
                state[:]
                                new_blank_index = new_row * self.size + new_col

                                # Swap the blank tile with the neighbor
                new_state[blank_index], new_state[new_blank_index] =
new_state[new_blank_index], new_state[blank_index]

                neighbors.append(new_state)

        return neighbors
    def is_goal_state(self, state, target_state):
        return state == target_state

    def bfs(self, initial_state, target_state):
        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))
                                neighbors = self.get_neighbors(current_state)

                for neighbor in neighbors:
                    queue.append((neighbor, path + [neighbor])) return None

# Example usage:
initial_state = [1, 2, 3, 4, -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.")
```

Output:

```
Solution found:
Step 1:
1 2 3
4 5 -1
6 7 8

Step 2:
1 2 3
4 5 8
6 7 -1

Step 3:
1 2 3
4 5 8
6 -1 7

Step 4:
1 2 3
4 5 8
-1 6 7

Step 5:
1 2 3
-1 5 8
4 6 7

Step 6:
1 2 3
5 -1 8
4 6 7

Step 7:
1 2 3
5 6 8
4 -1 7

Step 8:
1 2 3
5 6 8
4 7 -1

Step 9:
1 2 3
5 6 -1
4 7 8

Step 10:
1 2 3
5 -1 6
4 7 8
```

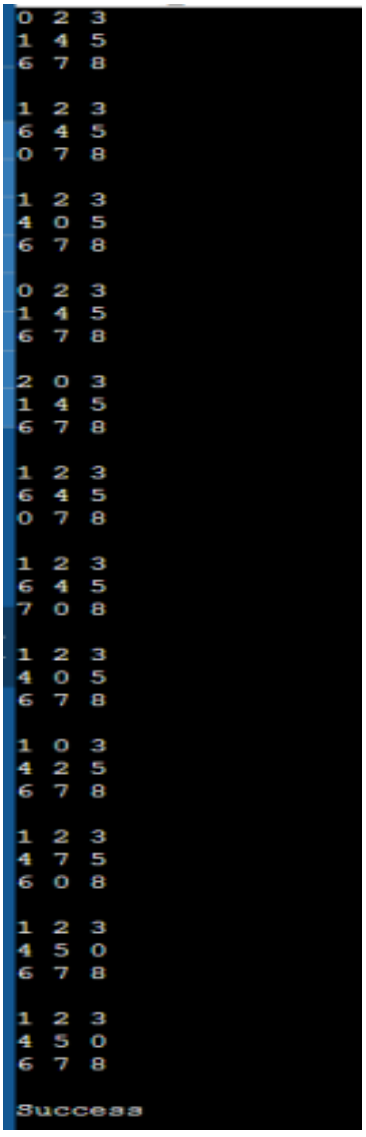

Step 10:
1 2 3
5 -1 6
4 7 8
Step 11:
1 2 3
-1 5 6
4 7 8
Step 12:
1 2 3
4 5 6
-1 7 8
Step 13:
1 2 3
4 5 6
7 -1 8
Step 14:
1 2 3
4 5 6
7 8 -1

IDS 8-PUZZLE

```
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")

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

 iterative_deepening_search(src, target)
```

Output:

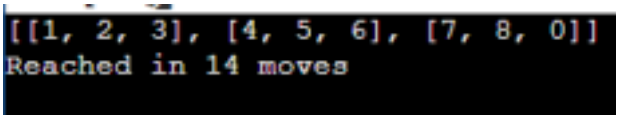


8-PUZZLE-A*

```
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 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, moves + 1, new_state))
                    if new_state == goal: # Print only states leading to the 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")
```

OUTPUT:

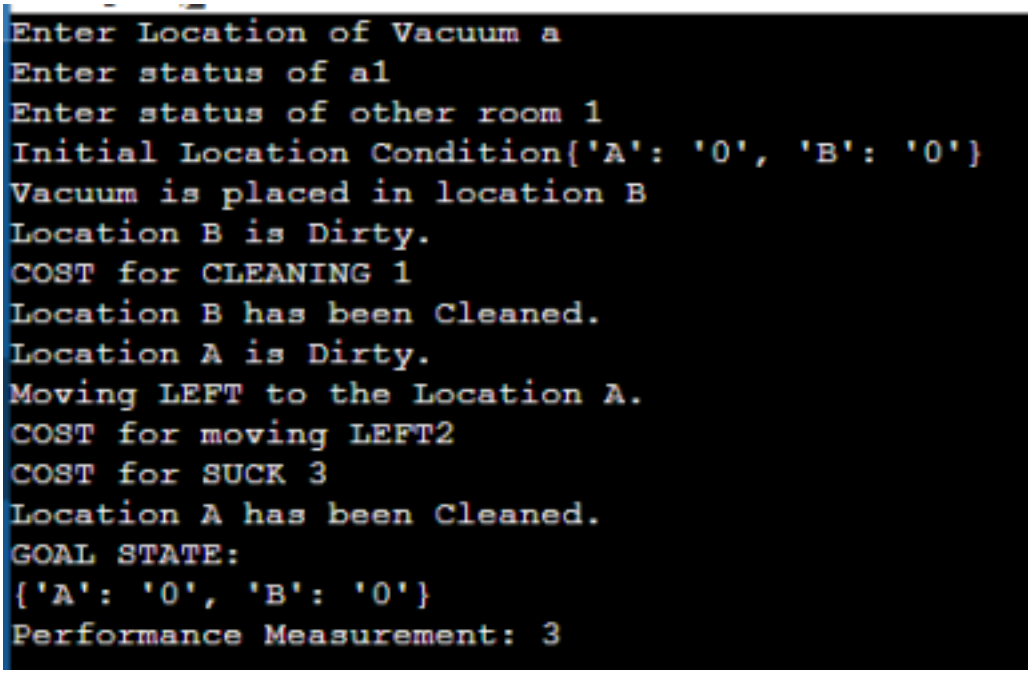


VACUUM CLEANER

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

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

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

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

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

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

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

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

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

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

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

vacuum_world()
```

