# B.M.S. COLLEGE OF ENGINEERING BENGALURU

Autonomous Institute, Affiliated to VTU

Lab Record

# Artificial Intelligence

*Submitted in partial fulfillment for the 5th Semester Laboratory*

Bachelor of Technology in

Computer Science and Engineering

*Submitted by:*

# Mohammad Adnan Khan

1BM21CS107

Department of Computer Science and Engineering

B.M.S. College of Engineering

Bull Temple Road, Basavanagudi, Bangalore 560 019 Nov-Mar 2024

**B.M.S. COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



***CERTIFICATE***

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Mohammad Adnan Khan (1BM21CS107) during the 5th Semester Nov-March- 2024.

Signature of the Faculty Incharge:

Prof. Sneha S Bagalkot

Assistant Professor

Department of Computer Science and Engineering

B.M.S. College of Engineering, Bangalore

# Table of Contents

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Title** | **Page No.** |
| 1. | Tic Tac Toe | 1 – 5 |
| 2. | 8 Puzzle Breadth First Search Algorithm | 6-8 |
| 3. | 8 Puzzle Iterative Deepening Search Algorithm | 9-11 |
| 4. | 8 Puzzle A\* Search Algorithm | 12-16 |
| 5. | Vacuum Cleaner | 17-19 |
| 6. | Knowledge Base Entailment | 20-22 |
| 7. | Knowledge Base Resolution | 23-26 |
| 8. | Unification | 27-29 |
| 9. | FOL to CNF | 30-31 |
| 10. | Forward reasoning | 32-34 |

1. **Tic-Tac-Toe**

**Code:**

# Create a 3x3 tic tac toe board of "" strings for each value

board = [' '] \* 9

# Create a function to display your board

def display\_board(board):

print(f" {board[0]} | {board[1]} | {board[2]} ")

print("---+---+---")

print(f" {board[3]} | {board[4]} | {board[5]} ")

print("---+---+---")

print(f" {board[6]} | {board[7]} | {board[8]} ")

#Create a function to check if anyone won, Use marks "X" or "O"

def check\_win(player\_mark, board):

win = [f'{player\_mark}'] \* 3

return board[:3] == win or board[3:6] == win or board[6:9] == win or \

[board[0], board[4], board[8]] == win or [board[2], board[4], board[6]] == win or \

[board[0], board[3], board[6]] == win or [board[1], board[4], board[7]] == win or [board[2], board[5], board[8]] == win

def check\_draw(board):

return ' ' not in board

# Create a Function that makes a copy of the board

def board\_copy(board):

new\_board = []

for c in board:

new\_board += c

return new\_board

def test\_win\_move(move, player\_mark, board):

copy = board\_copy(board)

copy[move] = player\_mark

return check\_win(player\_mark, copy)

def win\_strategy(board):

if board[4] == ' ':

return 4

for i in [0, 2, 6, 8]:

if board[i] == ' ':

return i

for i in [1, 3, 5, 7]:

if board[i] == ' ':

return i

def get\_agent\_move(board):

for i in range(9):

if board[i] == ' ' and test\_win\_move(i, 'X', board):

return i

for i in range(9):

if board[i] == ' ' and test\_win\_move(i, 'O', board):

return i

return win\_strategy(board)

def tictactoe():

playing = True

while playing:

in\_game = True

board = [' '] \* 9

print('Would you like to go first or second? (1/2)')

choice = input()

player\_marker = 'O' if choice == '1' else 'X'

display\_board(board)

while in\_game:

print('\n')

if player\_marker == 'O':

print('Player move: (0-8)')

move = int(input())

if board[move] != ' ':

print('Invalid move')

continue

else:

move = get\_agent\_move(board)

board[move] = player\_marker

if check\_win(player\_marker,board):

in\_game = False

display\_board(board)

if player\_marker == 'O':

print('O won')

else:

print('X won')

break

if check\_draw(board):

in\_game = False

display\_board(board)

print('The game was a draw.')

