Artificial Intelligence

Devarakonda Venkata Ravi Sridhar

RA1811003010732

CSE – D2

# Experiment-1

# Tower of Hanoi

## Aim:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e., a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

## Algorithm:

Take an example for 2 disks:

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1: Shift first disk from 'A' to 'B'.

Step 2: Shift second disk from 'A' to 'C'.

Step 3: Shift first disk from 'B' to 'C'.

The pattern here is:

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.

## Code:

*def* *tower\_of\_hanoi*(disks, source, auxiliary, target):

*if*(disks *==* 1):

        print(('Move disk 1 from rod {} to rod {}').format(source, target))

*return*

    tower\_of\_hanoi(disks *-* 1, source, target, auxiliary)

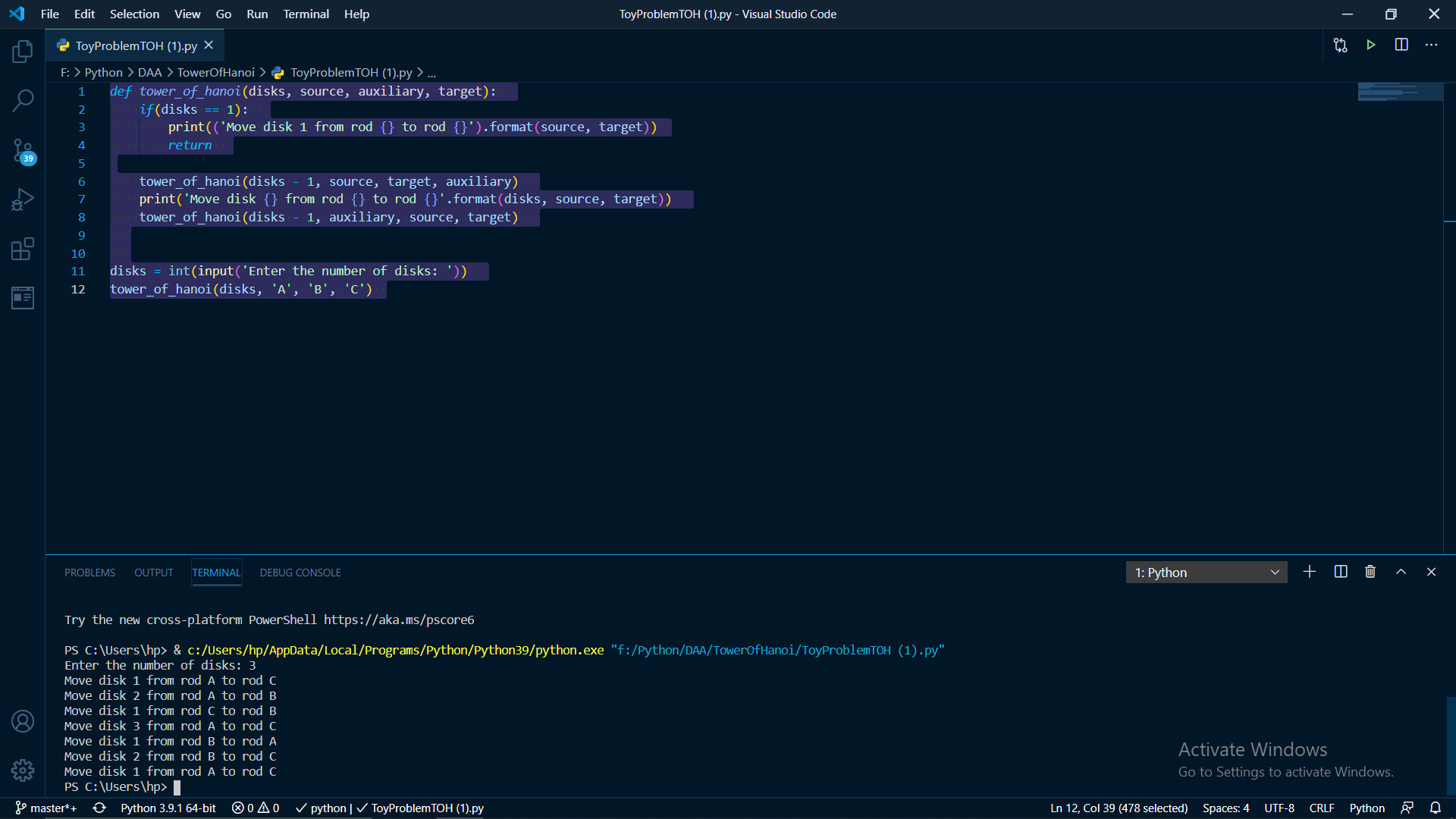
    print('Move disk {} from rod {} to rod {}'.format(disks, source, target))

    tower\_of\_hanoi(disks *-* 1, auxiliary, source, target)

disks *=* int(input('Enter the number of disks: '))

tower\_of\_hanoi(disks, 'A', 'B', 'C')

## Output:



# Experiment-2

# Camel and Banana Puzzle

## Aim:

A person has 3000 bananas and a camel. The person wants to transport the maximum number of bananas to a destination which is 1000 KMs away, using only the camel as a mode of transportation. The camel cannot carry more than 1000 bananas at a time and eats a banana every km it travels. What is the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed).

