**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**



**LAB REPORT**

**on**

Artificial Intelligence

***Submitted by***

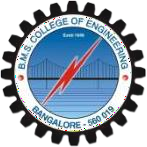
**RASKSHITHA DN (1BM22CS415)**

***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 **RAKSHITHA D N (1BM22CS415),** 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.

## Dr. M Lakshmi Neelima Dr. Jyothi S Nayak

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |
| --- | --- |
| **Lab Program No.** | **Program Details** |
| 1 | Implement Tic –Tac –Toe Game. |
| 2 | Solve 8 puzzle problems. |
| 3 | Implement Iterative deepening search algorithm. |
| 4 | Implement A\* search algorithm. |
| 5 | Implement vaccum cleaner agent. |
| 6 | Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not. |
| 7 | Create a knowledge base using prepositional logic and prove the given query using resolution |
| 8 | Implement unification in first order logic |
| 9 | Convert a given first order logic statement into Conjunctive Normal Form (CNF). |
| 10 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. |

**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. |

## 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)

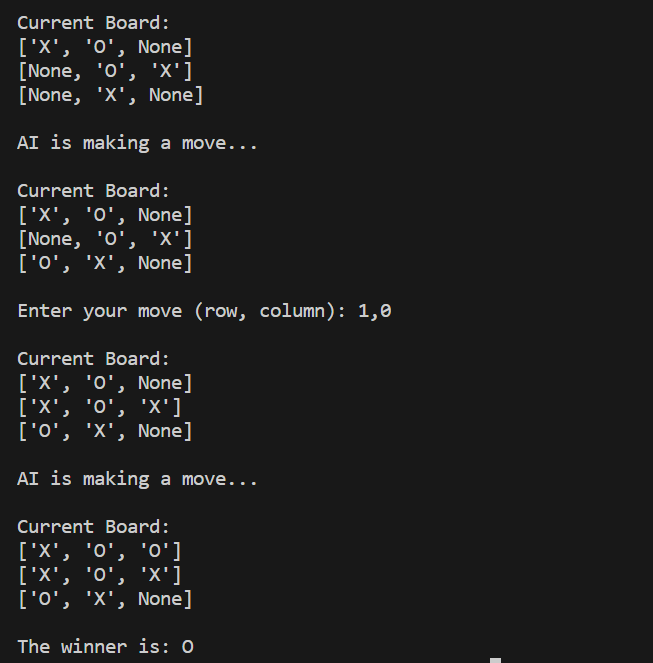
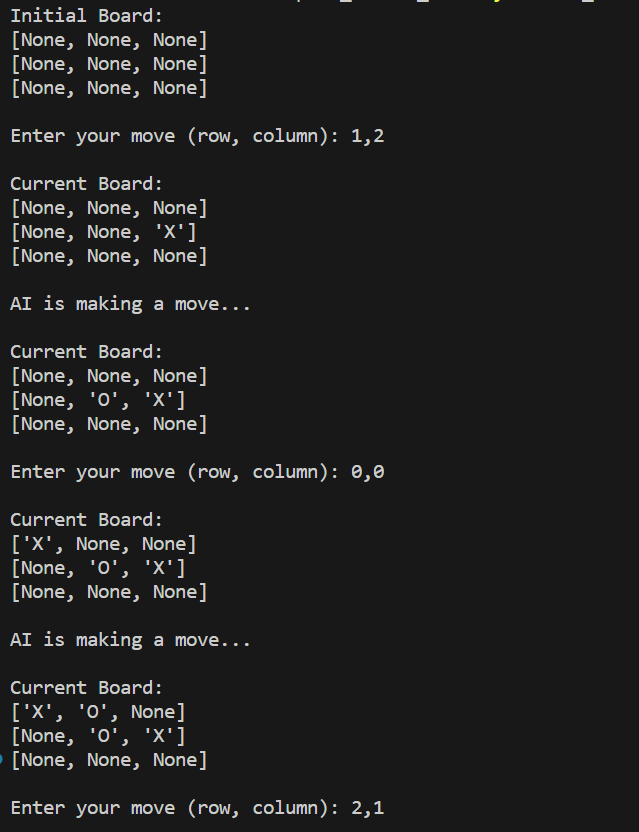
# Determine the winner

if winner(game\_board) is not None:

print(f"\nThe winner is: {winner(game\_board)}") else:

print("\nIt's a tie!")