break

display\_board(board)

if player\_marker == 'O':

player\_marker = 'X'

else:

player\_marker = 'O'

print('Continue playing? (y/n)')

ans = input()

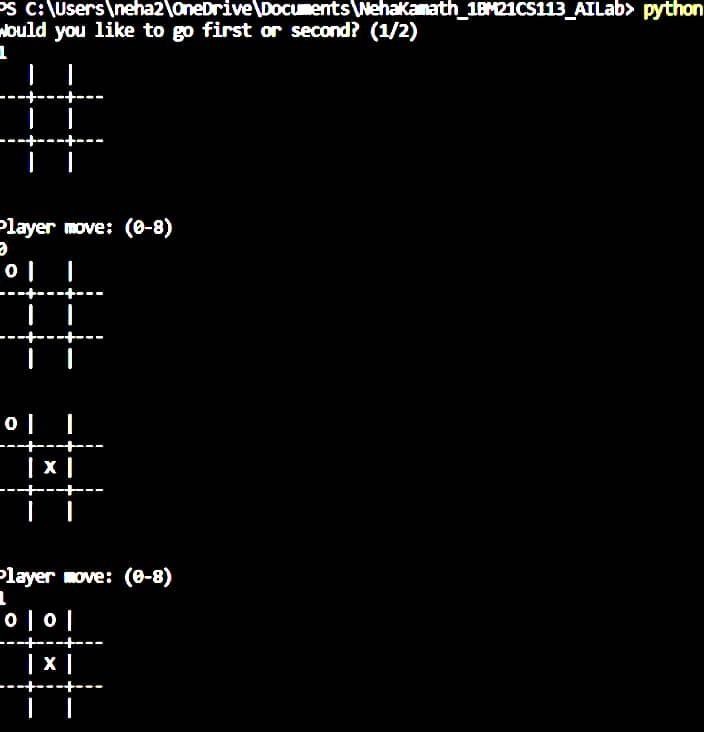
if ans not in 'yY':

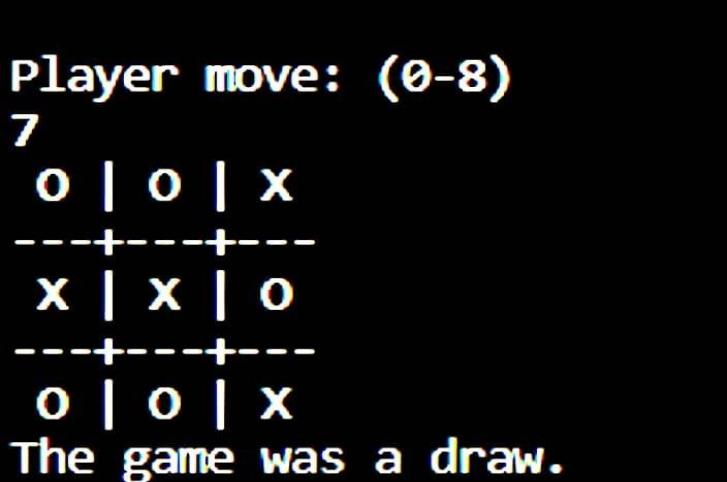
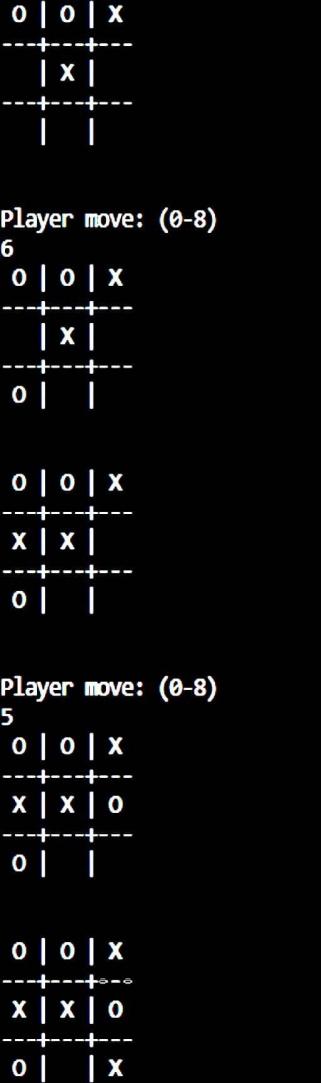
playing = False