## Algorithm:

1. **3000 – 5x = 2000** so we get **x = 200**
2. **2000-3y = 1000** so we get **y = 333.33** but here the distance is also the number of bananas and it cannot be fraction so we take y =333 and at**IP2** we have the number of bananas equal **1001,**so its**2000-3y = 1001**
3. **So,**the remaining distance to the market is**1000 -x-y =z**i.e.**1000-200-333 => z =467.**

So, from **IP2 to the destination** point camel eats **467 bananas.**The remaining**bananas are 533.**

**So,**the **maximum number of bananas that can be transferred is 533.**

## Source Code:

total=int(input('No. of Bananas'))  
distance=int(input('Distance'))  
load\_capacity=int(input('Max Load Capacity'))  
lose=0  
start=total  
for i in range(distance):  
 while start>0:  
 start=start-load\_capacity  
  
 if start==1:  
 lose=lose-1  
 lose=lose+2  
  
 lose=lose-2  
 start=total-lose  
 if start==0:  
 break  
print(start)

## Output:

# Experiment 3

# Graph Coloring Problem

## Aim:

[Graph coloring](http://en.wikipedia.org/wiki/Graph_coloring) problem is to assign colors to certain elements of a graph subject to certain constraints.

## Algorithm:

**1.** Color first vertex with first color.  
**2.** Do following for remaining V-1 vertices.  
 **a)**Consider the currently picked vertex and color it with the  
lowest numbered color that has not been used on any previously  
colored vertices adjacent to it. If all previously used colors  
appear on vertices adjacent to v, assign a new color to it.

## Source Code:

#Exp-2 Graph Coloring

colors=['red','green','blue']

print('Number of colors used: ', len(colors))

vertices=['1','2','3','4','5']

adj={'1':['2','5'],'2':['1','3','5'],'3':['2','4','5'],'4':['3','5'],'5':['4','1','2','3']}

result={}

for vertex in vertices:

  adj\_color=[]

  neighbour=adj[vertex]

  for i in neighbour:

    if i in result.keys():

      adj\_color.append(result[i])

  for color in colors:

    if color not in adj\_color:

      break

  result[vertex]=color

for (vertex,color) in result.items():

  print("Color assigned to vertex",vertex,"adjacent to",adj[vertex],"is",color)

## Output:

Number of colors used: 3

Color assigned to vertex 1 adjacent to ['2', '5'] is red

Color assigned to vertex 2 adjacent to ['1', '3', '5'] is green

Color assigned to vertex 3 adjacent to ['2', '4', '5'] is red

Color assigned to vertex 4 adjacent to ['3', '5'] is green

Color assigned to vertex 5 adjacent to ['4', '1', '2', '3'] is blue

# Experiment 4

## Crypt Arithmetic Problem

## AIM:

## Cryptarithmetic Problem is a type of [constraint satisfaction problem](https://www.tutorialandexample.com/constraint-satisfaction-problems-in-artificial-intelligence/) where the game is about digits and its unique replacement either with alphabets or other symbols. In **cryptarithmetic problem,**the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

## ALGORITHM:

* There should be a unique digit to be replaced with a unique alphabet.
* The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4, nothing else.
* Digits should be from **0-9** only.
* There should be only one carry forward, while performing the addition operation on a problem.
* The problem can be solved from both sides, i.e., left-hand side (L.H.S), or righthand side (R.H.S)

## SOURCE CODE:

*import* itertools  
*def* solve2():  
 letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')  
 digits = range(10)  
 *for* perm *in* itertools.permutations(digits, len(letters)):  
 sol = dict(zip(letters, perm))  
 *if* sol['s'] == 0 *or* sol['m'] == 0:  
 *continue* send = 1000 \* sol['s'] + 100 \* sol['e'] + 10 \* sol['n'] + sol['d']  
 more = 1000 \* sol['m'] + 100 \* sol['o'] + 10 \* sol['r'] + sol['e']  
 money = 10000 \* sol['m'] + 1000 \* sol['o'] + 100 \* sol['n'] + 10 \* sol['e'] + sol['y']  
 *if* send + more == money:  
 print(" send"," more"," money")  
 *return* send, more, money  
print(solve2())

OUTPUT:  
C:\Users\hp\AppData\Local\Programs\Python\Python39\python.exe "F:/Python/DAA/AI Practical/ConstraintArithemeticProb.py"

send more money

(9567, 1085, 10652)

Process finished with exit code 0

# Experiment-5A

## BFS

## AIM:

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes.