# OUTPUT:



## 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)

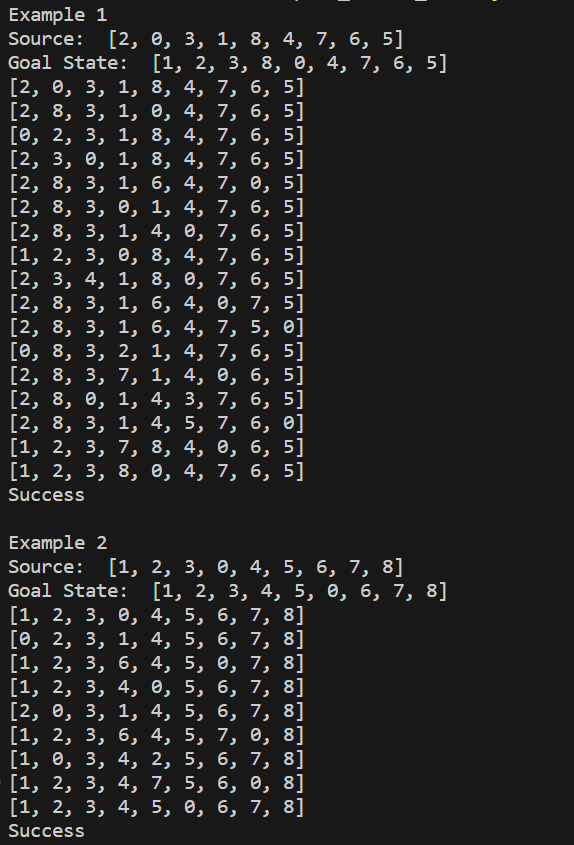
print("\nExample 2")

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:



## 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")

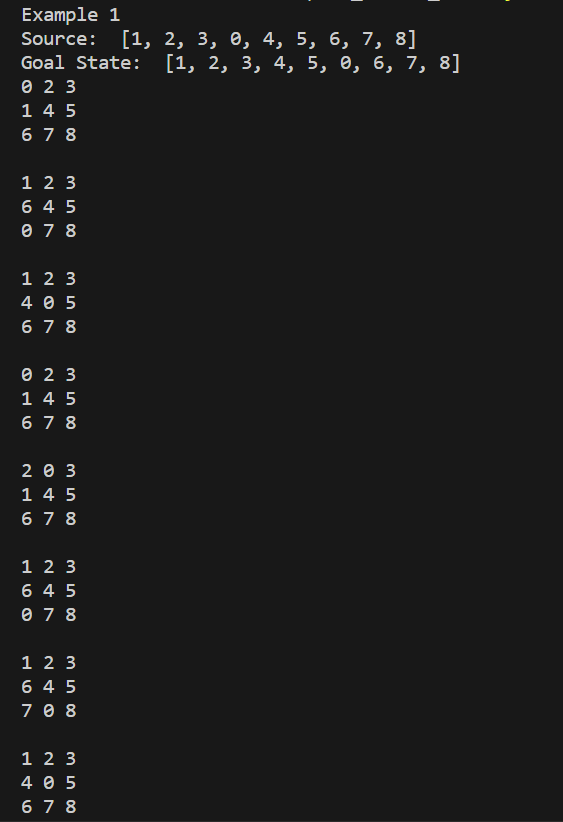
src = [1,2,3,0,4,5,6,7,8]

