VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence LAB

Submitted by

ROHAN SATISH KUMAR (1BM21CS168)

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
Oct-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 lab" carried out by **ROHAN SATISH KUMAR(1BM21CS168),** who is a 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 year 2023. The Lab report has been approved as it satisfies the academic requirementsin respect of a **Artificial Intelligence lab (22CS5PCAIN)** work prescribed for the said degree.

SARITHA A N
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr.Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

`

Program 1:
Implement the vacuum cleaner program(20/11/23)

```
def vacuum_world():
 # initializing goal_state
 # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
    status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))
    if location_input == 'A':
    # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
     # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1 #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")
            if status_input_complement == '1':
    # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1 #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1 #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                # suck and mark clean
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':# if B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1 #cost for moving right
                print("COST for moving RIGHT " + str(cost))
```

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

```
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")
            # done cleaning
            print("GOAL STATE: ")
            print(goal_state)
            print("Performance Measurement: " + str(cost))

vacuum_world()
```

Program-01

Implement vacuum cleaner agent.

```
def vacuum_world():
    goal_State = {'A':'0','B':'0'}
    cost = 0
    location_input = input("Enter location of vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")
    print("Initial location condition" + str(goal_State))
    if location_input == 'A':
        print("Location A is dirty")
        goal_state['A'] = '0'
        cost += 1
        print("cost for cleaning A" + str(cost))
        print("location A has been cleaned")
        if status_input_complement == '1':
            print("location B is dirty")
            print("Moving right to the location B")
            cost += 1
            print("cost for moving right" + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("cost for suck" + str(cost))
```

```
print(" location B has been cleaned")
else:
print("No action" + str(cost))
print(cost).
print("location B is already clean")
else:
print(" vacuum is placed in location B")
if status_input == '1':
print(" location B is dirty")
goal_state['B'] = '0'
cost += 1
print(" cost for cleaning" + str(cost))
print("Location B has been cleaned")
if status_input_complement == '1':
print(" location A is dirty")
print(" moving left to the location A")
cost += 1
print(" cost for moving left" + str(cost))
goal_state['A'] = '0'
cost += 1
print(" cost for suck" + str(cost))
print(" location A has been cleaned")
else:
print(cost)
if status_input_complement == '1':
print(" location A has been dirty").
```

```
print("Moving left to the location A")
cost += 1
print("cost for moving left" + str(cost))
goal_state['A'] = '0'
cost += 1
print("cost for suck" + str(cost))
print("location A has been cleaned")
else:
print("No action" + str(cost))
print("location A is already clean")
print("goal state:")
print("goal_state)
print("performance measurement:" + str(cost))
Vacuum_world()
```

Output:

```
Enter location of vacuum B
Enter status of B 0
Enter status of other room 1
Initial location condition {'A': '0', 'B': '0'}
location A is already clean
Location B is dirty
Moving right to location B
cost for moving right 1
Cost for Suck 2
location B has been cleaned.
```

output

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
```

Program 2:

Implement the 8 Puzzle Bíeadth Ïiíst Seaích Algoíithm.(11/12/23)

```
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)
    cost=0

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print("queue")
        for q in queue:
          print(q)
          print("..........***..........")

        print(source[0],'|',source[1],'|',source[2])
        print(source[3],'|',source[4],'|', source[5])
        print(source[6],'|', source[7],'|',source[8])
        print()
        cost=cost+1

        if source == target:
            print("success")
            print("path cost",cost)
            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)
```

11-Dec-2017

- Explore the working of Tic Tac Toe using min max strategy.

```
import sys

def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in
                range(3)):
            return True
        if all(board[i][j] == player for j in
                range(3)):
            return True

    if all(board[i][i] == player for i in range(3)
        or all(board[i][2-i] == player for i in
                range(3)):
            return True
    return False

def is_board_full(board):
    return all(board[i][j] != ' ' for i in range(
                for j in range(3))
```

```
def minimax(board, depth, is_maximizer):
    scores = {
        'x' : 10-depth,
        'o' : depth-10,
        'Tie' : 0
    }

    if check_winner(board, 'x'):
        return scores['x']
    elif check_winner(board, 'o'):
        return scores['o']
    elif is_board_full(board):
        return scores['Tie']

    if is_maximizer:
        best_score = -sys.maxsize
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'x'
                    score = minimax(board, depth+1,
                                    False)
                    board[i][j] = ' '
                    best_score = max(best_score, score)
        return best_score
    else:
        best_score = sys.maxsize
        for i in range(3):
            for j in range(3):
                if board[i][j] = ' ':
```