# Play!!!

tictactoe()

**Output:**





1. **8 Puzzle Breadth First Search Algorithm**

**Code:**

import numpy as np

import pandas as pd

import os

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 # Return the modified state

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 bfs(src, target):

queue = []

queue.append(src)

exp = []

while len(queue) > 0:

source = queue.pop(0)

exp.append(source)

print(source[0],'|',source[1],'|',source[2])

print(source[3],'|',source[4],'|', source[5])

print(source[6],'|', source[7],'|',source[8])

print()

if source == target:

print("success")

return

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:

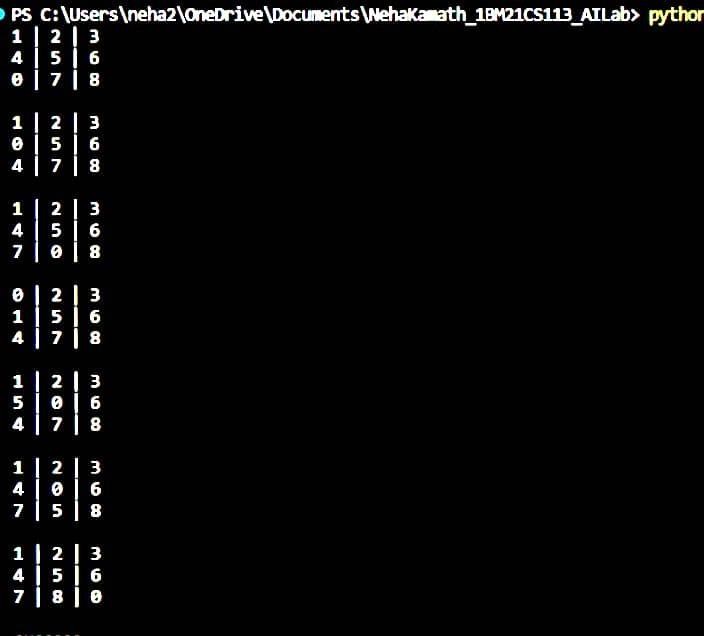
queue.append(move)

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

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

bfs(src, target)

**Output:**



1. **8 Puzzle Iterative Deepening Search Algorithm**

**Code:**

def id\_dfs(puzzle, goal, get\_moves):

import itertools

#get\_moves -> possible\_moves

def dfs(route, depth):

if depth == 0:

return

if route[-1] == goal:

return route

for move in get\_moves(route[-1]):

if move not in route:

next\_route = dfs(route + [move], depth - 1)

if next\_route:

return next\_route

for depth in itertools.count():

route = dfs([puzzle], depth)

if route:

return route

def possible\_moves(state):

b = state.index(0) # ) indicates White space -> so b has index of it.

d = [] # direction

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 = []

for i in d:

pos\_moves.append(generate(state, i, b))

return pos\_moves

def generate(state, m, b):

temp = state.copy()

if m == 'd':

temp[b + 3], temp[b] = temp[b], temp[b + 3]

if m == 'u':

temp[b - 3], temp[b] = temp[b], temp[b - 3]

if m == 'l':

temp[b - 1], temp[b] = temp[b], temp[b - 1]

if m == 'r':

temp[b + 1], temp[b] = temp[b], temp[b + 1]

return temp

# calling ID-DFS

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

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

route = id\_dfs(initial, goal, possible\_moves)

if route:

print("Success!! It is possible to solve 8 Puzzle problem")

print("Path:", route)

else:

print("Failed to find a solution")

**Output:**



1. **8 Puzzle A\* Search Algorithm**

**Code:**

class Node:

def \_\_init\_\_(self,data,level,fval):

