**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.

****

**LAB REPORT**

**on**

**ARTIFICIAL INTELLIGENCE**

*Submitted by*

**NEHA BHASKAR KAMATH(1BM21CS113)**

***Under the Guidance of***

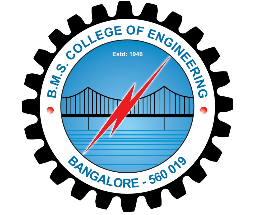
|  |  |
| --- | --- |
| **Prof. Sneha S Bagalkot**  **Assistant Professor, BMSCE** |  |

*in partial fulfilment 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**

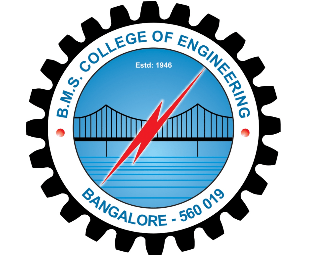
**November 2023-February 2024**

**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 **Neha Bhaskar Kamath(1BM21CS113)** , who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24.

The Lab report has been approved as it satisfies the academic requirements in respect of **Artificial Intelligence- (22CS5PCAIN)** work prescribed for the said degree.

Prof. Sneha S Bagalkot       Dr. Jyothi Nayak

Assistant professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**B. M. S. College of EngineerinG**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

****

***DECLARATION***

I, Neha Bhaskar Kamath (1BM21CS113), student of 5th Semester, B.E, Department of Computer Science and Engineering, B. M. S. College of Engineering, Bangalore, here by declare that, this lab report entitled " **Artificial Intelligence**" has been carried out by me under the guidance of **Prof. Sneha S Bagalkot**, Assistant Professor, Department of CSE, B. M. S. College of Engineering, Bangalore during the academic semester November-2023-February-2024.

I also declare that to the best of my knowledge and belief, the development reported here is not from part of any other report by any other students.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Title** | **Page No.** |
| 1 | Implementation of Tic Tac Toe | 5-9 |
| 2 | 8 Puzzle Breadth First Search Algorithm | 10-12 |
| 3 | 8 Puzzle Iterative Deepening Search Algorithm | 13-15 |
| 4 | 8 Puzzle A\* Search Algorithm | 16-20 |
| 5 | Vacuum Cleaner | 21-22 |
| 6 | Knowledge Base Entailment | 23-24 |
| 7 | Knowledge Base Resolution | 25-28 |
| 8 | Unification | 29-33 |
| 9 | FOL to CNF | 34-35 |
| 10 | Forward reasoning | 36-39 |

**Program 1**

**Implementation of 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()

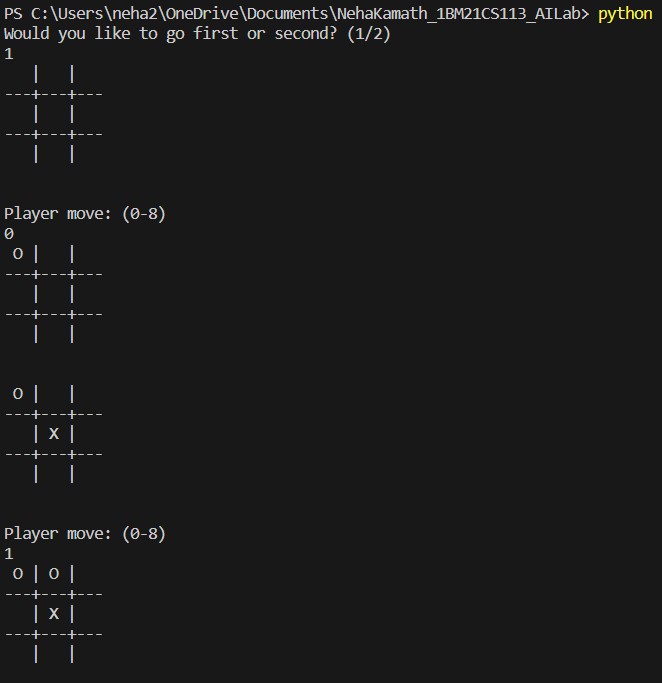
if ans not in 'yY':