```
                board[i][j] = 'O'
                score = minimax(board, depth+1, True)
                board[i][j] = ' '
                best_score = min(best_score, score)
        return best_score

def find_best_move(board):
    best_score = -sys.maxsize
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                score = minimax(board, 0, False)
                board[i][j] = ' '

                if score > best_score:
                    best_score = score
                    best_move = (i, j)
    return best_move

def play_game():
    board = [[' ' for _ in range(3)] for _ in
             range(3)]
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while not check_winner(board, 'X') and not
      check_winner(board, 'O') and not is_board_full(
```

(5). Implement the 8 puzzle Breath First search algorithm

```
from collections import deque

def find_zero(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i,j

def generate_next_states(state):
    next_states = []
    zero_i, zero_j = find_zero(state)

    movements = [(0,1), (0,-1), (1,0), (-1,0)]

    for move in movements:
        new_i, new_j = zero_i + move[0], zero_j +
                                        move[1]
        if 0 <= new_i < 3 and
           0 <= new_j < 3:

            new_state = [row[:] for row in state]
            new_state[zero_i][zero_j], new_state[new_i]
            [new_j] = new_state[new_i][new_j],
                      new_state[zero_i], zero_j]
            new_states.append(new_state)

    return next_states

def bfs(start):
    queue = deque([(start, [])])
```

```
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        visited.add(str(current_state))

        if is_goal(current_state):
            return path

        for next_state in generate_next_states(cu_state):
            if str(next_state) not in visited:
                queue.append((next_state, path +
                              [next_state]))
                visited.add(str(next_state))

    return None

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

solution = bfs(initial_state)

if solution:
    print("solution found!")
    print("steps to reach the goal:")
    for step in solution:
        print(step)

else:
    print("No solution exists for this puzzle.")
```

Output: Welcome to Tic Tac Toe!

Enter your move(row and column): 1 1

O

AI's move:

X

O

Enter your move: 0 2

X

O

AI's move:

X O

O

X

Enter your move: 1 0

X

O O O

X

AI's move:

X O

O O X

X

Enter your move: 2 1

X X O

O O X

X O X

AI's move:

X X O

O O X

X O

Enter your move: 2 2

X X O

O O X

X O O

It's a tie!

Output

```
queue
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8

queue
[1, 2, 3, 4, 5, 6, 7, 0, 8]
----------***------------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8

queue
[0, 2, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 0, 6, 4, 7, 8]
----------***------------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
```

```
queue
[1, 2, 3, 5, 0, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 4, 0, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 4, 5, 6, 7, 8, 0]
----------***------------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8

queue
[1, 2, 3, 4, 0, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 4, 5, 6, 7, 8, 0]
----------***------------
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8

queue
[1, 2, 3, 4, 5, 6, 7, 8, 0]
----------***------------
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 0, 3, 5, 2, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 7, 6, 4, 0, 8]
----------***------------
[1, 2, 3, 5, 6, 0, 4, 7, 8]
----------***------------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8

queue
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 0, 3, 5, 2, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 7, 6, 4, 0, 8]
----------***------------
[1, 2, 3, 5, 6, 0, 4, 7, 8]
----------***------------
[1, 0, 3, 4, 2, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 0, 4, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 4, 6, 0, 7, 5, 8]
----------***------------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0

success
path cost 7
```

Program 3:

Exploíe the woíking of Ɪ'ic Ɪ'ac Ɪ'oe using Min max stíategy.(11/12/23)

```
board = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]
print("0,0|0,1|0,2")
print("1,0|1,1|1,2")
print("2,0|,2,1|2,2 \n\n")
def print_board():
  for row in board:
    print("|".join(row))
    print("-" * 5)

def check_winner(player):
  for i in range(3):
    if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):
      return True

  if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):
    return True
  return False

def is_full():
  return all([cell != " " for row in board for cell in row])

def minimax(depth, is_maximizing):
  if check_winner("X"):
    return -1
  if check_winner("O"):
    return 1
  if is_full():
    return 0
  if is_maximizing:
    max_eval = float("-inf")
    for i in range(3):
      for j in range(3):
        if board[i][j] == " ":
          board[i][j] = "O"
          eval = minimax(depth + 1, False)
          board[i][j] = " "
          max_eval = max(max_eval, eval)
    return max_eval
  else:
    min_eval = float("inf")
```