""" Initialize the node with the data, level of the node and the calculated fvalue """

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

""" Generate child nodes from the given node by moving the blank space

either in the four directions {up,down,left,right} """

x,y = self.find(self.data,'\_')

""" val\_list contains position values for moving the blank space in either of

the 4 directions [up,down,left,right] respectively. """

val\_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]

children = []

for i in val\_list:

child = self.shuffle(self.data,x,y,i[0],i[1])

if child is not None:

child\_node = Node(child,self.level+1,0)

children.append(child\_node)

return children

def shuffle(self,puz,x1,y1,x2,y2):

""" Move the blank space in the given direction and if the position value are out

of limits the return None """

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp\_puz = []

temp\_puz = self.copy(puz)

temp = temp\_puz[x2][y2]

temp\_puz[x2][y2] = temp\_puz[x1][y1]

temp\_puz[x1][y1] = temp

return temp\_puz

else:

return None

def copy(self,root):

""" Copy function to create a similar matrix of the given node"""

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self,puz,x):

""" Specifically used to find the position of the blank space """

for i in range(0,len(self.data)):

for j in range(0,len(self.data)):

if puz[i][j] == x:

return i,j

class Puzzle:

def \_\_init\_\_(self,size):

""" Initialize the puzzle size by the specified size,open and closed lists to empty """

self.n = size

self.open = []

self.closed = []

def accept(self):

""" Accepts the puzzle from the user """

puz = []

for i in range(0,self.n):

temp = input().split(" ")

puz.append(temp)

return puz

def f(self,start,goal):

""" Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """

return self.h(start.data,goal)+start.level

def h(self,start,goal):

""" Calculates the different between the given puzzles """

temp = 0

for i in range(0,self.n):

for j in range(0,self.n):

if start[i][j] != goal[i][j] and start[i][j] != '\_':

temp += 1

return temp

def process(self):

""" Accept Start and Goal Puzzle state"""

print("Enter the start state matrix \n")

start = self.accept()

print("Enter the goal state matrix \n")

goal = self.accept()

start = Node(start,0,0)

start.fval = self.f(start,goal)

""" Put the start node in the open list"""

self.open.append(start)

print("\n\n")

while True:

cur = self.open[0]

print("")

print(" | ")

print(" | ")

print(" \\'/ \n")

for i in cur.data:

for j in i:

print(j,end=" ")

print("")

""" If the difference between current and goal node is 0 we have reached the goal node"""

if(self.h(cur.data,goal) == 0):

break

for i in cur.generate\_child():

i.fval = self.f(i,goal)

self.open.append(i)

self.closed.append(cur)

del self.open[0]

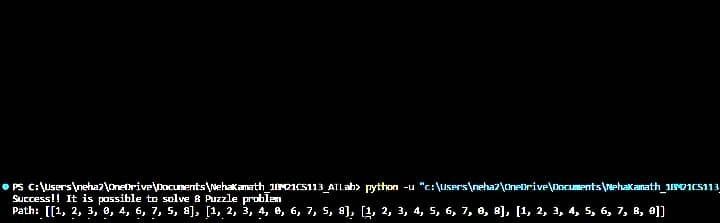
""" sort the opne list based on f value """

self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)

puz.process()

**Output:**



1. **Vacuum Cleaner**

**Code:**

**For 2 rooms:**

def clean\_room(room\_name, is\_dirty):

if is\_dirty:

print(f"Cleaning {room\_name} (Room was dirty)")

print(f"{room\_name} is now clean.")

return 0 # Updated status after cleaning

else:

print(f"{room\_name} is already clean.")