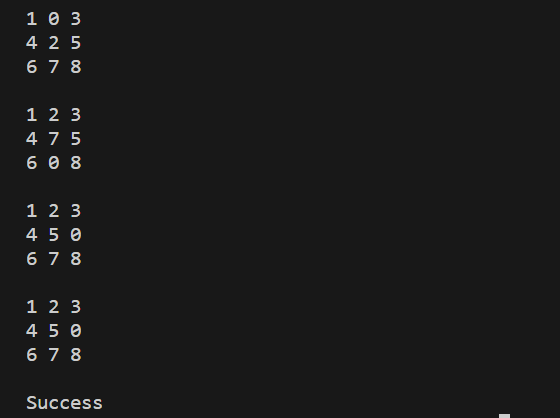
target = [1,2,3,4,5,0,6,7,8]

print("Source: " , src) print("Goal State: " , target)

iterative\_deepening\_search(src, target)

# OUTPUT:





## 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

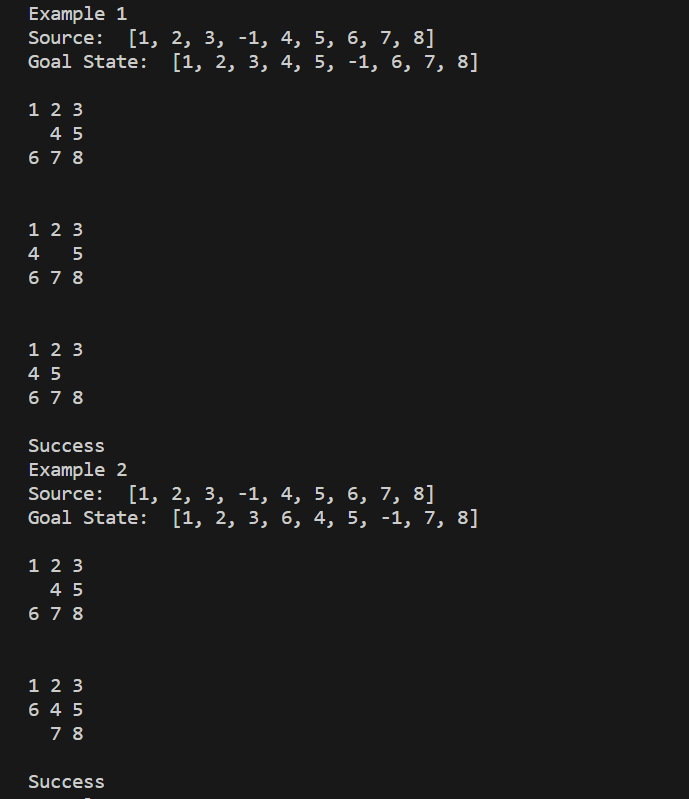
print("Example 3")