```
    for i in range(3):
      for j in range(3):
        if board[i][j] == " ":
            board[i][j] = "X"
            eval = minimax(depth + 1, True)
            board[i][j] = " "
            min_eval = min(min_eval, eval)

    return min_eval

def ai_move():
  best_move = None
  best_eval = float("-inf")
  for i in range(3):
    for j in range(3):
      if board[i][j] == " ":
        board[i][j] = "O"
        eval = minimax(0, False)
        board[i][j] = " "
        if eval > best_eval:
          best_eval = eval
          best_move = (i, j)

  return best_move

while not is_full() and not check_winner("X") and not check_winner("O"):
  print_board()
  row = int(input("Enter row (0, 1, or 2): "))
  col = int(input("Enter column (0, 1, or 2): "))
  if board[row][col] == " ":
    board[row][col] = "X"
    if check_winner("X"):
    print_board()
    print("You win!")
    break
    if is_full():
      print_board()
      print("It's a draw!")
      break
    ai_row, ai_col = ai_move()
    board[ai_row][ai_col] = "O"
    if check_winner("O"):
    print_board()
      print("AI wins!")
```

```
      break

  else:
    print("Cell is already occupied. Try again.")
```

output

```
player_row, player_col = map(int, input("Enter
        your move (row & column):").split())
if board[player_row][player_col] != ' ':
    print("Invalid move. Try again:")
    continue

board[player_row][player_col] = 'O'
print_board(board)

if check_winner(board, 'O'):
    print("Congratulations! You win!")
    break
if is_board_full(board):
    print("It's a tie!")
    break

print("AI's move:")
ai_row, ai_col = find_best_move(board)
board[ai_row][ai_col] = 'X'
print_board(board)

if check_winner(board, 'X'):
    print("AI wins!")
    break.
if is_board_full(board):
    print("It's a tie")
    break

play_game()
```

Iterative deepening Search

```
def iterative_deepening_search(graph, start, goal):
    depth_limit = 0
    while true:
        result, path = depth_limited_search(graph, start,
                                            goal, depth_limit, [limit])
        if result == goal
            return result, path
        depth_limit += 1

def depth_limited_search(graph, current, goal, depth_limi
                                                path):
    if current == goal:
        return current, path
    if depth_limit == 0:
        return None, ()
    if depth_limit >= 0:
        for neighbor in graph[current]:
            return
            result, new_path = depth_limited_search(
                graph, neighbor, goal, depth_limit-1,
                    path + [neighbor]).
    if result == goal
        return result, new_path
return None, ()
def main():
    graph = {}
```

Output:

Solution found!

Steps to reach the goal:

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

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

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

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

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

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

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

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

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

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

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

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

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

P8/12/23

```
0,0|0,1|0,2
1,0|1,1|1,2
2,0|,2,1|2,2


 | |
-----
 | |
-----
 | |
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
O| |
-----
 |X|
-----
 | |
-----
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 2
O| |X
-----
 |X|
-----
O| |
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 0
O| |X
-----
X|X|O
-----
O| |
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
O|O|X
-----
X|X|O
-----
O|X|
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
O|O|X
-----
X|X|O
-----
O|X|X
-----
It's a draw!
```

Program 4:

Implement Iteíative deepening seaích algoíithm.(18/12/23)

```
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def__init__(self,vertices):

        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)
        self.ans = list()

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search
    # from given source 'src'
    def DLS(self,src,target,maxDepth,l):

        if src == target :
          #  print(self.ans)
            return True

        # If reached the maximum depth, stop recursing.
        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1,l)):
                    l.append(i)
                    return True
        return False