## ALGORITM:

* **Step 1:** SET STATUS = 1 (ready state)  
  for each node in G
* **Step 2:** Enqueue the starting node A  
  and set its STATUS = 2  
  (waiting state)
* **Step 3:** Repeat Steps 4 and 5 until  
  QUEUE is empty
* **Step 4:** Dequeue a node N. Process it  
  and set its STATUS = 3  
  (processed state).
* **Step 5:** Enqueue all the neighbours of  
  N that are in the ready state  
  (whose STATUS = 1) and set  
  their STATUS = 2  
  (waiting state)  
  [END OF LOOP]
* **Step 6:**EXIT

## SOURCE CODE:

#BFS  
  
*import* time  
graph = {  
 'Chennai Central' : ['Potheri','Chennai Beach'],  
 'Chennai Beach' : ['Tambaram', 'Kattankulathur'],  
 'Potheri' : ['Chennai Central'],  
 'Kattankulathur' : [],  
 'Tambaram' : ['Chennai Central'],  
}  
  
visited = [] # List to keep track of visited nodes.  
queue = [] #Initialize a queue  
  
*def* bfs(*visited*, *graph*, *node*):  
 *visited*.append(*node*)  
 queue.append(*node*)  
  
 *while* queue:  
 s = queue.pop(0)  
 print (s, end = " ")  
  
 *for* neighbour *in graph*[s]:  
 *if* neighbour *not in visited*:  
 *visited*.append(neighbour)  
 queue.append(neighbour)  
  
# Driver Code  
t0=time.time()  
bfs(visited, graph, 'Chennai Central')  
t1=time.time() - t0  
print('\nTime taken for execution: {}'.format(t1))

OUTPUT:  
C:\Users\hp\AppData\Local\Programs\Python\Python39\python.exe "F:/Python/DAA/AI Practical/BFS.py"

Chennai Central Potheri Chennai Beach Tambaram Kattankulathur

Time taken for execution: 0.000003648

Process finished with exit code 0

# EXPERIMENT-5B

## DFS

## AIM:

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

## ALGORITHM:

* Step 1: SET STATUS = 1 (ready state) for each node in G
* Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
* Step 3: Repeat Steps 4 and 5 until STACK is empty
* Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
* Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their  
  STATUS = 2 (waiting state)  
  [END OF LOOP]
* Step 6: EXIT

SOURCE CODE:  
# DFS  
  
*import* time  
graph = {  
 'Sony' : ['Noise','Beats'],  
 'Beats' : ['Skullcandy', 'Boat'],  
 'Noise' : ['Sony'],  
 'Boat' : [],  
 'Skullcandy' : ['Sony'],  
}  
  
visited = set() # Set to keep track of visited nodes.  
  
*def* dfs(*visited*, *graph*, *node*):  
 *if node not in visited*:  
 print (*node*,end=' ')  
 *visited*.add(*node*)  
 *for* neighbour *in graph*[*node*]:  
 dfs(*visited*, *graph*, neighbour)  
  
# Driver Code  
t0=time.time()  
dfs(visited, graph, 'Sony')  
t1=time.time() - t0  
print('\nTime taken for execution: {}'.format(t1))

OUTPUT:  
C:\Users\hp\AppData\Local\Programs\Python\Python39\python.exe "F:/Python/DAA/AI Practical/DFS.py"

Sony Noise Beats Skullcandy Boat

Time taken for execution: 0.0000000455169

Process finished with exit code 0

# EXP: 5C

## A\* Algorithm

## AIM:

To Implement A\* Algorithm using Python

1. Initialize the open list
2. Initialize the closed list

put the starting node on the open list (you can leave its f at zero)

1. while the open list is not empty
   1. find the node with the least f on the open list, call it "q"
   2. pop q off the open list
   3. generate q's 8 successors and set their parents to q
   4. for each successor

i) if successor is the goal, stop search

successor.g = q.g + distance between successor and q successor.h = distance from goal to successor successor.f = successor.g + successor.h

1. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
2. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor

otherwise, add the node to the open list end (for loop)

1. push q on the closed list end (while loop)

Code:

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

def h(self, n):

H = {

'S':7,

'A': 6,

'B': 4,

'C': 2,

'G':0,

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found:')