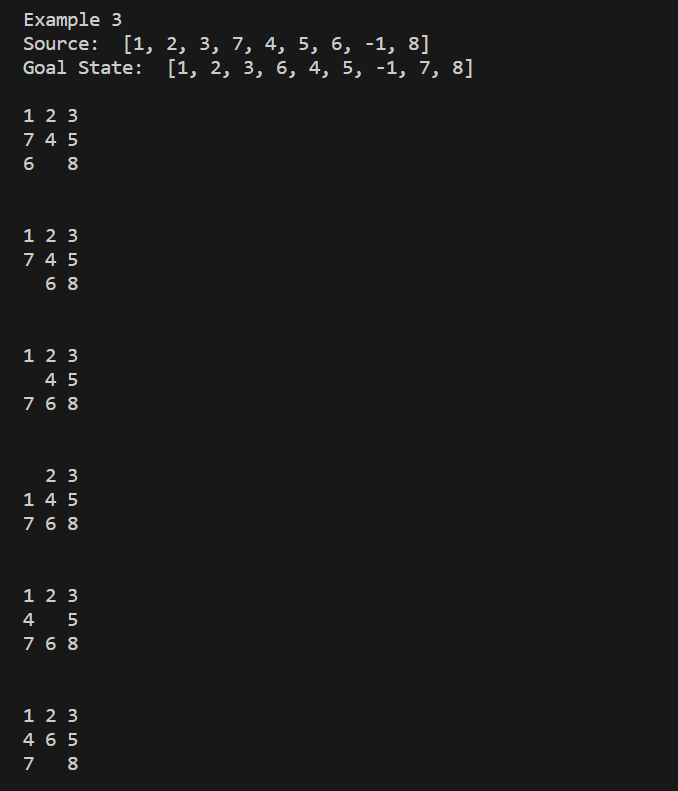
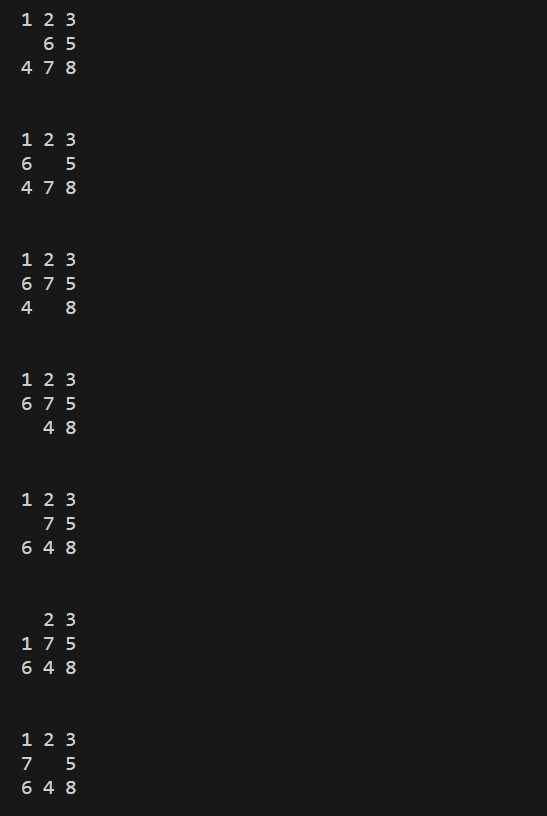
src = [1,2,3,7,4,5,6,-1,8]

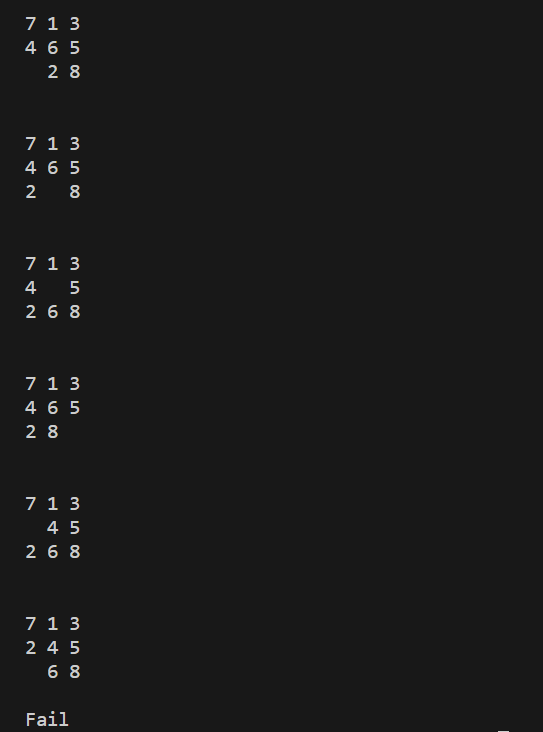
target=[1,2,3,6,4,5,-1,7,8]

print("Source: " , src) print("Goal State: " , target) astar(src, target)