    # IDDFS to search if target is reachable from v.
    # It uses recursive DLS()
    def IDDFS(self,src, target, maxDepth):
```

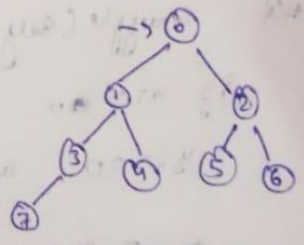
```
        # Repeatedly depth-limit search till the
        # maximum depth
        for i in range(maxDepth):
            l = []
            if (self.DLS(src, target, i,l)):
                l.append(src)
                l.reverse()
                return l
        return l

# Create a graph given in the above diagram
n,e = map(int ,input("Enter no.of vertices and edges").split())
g = Graph (n);
for i in range(e):
    a,b = map(int , input().split())
    g.addEdge(a,b)

# g.addEdge(0, 1)
# g.addEdge(0, 2)
# g.addEdge(1, 3)
# g.addEdge(1, 4)
# g.addEdge(2, 5)
# g.addEdge(2, 6)

target = int(input("Enter the target vertex"))
maxDepth = int(input("Enter the max depth"))
src = 0
l = g.IDDFS(src, target, maxDepth)
if len(l)!=0:
    print(l)
    print ("Target is reachable from source " +
        "within max depth")
else :
    print ("Target is NOT reachable from source " +
        "within max depth")
```

Output

```
while True:
    Node = input("enter a node (or 'done' to
                                finish):")
    if node.lower() == 'done':
        break
neighbor = input("enter neighbor for {node}").spli
    graph[node] = neighbors
Start-node = input("enter the Start node:")
goal-node = input("enter the goal node:")
result, path = Iterative_deepening_search
                    (graph, start_node, goal_node)
if result:
    print("Goal '{goal_node}' found.path = {path}")
else:
    Print("Goal '{goal_node}' not found")
if __name__ == "__main__":
    main()
```

o/p

```
enter no. of vertices : 8
enter no. of edges : 7
enter the edge(x,y) : 0 1
enter the edge(x,y) : 0 2
enter the edge(x,y) : 1 3
enter the edge(x,y) : 1 4
enter the edge(x,y) : 2 5
enter the edge(x,y) : 2 6
enter the edge(x,y) : 3 7
enter the source node : 0
enter the target node : 7
enter max-depth : 4
Target 7 is reachable from source 0
Traversal path 0->3->7
```



Status (5) Iterate Using DFS Search

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '_')
        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):
        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):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append()
            temp.append()
        return temp
    def find(self, puz, x):
        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):
        self.n = size
        self.open() = []
        self.closed = []
    def accept(self):
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
```

```
def f(self, start.goal):
    return self.h(start.data, goal) + start.level

def h(self, start, goal):
    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):
    print("Take the start of matrix\n")
    start = self.accept()
    print("enter the goal state of matrix\n")
    goal = self.accept()
    start = Node(start, 0, 0)
    start.fval = self.f(start, goal)
    self.open.append(start)
    print("\n\n")
    while(True):
        cur = self.open(0)
        print(" ")
        print(" | ")
        print(" \ ")
        print("\t" / "\n")
```

```
        for i in cur.data:
            for j in i:
                print(j, end = " ")
            print(" ")
        if(self.h(cur.data, goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.h(i, goal)
            self.open.append(i)
        self.closed.append(cur.data)

def self.open(0):
    self.open = sort(key = lambda x: x.fval,
                     reverse = False)

puz = puzzle(2)
puz.process()
```

Op:

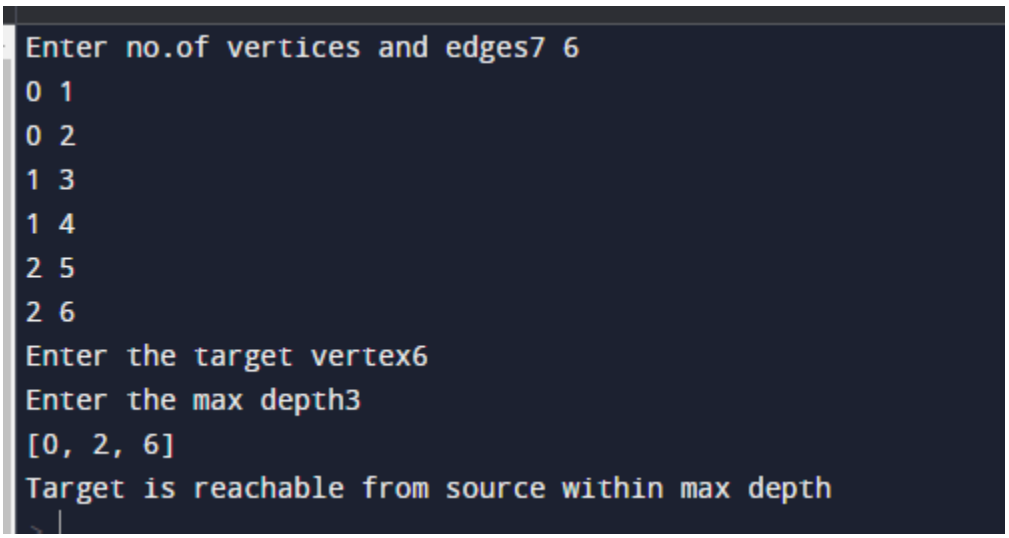
Enter the start state matrix.