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'S': [('A', 1), ('B', 4)],

'A': [('B', 2),('C',5),('G',12)],

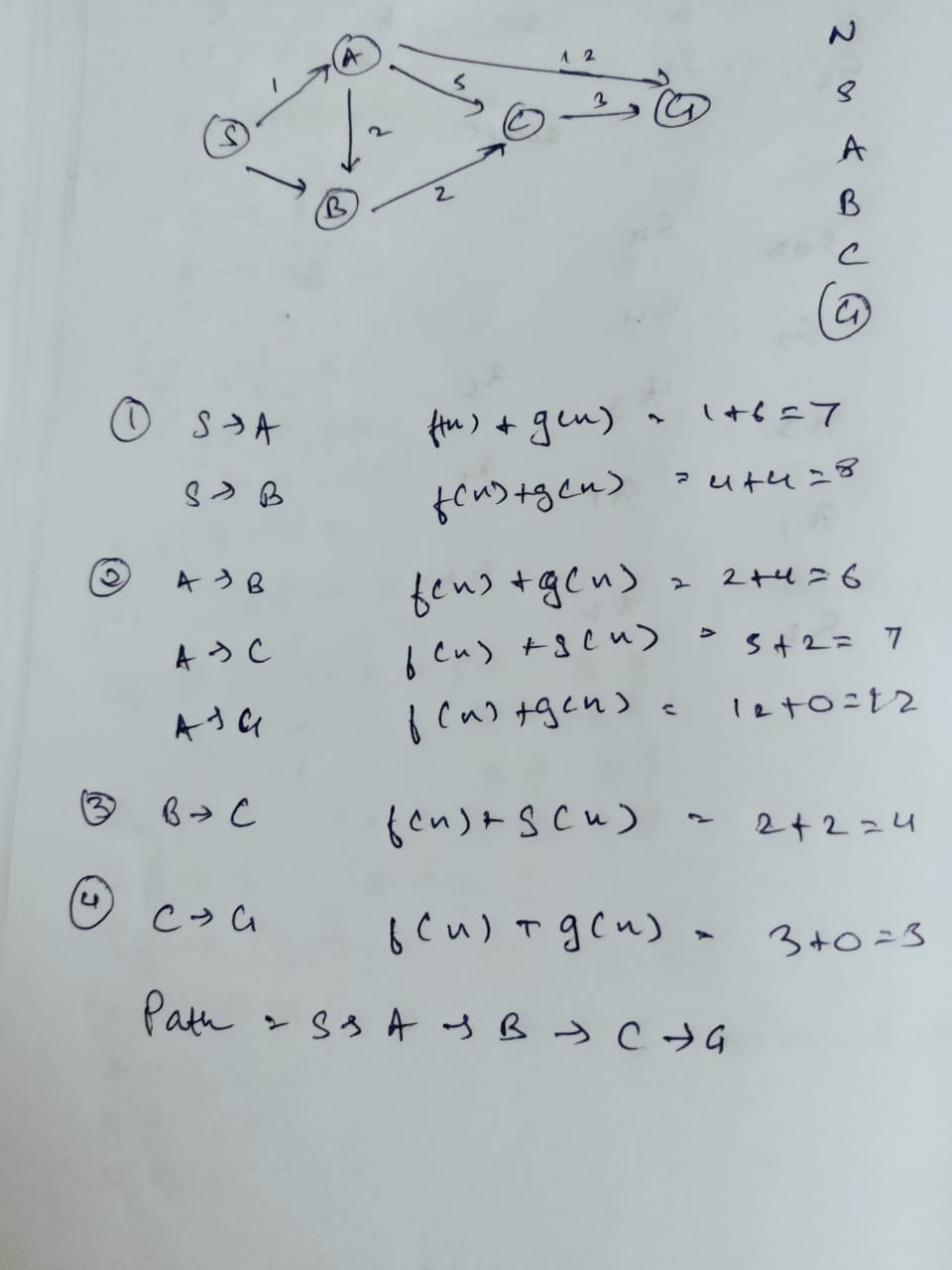
'B': [('C', 2)],

'C': [('G', 3)],

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('S', 'G')



# EXP:6

# Tic- tac-toe with Min max algorithm

## Aim:

To implement tic-tac-toe problem using min max algorithm

## Procedure:

Assuming X is the player, then we perform following steps:

* If the game is over, return the score from X's perspective.
* Otherwise get a list of new game states for every possible move
* Create a scores list
* For each of these states add the minimax result of that state to the scores list
* If it's X's turn, return the maximum score from the scores list
* If it's O's turn, return the minimum score from the scores list

## Program:

from math import inf as infinity

from random import choice

import platform

import time

from os import system

HUMAN = -1

COMP = +1

board = [

[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

]

def evaluate(state):

if wins(state, COMP):

score = +1

elifwins(state, HUMAN):

score = -1

else:

score = 0

return score

def wins(state, player):

win\_state = [

[state[0][0], state[0][1], state[0][2]],

[state[1][0], state[1][1], state[1][2]],

[state[2][0], state[2][1], state[2][2]],

[state[0][0], state[1][0], state[2][0]],

[state[0][1], state[1][1], state[2][1]],

[state[0][2], state[1][2], state[2][2]],

[state[0][0], state[1][1], state[2][2]],

[state[2][0], state[1][1], state[0][2]],

]

if [player, player, player] in win\_state:

return True

else:

return False

def game\_over(state):

return wins(state, HUMAN) or wins(state, COMP)

def empty\_cells(state):

cells = []

for x, row in enumerate(state):

for y, cell in enumerate(row):

if cell == 0:

cells.append([x, y])

return cells

def valid\_move(x, y):

if [x, y] in empty\_cells(board):

return True

else:

return False

def set\_move(x, y, player):

if valid\_move(x, y):

board[x][y] = player

return True

else:

return False

def minimax(state, depth, player):

if player == COMP:

best = [-1, -1, -infinity]

else:

best = [-1, -1, +infinity]

if depth == 0 or game\_over(state):

score = evaluate(state)

return [-1, -1, score]

for cell in empty\_cells(state):

x, y = cell[0], cell[1]

state[x][y] = player

score = minimax(state, depth - 1, -player)

state[x][y] = 0

score[0], score[1] = x, y

if player == COMP:

if score[2] > best[2]:

best = score # max value

else:

if score[2] < best[2]:

best = score # min value

return best

def clean():