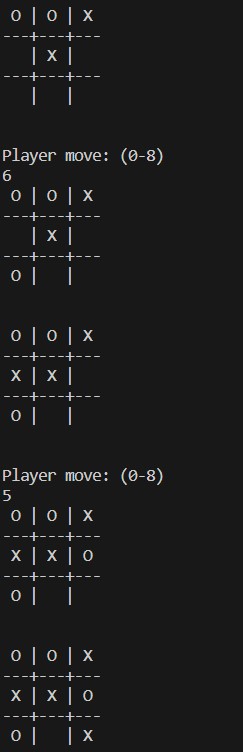
playing = False

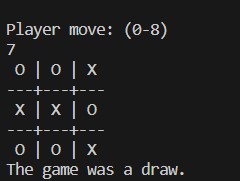
# Play!!!

tictactoe()

**Output**







**Program 2**

**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:

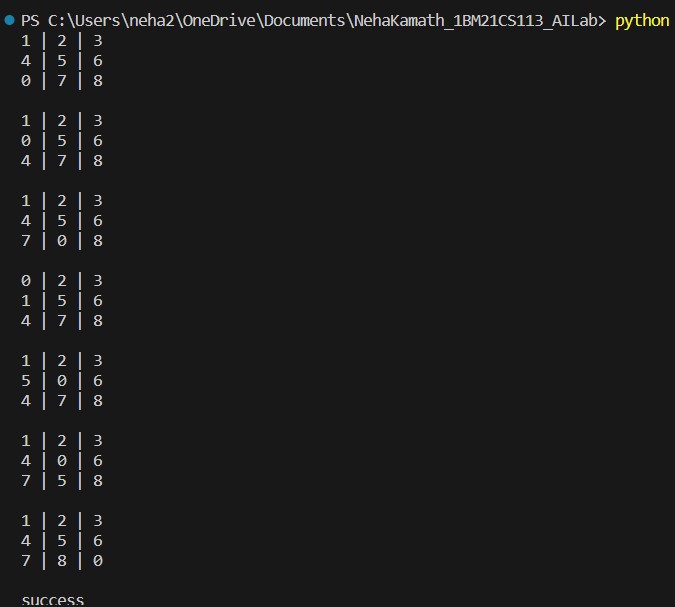
queue.append(move)

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

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

bfs(src, target)

**Output**



**Program 3**

**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:

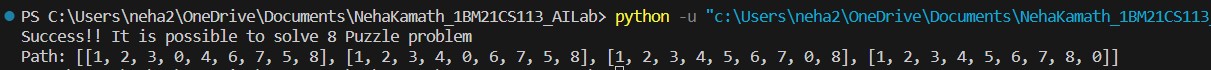
print("Success!! It is possible to solve 8 Puzzle problem")

print("Path:", route)

else:

print("Failed to find a solution")

**Output**



**Program 4**

**8 Puzzle A\* 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]

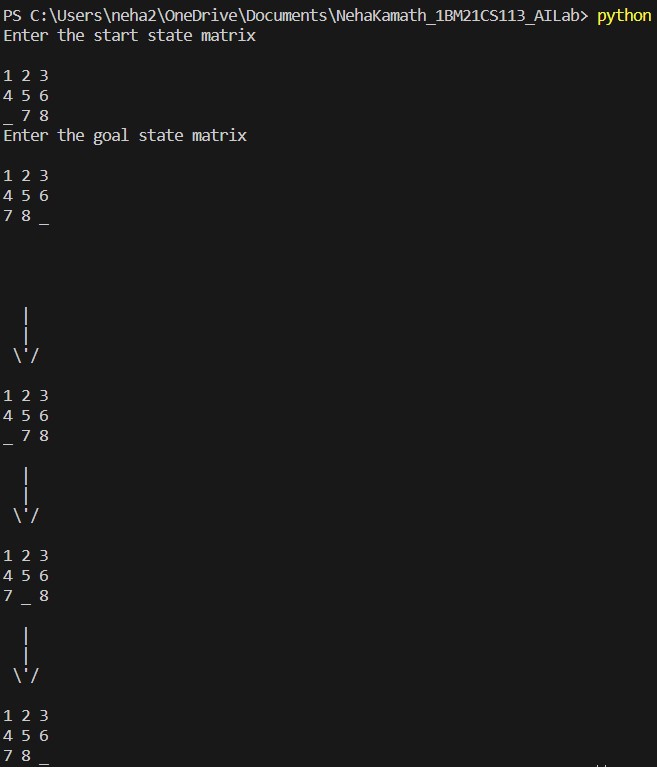
""" sort the opne list based on f value """