# OUTPUT:





## 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],

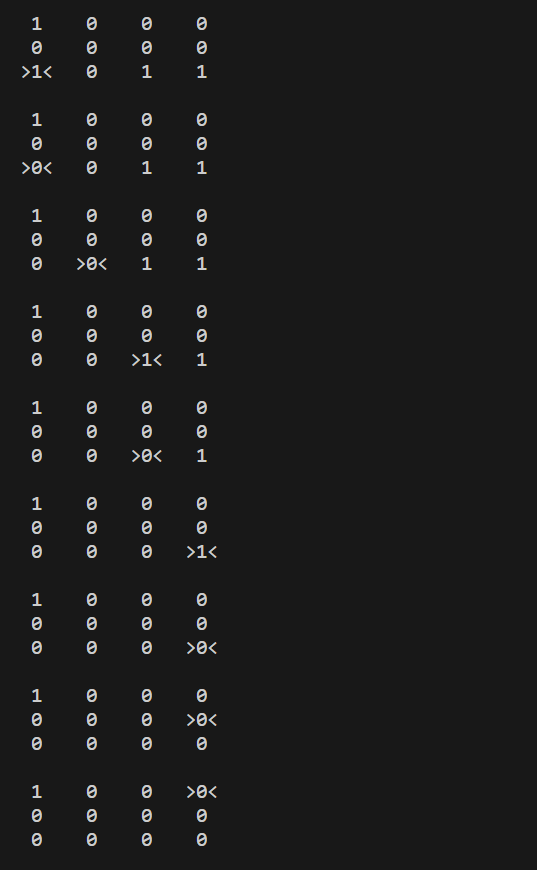
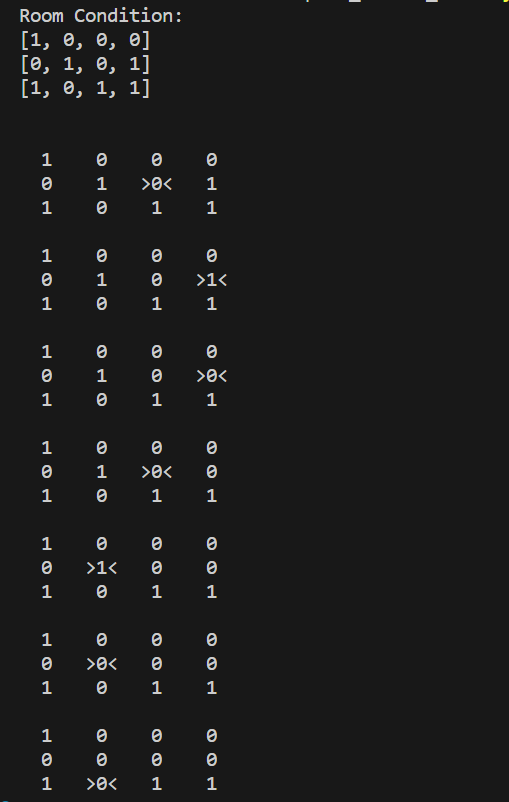
[0, 1, 0, 1],

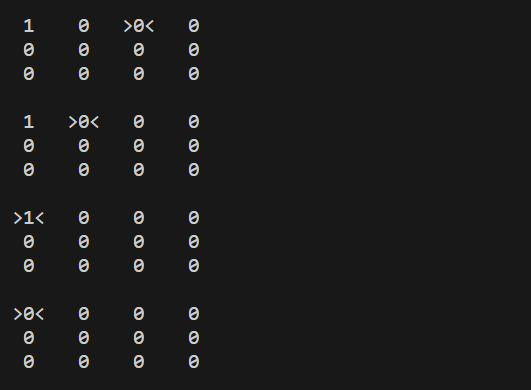
[1, 0, 1, 1]]

print("Room Condition: ") for row in floor:

print(row) print("\n") clean(floor, 1, 2)