"""

Clears the console

"""

os\_name = platform.system().lower()

if 'windows' in os\_name:

system('cls')

else:

system('clear')

def render(state, c\_choice, h\_choice):

"""

Print the board on console

:param state: current state of the board

"""

chars = {

-1: h\_choice,

+1: c\_choice,

0: ' '

}

str\_line = '---------------'

print('\n' + str\_line)

for row in state:

for cell in row:

symbol = chars[cell]

print(f'| {symbol} |', end='')

print('\n' + str\_line)

def ai\_turn(c\_choice, h\_choice):

depth = len(empty\_cells(board))

if depth == 0 or game\_over(board):

return

clean()

print(f'Computer turn [{c\_choice}]')

render(board, c\_choice, h\_choice)

if depth == 9:

x = choice([0, 1, 2])

y = choice([0, 1, 2])

else:

move = minimax(board, depth, COMP)

x, y = move[0], move[1]

set\_move(x, y, COMP)

time.sleep(1)

def human\_turn(c\_choice, h\_choice):

depth = len(empty\_cells(board))

if depth == 0 or game\_over(board):

return

# Dictionary of valid moves

move = -1

moves = {

1: [0, 0], 2: [0, 1], 3: [0, 2],

4: [1, 0], 5: [1, 1], 6: [1, 2],

7: [2, 0], 8: [2, 1], 9: [2, 2],

}

clean()

print(f'Human turn [{h\_choice}]')

render(board, c\_choice, h\_choice)

while move < 1 or move > 9:

try:

move = int(input('Use numpad (1..9): '))

coord = moves[move]

can\_move = set\_move(coord[0], coord[1], HUMAN)

if not can\_move:

print('Bad move')

move = -1

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

def main():

"""

Main function that calls all functions

"""

clean()

h\_choice = '' # X or O

c\_choice = '' # X or O

first = '' # if human is the first

# Human chooses X or O to play

while h\_choice != 'O' and h\_choice != 'X':

try:

print('')

h\_choice = input('Choose X or O\nChosen: ').upper()

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

# Setting computer's choice

if h\_choice == 'X':

c\_choice = 'O'

else:

c\_choice = 'X'

# Human may starts first

clean()

while first != 'Y' and first != 'N':

try:

first = input('First to start?[y/n]: ').upper()

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

# Main loop of this game

while len(empty\_cells(board)) > 0 and not game\_over(board):

if first == 'N':

ai\_turn(c\_choice, h\_choice)

first = ''

human\_turn(c\_choice, h\_choice)

ai\_turn(c\_choice, h\_choice)

# Game over message

if wins(board, HUMAN):

clean()

print(f'Human turn [{h\_choice}]')

render(board, c\_choice, h\_choice)

print('YOU WIN!')

elifwins(board, COMP):

clean()

print(f'Computer turn [{c\_choice}]')

render(board, c\_choice, h\_choice)

print('YOU LOSE!')

else:

clean()