1 2 3
4 5 6
- 7 8

Enter the goal state matrix

1 2 3
4 5 6
7 8 -

↓
↓
↓



Program 5:

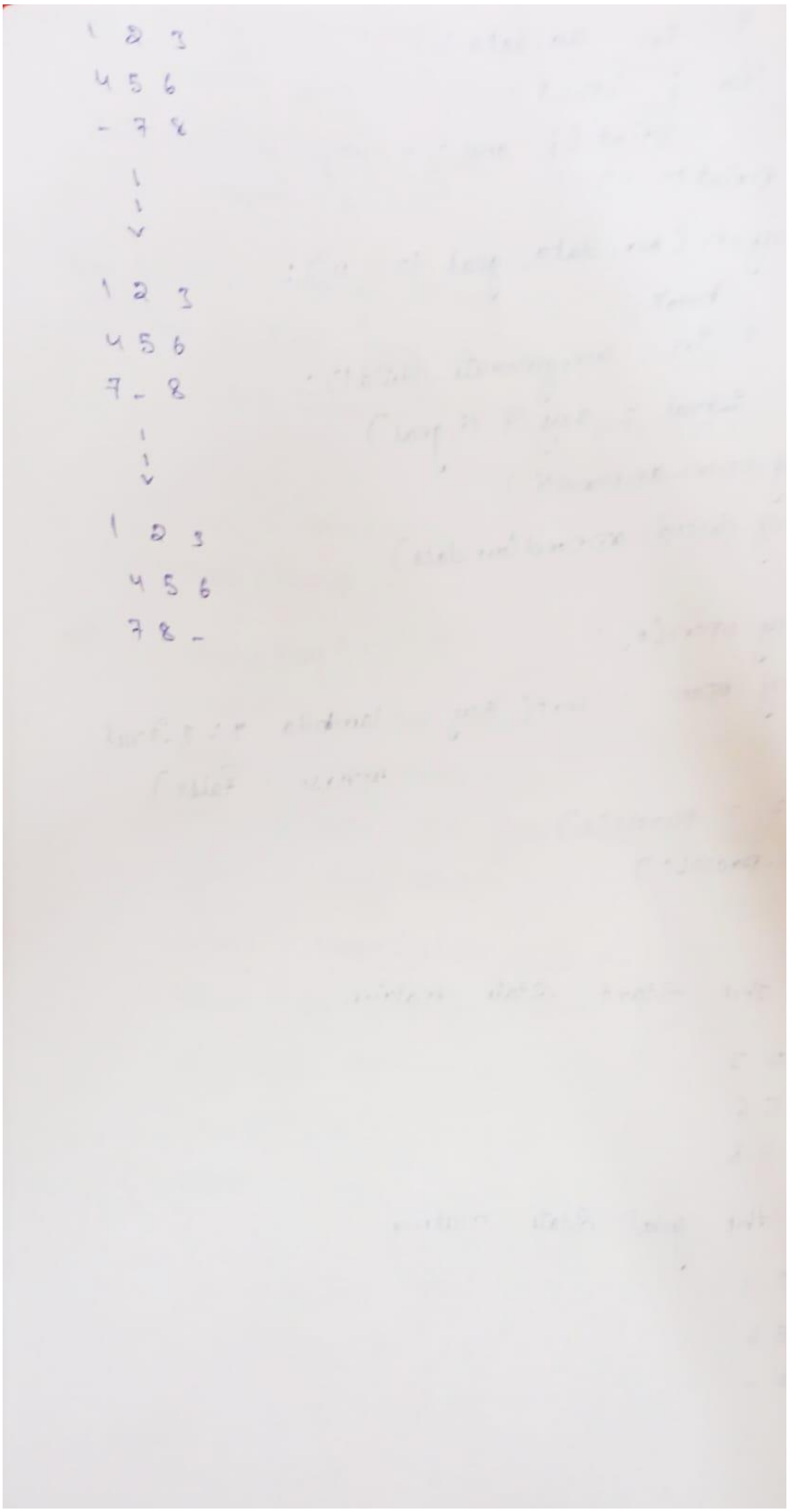
Implement A* foí 8 puzzle píoblem(8/1/24)