return 0 # Status remains clean

def main():

rooms = ["Room 1", "Room 2"]

room\_statuses = []

for room in rooms:

status = int(input(f"Enter clean status for {room} (1 for dirty, 0 for clean): "))

room\_statuses.append((room, status))

print(room\_statuses)

for i, (room, status) in enumerate(room\_statuses):

room\_statuses[i] = (room,clean\_room(room, status)) # Update status after cleaning

print(f"Returning to {rooms[0]} to check if it has become dirty again:")

room\_statuses[0]= (rooms[0],clean\_room(rooms[0], room\_statuses[0][1])) # Checking Room 1 after cleaning all rooms

print(f"{rooms[0]} is {'dirty' if room\_statuses[0][1] else 'clean'} after checking.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**For 4 rooms :**

**Code:**

def clean\_room(floor, room\_row, room\_col):

if floor[room\_row][room\_col] == 1:

print(f"Cleaning Room at ({room\_row + 1}, {room\_col + 1}) (Room was dirty)")

floor[room\_row][room\_col] = 0

print("Room is now clean.")

else:

print(f"Room at ({room\_row + 1}, {room\_col + 1}) is already clean.")

def main():

rows = 2

cols = 2

floor = [[0, 0], [0, 0]] # Initialize a 2x2 floor with clean rooms

for i in range(rows):

for j in range(cols):

status = int(input(f"Enter clean status for Room at ({i + 1}, {j + 1}) (1 for dirty, 0 for clean): "))

floor[i][j] = status

for i in range(rows):

for j in range(cols):

clean\_room(floor, i, j)

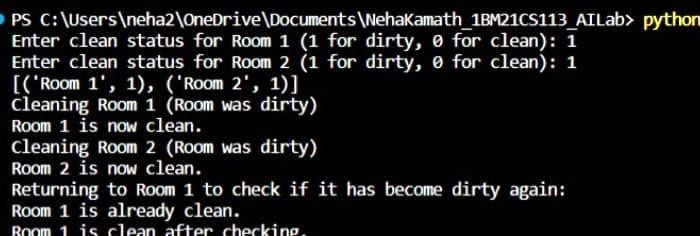
print("Returning to Room at (1, 1) to check if it has become dirty again:")

clean\_room(floor, 0, 0) # Checking Room at (1, 1) after cleaning all rooms

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**



1. **Knowledge Base Entailment**

**Code:**

from sympy import symbols, And, Not, Implies, satisfiable

def create\_knowledge\_base():

# Define propositional symbols

p = symbols('p')

q = symbols('q')

r = symbols('r')

a

# Define knowledge base using logical statements

knowledge\_base = And(

Implies(p, q), # If p then q

Implies(q, r), # If q then r

Not(r) # Not r

)

return knowledge\_base

def query\_entails(knowledge\_base, query):

# Check if the knowledge base entails the query

entailment = satisfiable(And(knowledge\_base, Not(query)))

# If there is no satisfying assignment, then the query is entailed

return not entailment

if \_\_name\_\_ == "\_\_main\_\_":

# Create the knowledge base

kb = create\_knowledge\_base()

# Define a query

query = symbols('p')

# Check if the query entails the knowledge base

result = query\_entails(kb, query)

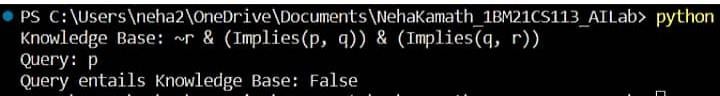
# Display the results

print("Knowledge Base:", kb)

print("Query:", query)

print("Query entails Knowledge Base:", result)