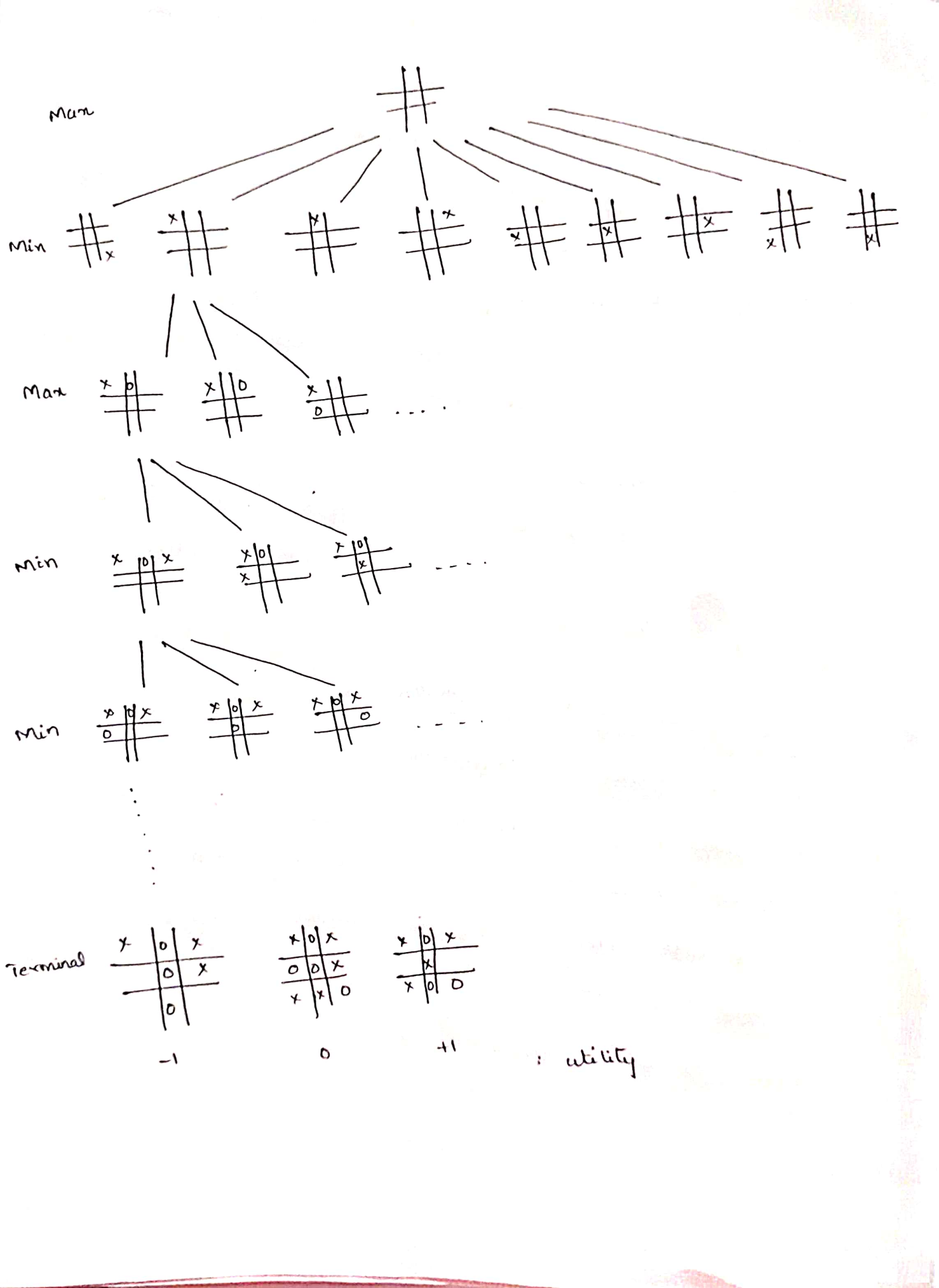
render(board, c\_choice, h\_choice)

print('DRAW!')

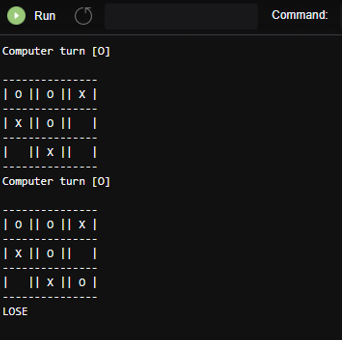
exit()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Manual Output:

Screenshot of output:

.

Result:

Tic tac toe problem using min max algorithm is successfully implemented.

# Experiment 7

# UNIFICATION AND RESOLUTION

## AIM:

* Implementation of unification and resolution

## ALGORITHM :

* Initialize the substitution set to be empty.
* Recursively unify atomic sentences:
* Check for Identical expression match.
* If one expression is a variable vi , and the other is a term ti which does not contain variable vi , then: Substitute ti / vi in the existing substitutions
* Add ti /vi to the substitution setlist.
* If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

## CODE:

**import** copy

**import** time

**class** Parameter:

variable\_count = 1

**def** \_\_init\_\_(self, name=**None**):

**if** name:

self.type = **"Constant"**

self.name = name

**else**:

self.type = **"Variable"**

self.name = **"v"** + str(Parameter.variable\_count)

Parameter.variable\_count += 1

**def** isConstant(self):

**return** self.type == **"Constant"**

**def** unify(self, type\_, name):

self.type = type\_

self.name = name

**def** \_\_eq\_\_(self, other): **return** self.name == other.name

**def** \_\_str\_\_(self):

**return** self.name

**class** Predicate:

**def** \_\_init\_\_(self, name, params):

self.name = name

self.params = params

**def** \_\_eq\_\_(self, other):

**return** self.name == other.name **and** all(a == b **for** a, b

**in** zip(self.params, other.params))

**def** \_\_str\_\_(self):

**return** self.name + **"("** + **","**.join(str(x) **for** x **in**

self.params) + **")"**

**def** getNegatedPredicate(self):

**return** Predicate(negatePredicate(self.name),

self.params)

**class** Sentence:

sentence\_count = 0

**def** \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

**for** predicate **in** string.split(**"|"**):

name = predicate[:predicate.find(**"("**)]

params = []

**for** param **in** predicate[predicate.find(**"("**) + 1:

predicate.find(**")"**)].split(**","**):

**if** param[0].islower():

**if** param **not in** local: *# Variable*

local[param] = Parameter()

self.variable\_map[local[param].name] =

local[param]

new\_param = local[param]