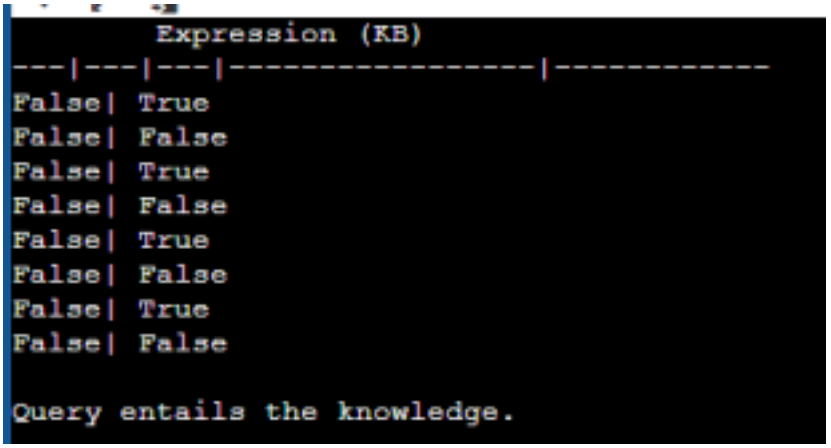
Output :


Enter Location of Vacuum a
Enter status of a1
Enter status of other room 1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum 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 LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3

Q)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 q
    return expression_result

def generate_truth_table():
    print("\tExpression (KB)")
    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"{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("\nQuery entails the knowledge.")
    else:
        print("\nQuery does not entail the knowledge.")

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

OUTPUT:

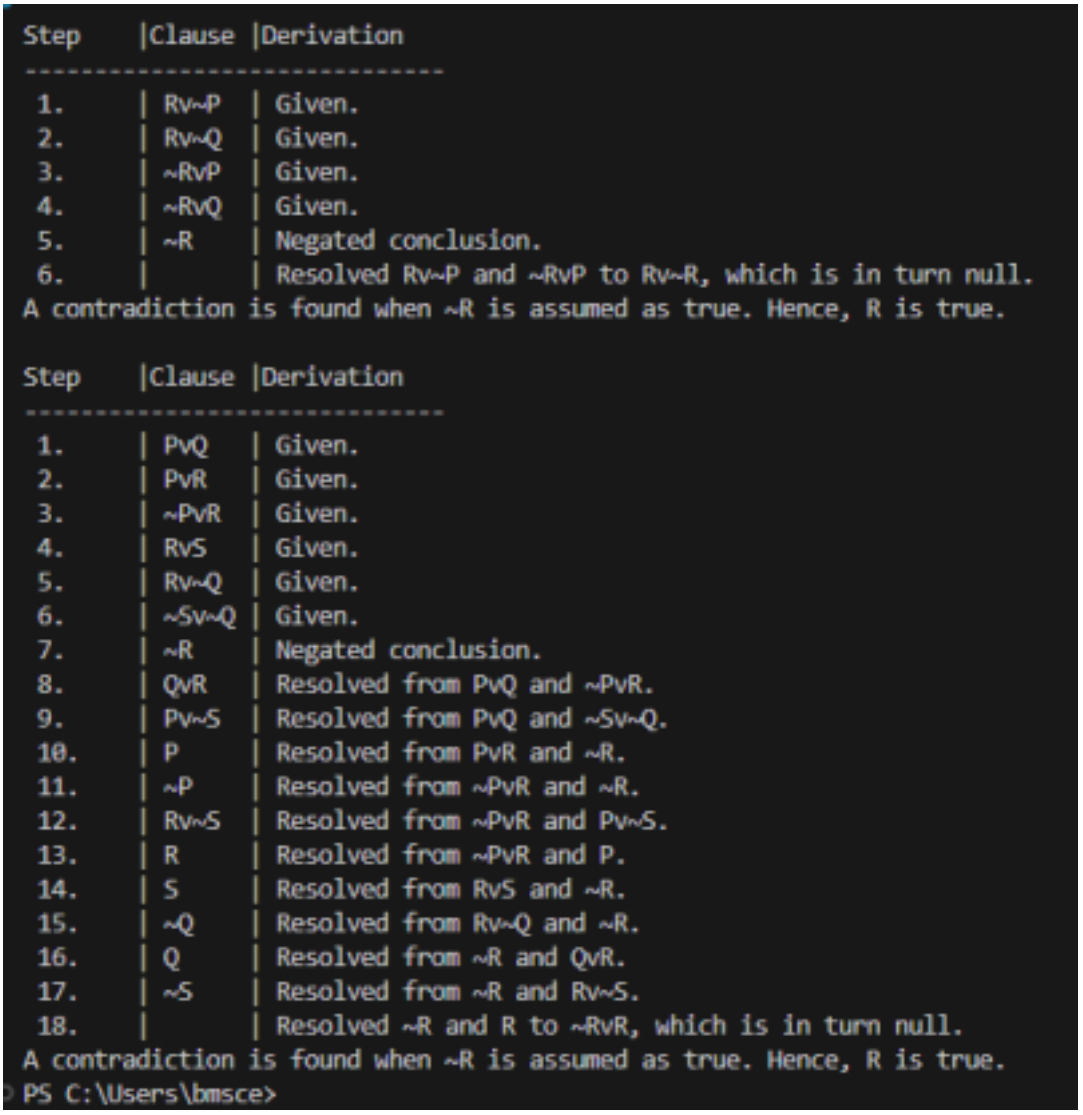

Expression (KB)
---|---|---|------------------|------------
False| True
False| False
False| True
False| False
False| True
False| False
False| True
False| False
Query entails the knowledge.

Q) Create a knowledgebase using propositional logic and prove the given query using Resolution.

```
import re

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

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

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''
def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_terms('~PvR')
['~P', 'R']
def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']
    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'
goal = 'R'
main(rules, goal)
rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q'
main(rules, 'R')
```

OUTPUT:



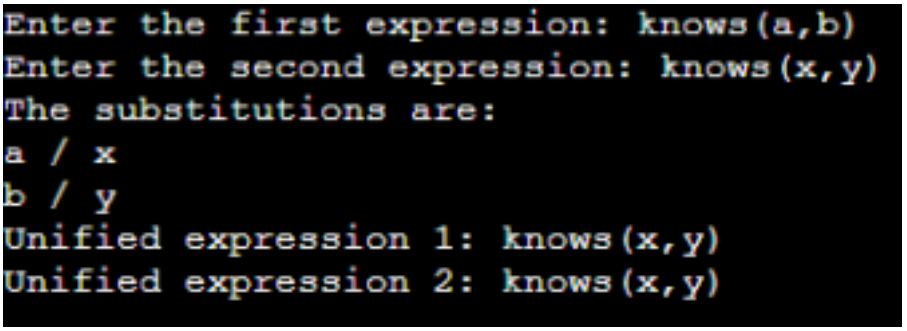
```
Step    |Clause |Derivation
-----------------------------
 1.     | RV~P  | Given.
 2.     | RV~Q  | Given.
 3.     | ~RVP  | Given.
 4.     | ~RVQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved RV~P and ~RVP to RV~R, which is in turn null.
A contradiction is found when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

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.
PS C:\Users\bmsce>
```

UNIFICATION

```
def unify(expr1, expr2):
    # Split expressions into function and arguments
    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    # Check if functions are the same
    if func1 != func2:
        print("Expressions cannot be unified. Different functions.") return None

    args1 = args1.rstrip(')').split(',')
    args2 = args2.rstrip(')').split(',')

    substitution = {}

    # Unify arguments
    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(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return expr


# Main program
if __name__ == "__main__":
    # Sample input
    expr1 = input("Enter the first expression: ")
    expr2 = input("Enter the second expression: ")
    # Unify expressions
    substitution = unify(expr1, expr2)

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

        # Apply substitution to original expressions
        expr1_result = apply_substitution(expr1, substitution) expr2_result =
```

```
    apply_substitution(expr2, substitution)

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

OUTPUT:



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-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("[american(x)^weapon(y)^sells(x,y,z)^hostile(z)] => criminal(x)"))
```

OUTPUT:

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

```
import re

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

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

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

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

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

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

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

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

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

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

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

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

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

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