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

puz = Puzzle(3)

puz.process()

**Output**



**Program 5**

**Vacuum Cleaner**

**Code**

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:")

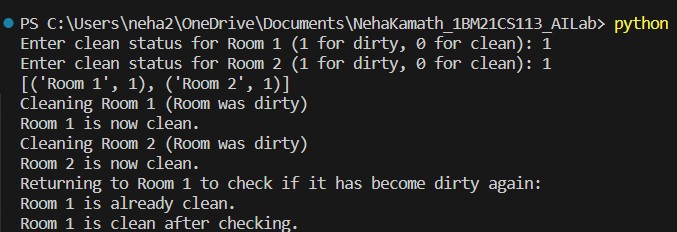
room\_statuses[0]= (rooms[0],clean\_room(rooms[0], room\_statuses[0][1])) # Checking Room 1 after cleaning all rooms

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

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

main()

**Output**



**Program 6**

**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')

# 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)

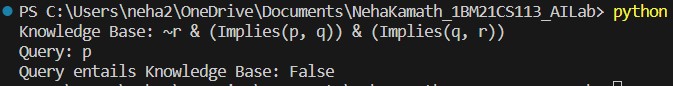
# Display the results

print("Knowledge Base:", kb)

print("Query:", query)

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

**Output**



**Program 7**

**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

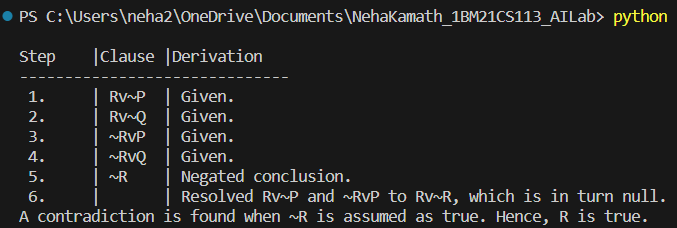
return steps

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

goal = 'R'

main(rules, goal)

**Output**

****

**Program 8**

**Unification**

**Code**

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

exp1 = "knows(A,x)"

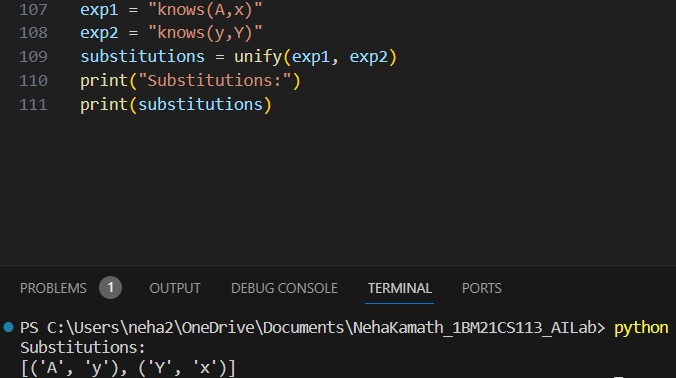
exp2 = "knows(y,Y)"

substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)

**Output**



**Program 9**

**FOL to CNF**

**Code**

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:]

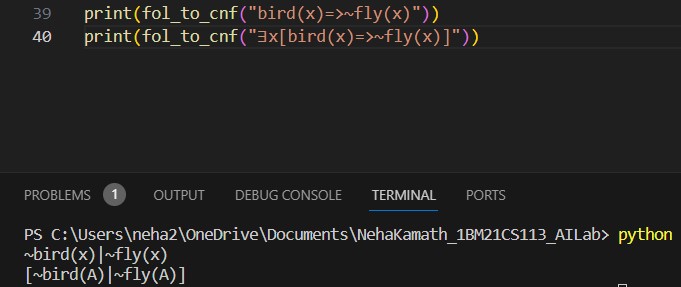
statement = statement[:br] + new\_statement if br > 0 else new\_statement

return Skolemization(statement)

print(fol\_to\_cnf("bird(x)=>~fly(x)"))

print(fol\_to\_cnf("∃x[bird(x)=>~fly(x)]"))

**Output**



**Program 10**

**Forward Chaining**

**Code**

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()

**Output**