# OUTPUT:





## 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()

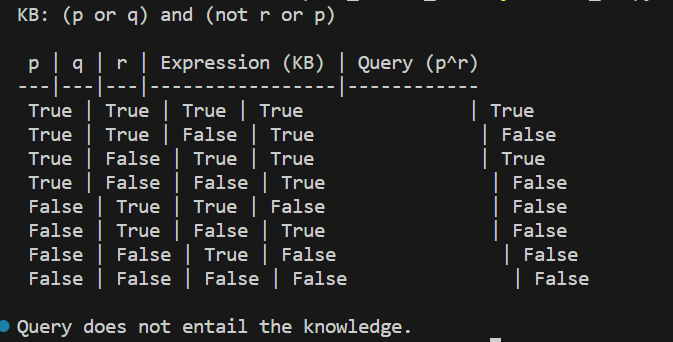
if query\_entails\_knowledge(): print("\nQuery entails the knowledge.")

else:

print("\nQuery does not entail the knowledge.")

if name == " main ": main()

# OUTPUT:



## 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:

turn null. \

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

\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'

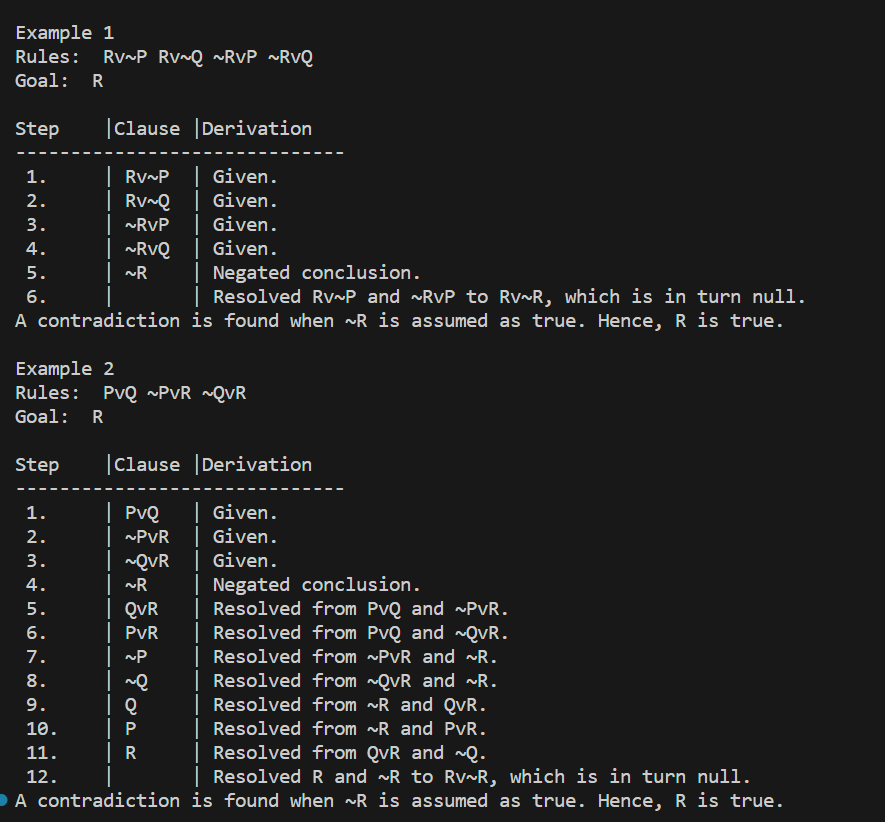
print('Rules: ',rules) print("Goal: ",goal) main(rules, goal)