**else**:

new\_param = Parameter(param) self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

**def** getPredicates(self):

**return** [predicate.name **for** predicate **in** self.predicates]

**def** findPredicates(self, name):

**return** [predicate **for** predicate **in** self.predicates **if**

predicate.name == name]

**def** removePredicate(self, predicate):

self.predicates.remove(predicate)

**for** key, val **in** self.variable\_map.items():

**if not** val:

self.variable\_map.pop(key)

**def** containsVariable(self):

**return** any(**not** param.isConstant() **for** param **in**

self.variable\_map.values())

**def** \_\_eq\_\_(self, other):

**if** len(self.predicates) == 1 **and** self.predicates[0] ==

other:

**return True**

**return False**

**def** \_\_str\_\_(self):

**return ""**.join([str(predicate) **for** predicate **in**

self.predicates])

**class** KB:

**def** \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(**" "**, **""**) **for** x **in**

inputSentences]

self.sentences = []

self.sentence\_map = {}

**def** prepareKB(self):

self.convertSentencesToCNF()

**for** sentence\_string **in** self.inputSentences:

sentence = Sentence(sentence\_string)

**for** predicate **in** sentence.getPredicates():

self.sentence\_map[predicate] =

self.sentence\_map.get(predicate, []) + [sentence] **def** convertSentencesToCNF(self):

**for** sentenceIdx **in** range(len(self.inputSentences)):

**if "=>" in** self.inputSentences[sentenceIdx]: *# Do*

*negation of the Premise and add them as literal*

self.inputSentences[sentenceIdx] =

negateAntecedent(self.inputSentences[sentenceIdx])

**def** askQueries(self, queryList):

results = []

**for** query **in** queryList:

negatedQuery =

Sentence(negatePredicate(query.replace(**" "**, **""**)))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] =

self.sentence\_map.get(negatedPredicate.name, []) +

[negatedQuery]

self.timeLimit = time.time() + 40

**try**:

result = self.resolve([negatedPredicate],

[**False**]\*(len(self.inputSentences) + 1))

**except**:

result = **False**

self.sentence\_map = prev\_sentence\_map

**if** result:

results.append(**"TRUE"**)

**else**:

results.append(**"FALSE"**)

**return** results

**def** resolve(self, queryStack, visited, depth=0):

**if** time.time() > self.timeLimit:

**raise** Exception

**if** queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

**if** queryPredicateName **not in** self.sentence\_map:

**return False**

**else**:

queryPredicate = negatedQuery **for** kb\_sentence **in**

self.sentence\_map[queryPredicateName]:

**if not** visited[kb\_sentence.sentence\_index]:

**for** kbPredicate **in**

kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution =

performUnification(copy.deepcopy(queryPredicate),

copy.deepcopy(kbPredicate))

**if** canUnify:

newSentence =

copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack =

copy.deepcopy(queryStack)

**if** substitution:

**for** old, new **in**

substitution.items():

**if** old **in**

newSentence.variable\_map:

parameter =

newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(**"Variable" if** new[0].islower() **else "Constant"**,

new)

newSentence.variable\_map[new] = parameter

**for** predicate **in**

newQueryStack:

**for** index, param **in**

enumerate(predicate.params):

**if** param.name **in**

substitution:

new =

substitution[param.name]

predicate.params[index].unify(**"Variable" if** new[0].islower()

**else "Constant"**, new)

**for** predicate **in**

newSentence.predicates:

newQueryStack.append(predicate)

new\_visited =

copy.deepcopy(visited)

**if**

kb\_sentence.containsVariable() **and** len(kb\_sentence.predicates) >

1:

new\_visited[kb\_sentence.sentence\_index] = **True**

**if** self.resolve(newQueryStack,

new\_visited, depth + 1):

**return True**

**return False**

**return True**

**def** performUnification(queryPredicate, kbPredicate):

substitution = {}

**if** queryPredicate == kbPredicate:

**return True**, {}

**else**:

**for** query, kb **in** zip(queryPredicate.params,

kbPredicate.params):

**if** query == kb:

**continue**

**if** kb.isConstant():

**if not** query.isConstant():

**if** query.name **not in** substitution:

substitution[query.name] = kb.name

**elif** substitution[query.name] != kb.name:

**return False**, {}

query.unify(**"Constant"**, kb.name)

**else**:

**return False**, {}

**else**:

**if not** query.isConstant():

**if** kb.name **not in** substitution:

substitution[kb.name] = query.name

**elif** substitution[kb.name] != query.name:

**return False**, {}

kb.unify(**"Variable"**, query.name)

**else**:

**if** kb.name **not in** substitution:

substitution[kb.name] = query.name

**elif** substitution[kb.name] != query.name: **return False**, {}

**return True**, substitution

**def** negatePredicate(predicate):

**return** predicate[1:] **if** predicate[0] == **"~" else "~"** +

predicate

**def** negateAntecedent(sentence):

antecedent = sentence[:sentence.find(**"=>"**)]

premise = []