**Output:**



1. **Knowledge Base Resolution**

**Code:**

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

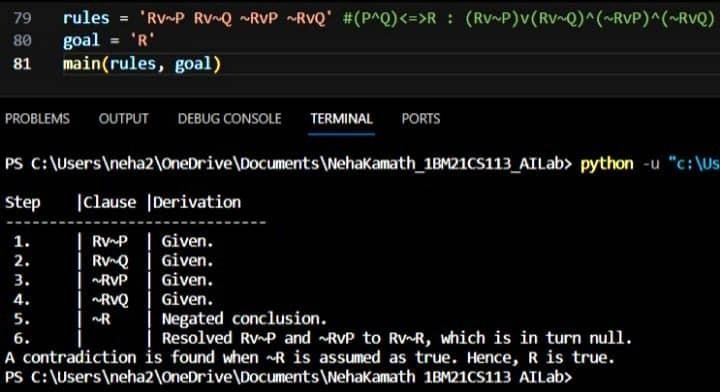
return steps

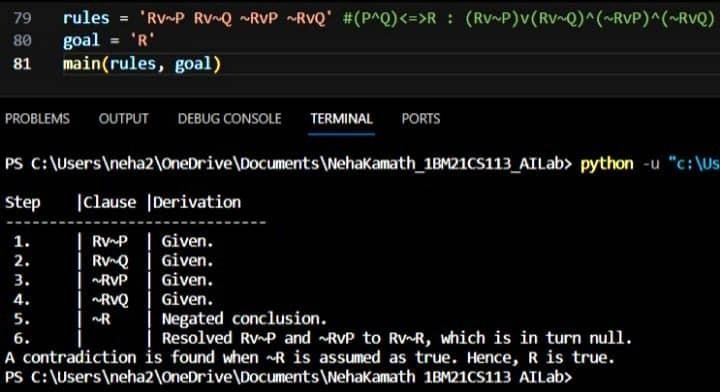
rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)

goal = 'R'

main(rules, goal)

**Output:**





1. **Unification**

**Code:**

def unify\_var(var, x, theta):

"""

Helper function for unifying a variable with a term.

"""

if var in theta:

return unify(theta[var], x, theta)

elif x in theta:

return unify(var, theta[x], theta)

else:

theta[var] = x

return theta

def unify(x, y, theta={}):

"""

Unify two expressions x and y with the given substitution theta.

"""

if theta is None:

return None

elif x == y:

return theta

elif isinstance(x, str) and x[0].islower():

return unify\_var(x, y, theta)

elif isinstance(y, str) and y[0].islower():

return unify\_var(y, x, theta)

elif isinstance(x, list) and isinstance(y, list):

if len(x) != len(y):

return None

for xi, yi in zip(x, y):

theta = unify(xi, yi, theta)

if theta is None:

return None

return theta

else:

return None

# Example usage:

x = ['P', 'a', 'x']

y = ['P', 'y', 'z']

result = unify(x, y)

print(result)

# Sample input

expression1 = ['P', 'a', 'x']

expression2 = ['P', 'y', 'z']

# Unify the expressions

result = unify(expression1, expression2)

# Display the result

print("Input:")

print("Expression 1:", expression1)

print("Expression 2:", expression2)

print("\nOutput:")

if result is not None:

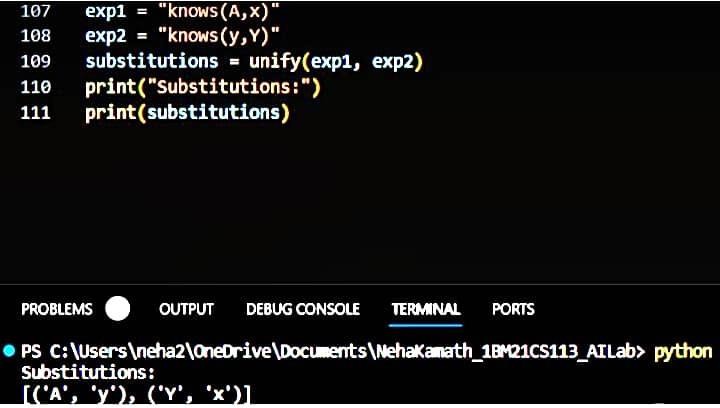
print("Unification Successful!")