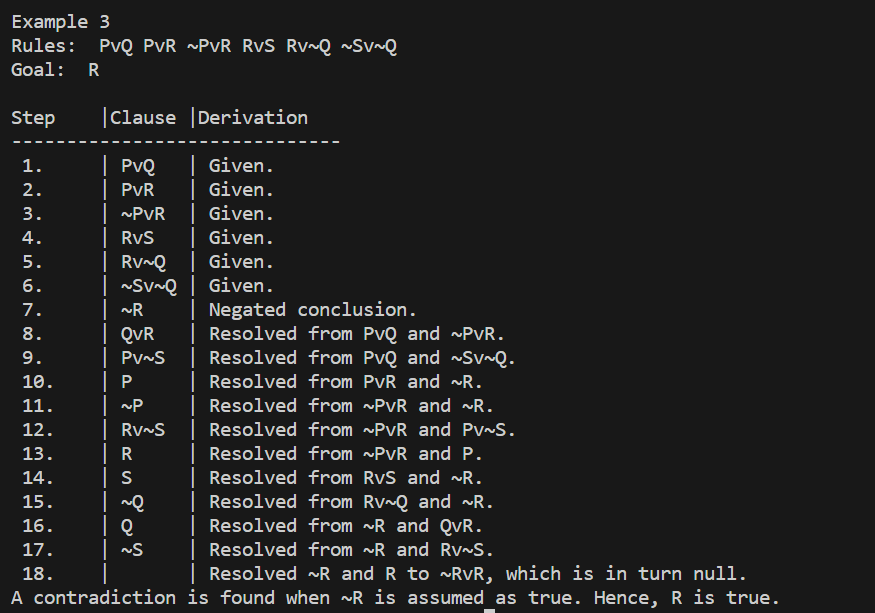
rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)

goal = 'R' print('Rules: ',rules)

print("Goal: ",goal) main(rules, goal)

# OUTPUT:





## 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)

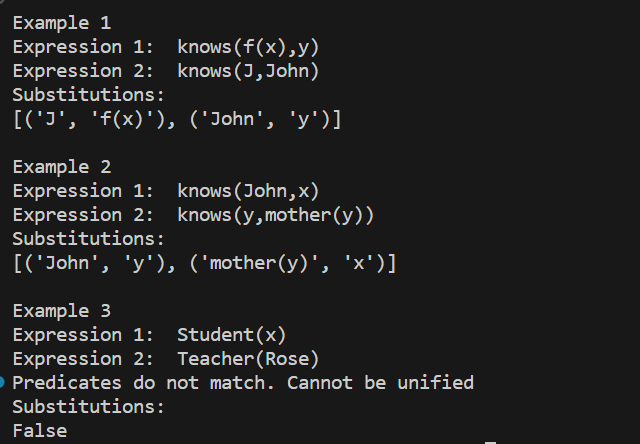
print("\nExample 3") exp1 = "Student(x)" exp2 = "Teacher(Rose)"

print("Expression 1: ",exp1)

print("Expression 2: ",exp2)

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

# OUTPUT:



## 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('-')

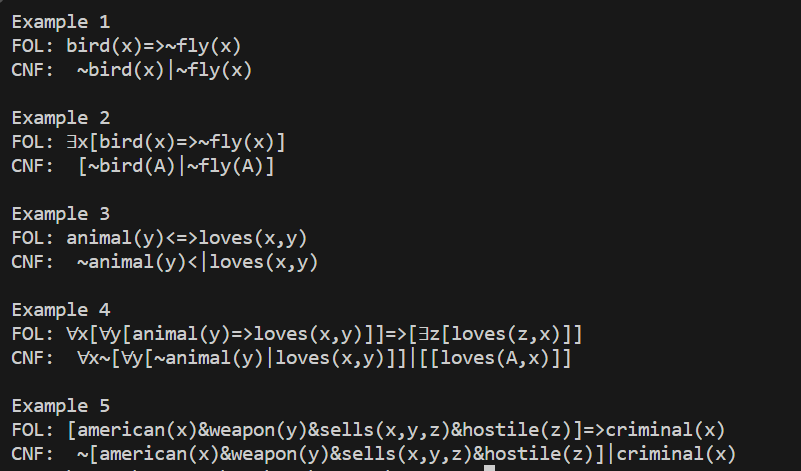
br = statement.index('[') if '[' in statement else 0 new\_statement = '~' + statement[br:i] + '|' + statement[i+1:]

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:



## 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:**