**for** predicate **in** antecedent.split(**"&"**):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find(**"=>"**) + 2:])

**return "|"**.join(premise)

**def** getInput(filename):

**with** open(filename, **"r"**) **as** file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() **for** \_ **in**

range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip() **for** \_ **in**

range(noOfSentences)]

**return** inputQueries, inputSentences

**def** printOutput(filename, results):

print(results)

**with** open(filename, **"w"**) **as** file:

**for** line **in** results:

file.write(line)

file.write(**"\n"**)

file.close()

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

inputQueries\_, inputSentences\_ = getInput(**"input.txt"**)

knowledgeBase = KB(inputSentences\_)

knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput(**"output.txt"**, results\_)

INPUT:

2

Friends(Alice,Bob,Charlie,Diana)

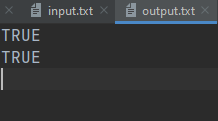
Friends(Diana,Charlie,Bob,Alice)

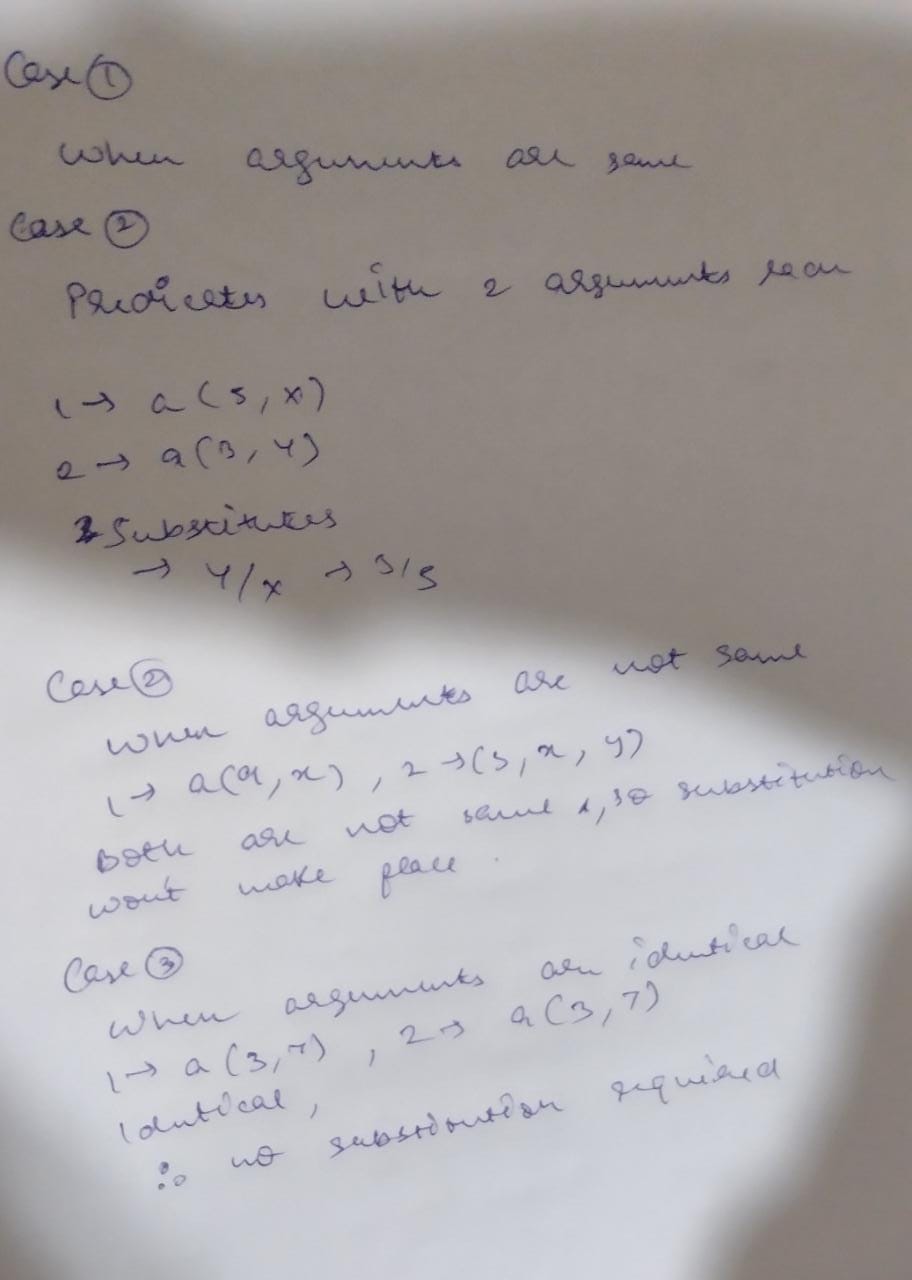
2

Friends(a,b,c,d)

NotFriends(a,b,c,d)

OUTPUT:



MANUAL OUTPUT: 

RESULT:

* Hence successfully implemented unification and resolution using python.