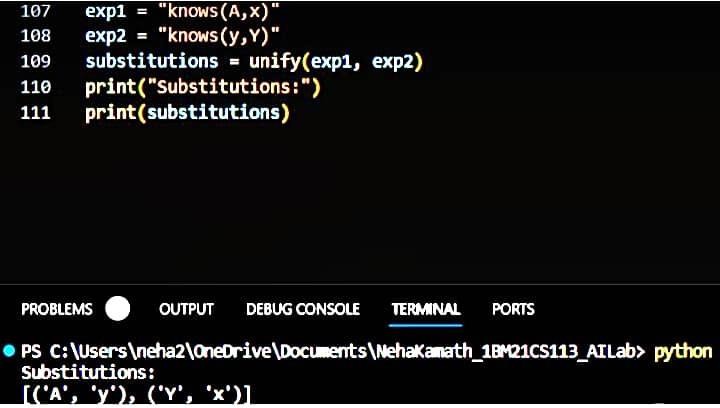
print("Substitution theta:", result)

else:

print("Unification Failed.")

**Output:**





1. **FOL to CNF**

**Code:**

from sympy import symbols, to\_cnf, parse\_expr

def convert\_to\_cnf(logic\_statement):

# Parse the logic statement

parsed\_statement = parse\_expr(logic\_statement)

# Convert to CNF

cnf = to\_cnf(parsed\_statement)

return cnf

if \_\_name\_\_ == "\_\_main\_\_":

# Example: (A & B) | (~C & D)

logic\_statement = "(A & B) | (~C & D)"

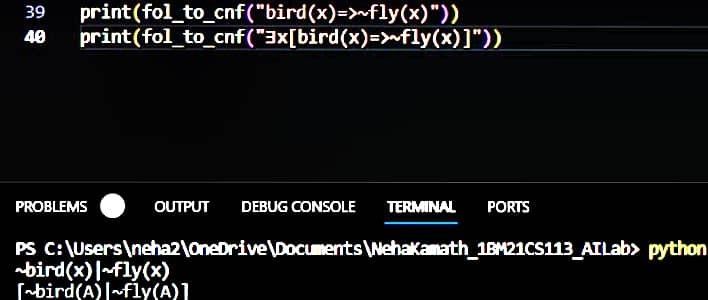
# Convert to CNF

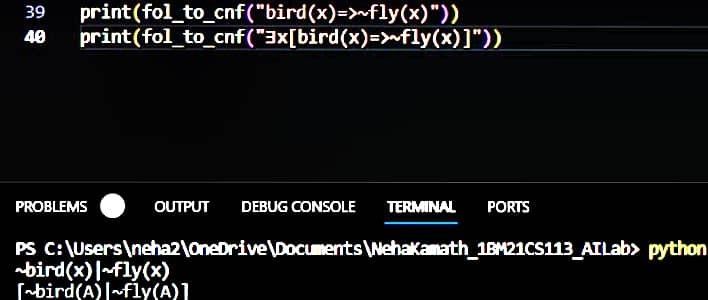
cnf\_result = convert\_to\_cnf(logic\_statement)

print("Original Statement:", logic\_statement)

print("CNF Form:", cnf\_result)

**Output:**





1. **Forward reasoning**

**Code:**

from sympy import symbols, Eq, And, Or, Implies, ask, satisfiable

# Define individuals (family members)

John, Mary, Alice, Bob = symbols('John Mary Alice Bob')

# Define predicates

Parent = symbols('Parent')

Grandparent = symbols('Grandparent')

# Define knowledge base

knowledge\_base = [

Eq(Parent(John, Alice), True),

Eq(Parent(Mary, Alice), True),

Eq(Parent(Alice, Bob), True),

Implies(Parent(x, y), Grandparent(x, y)),

]

# Define query

query = Grandparent(John, Bob)

# Perform forward reasoning

def forward\_reasoning(knowledge\_base, query):

new\_facts = set()

while True:

for fact in knowledge\_base:

if ask(fact):

continue

if satisfiable(fact):

new\_facts.add(fact)

if not new\_facts:

break

knowledge\_base.extend(new\_facts)

return ask(query)

# Check if the query is true based on the knowledge base

result = forward\_reasoning(knowledge\_base, query)

# Print the result

print("Query:", query)

print("Result:", result)

**Output:**