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
class Node:
    def__init_(self,data,level,fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x,y = self.find(self.data,'_')
        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):
        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):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        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):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

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

    def h(self,start,goal):
        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):
        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)
        self.open.append(start)

        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(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 2 3
4 5 6
- 7 8
1 2 3
4 5 6
7 - 8
1 2 3
4 5 6
7 8 -

22-1-24 ⑥. Create a knowledge based bot using propositional logic & show that the given query entails the KB or not.

//Algorithm

function TT entails (KB, α) returns T/F

//Inputs : KB, the knowledge bot
α, the query : a, sentence

//Symbols : a list of proprtion symbols in KB & α

function TT. CHECK - ALL (KB, α, Symbol, model)
returns true or false

if EMPTY? (symbols) then
if PL_True? (KB, model, then
return PL-True (α, model)
else return true,
else do
P = FIRST (symbols)
rest = REST (symbols)
return TT-CHECK-ALL (KB, α, rest,
EXTEND (P, true, model) and
TT-CHECK-ALL (KB, α, rest, EXTEND (P, false
model))

Knowledge Base (Truth Table).

T	T	T	T	F	F	F
T	T	F	T	T	T	T
T	F	T	T	F	F	F
T	F	F	T	T	T	T
F	T	T	T	F	F	F
F	T	F	T	T	F	F
F	F	T	F	F	F	F
F	F	F	F	T	F	F

The knowledge base entails the query

Output enter result : $(p \vee q) \wedge (p \wedge r)$

enter query : p

** Truth Table Reference **

F	T
T	T
F	T
T	T
F	F
F	F
F	F
F	F

The KB entails the query

Q2 query = 'r'

The KB doesn't entail query

```
Enter the start state matrix

1 2 3
5 6 _
7 8 4
Enter the goal state matrix

1 2 3
5 8 6
_ 7 4
|
  |
 \'/
1 2 3
5 6 _
7 8 4
cost: 3

  |
  |
 \'/

1 2 3
5 _ 6
7 8 4
cost: 3

  |
  |
 \'/

1 2 3
5 8 6
7 _ 4
cost: 3

  |
  |
 \'/

1 2 3
```

Program 6:

Creation of Knowledge Base using prepositional logic and show that the query entails the KB or not (22/1/24)

```
combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False,
False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)]
variable={'p':0,'q':1, 'r':2}
kb=''
q=''
priority={'~':3,'v':1,'^':2}
def input_rules():
    global kb, y
    kb = (input("Enter rule: "))
    y = input("Enter the Query: ")
def entailment():
    global kb, q
    print('*'*10+"Truth Table Reference"+'*'*10)
    print('kb','alpha')
    print('*'*10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(y), comb)
        print(s, f)
        print('-'*10)
        if s and not f:
            return False
    return  True
def isOperand(c):
    return c.isalpha() and c!='v'

def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0
```

```
def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1]<=priority[c2]
    except KeyError:
        return False
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1, val2):
```

```
    if i == '^':
        return val2 and val1
    return val2 or val1

input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

22-1-24 (4). Create a knowledge not using propositional logic & prove the given query using resolution

Algorithm

Function PL-RESOLUTION (KB, α) return T/F

//Inputs : KB, the knowledge bot,
α sentence, Propositional logic α,
the query a sentence in PL

clauses ⇒ the set of clause in the CNF representation of KB ∧ ¬α

new ← { }

loop do

For each pair of clause (Ci, Cj) in clauses do

resolvent ⇐ PL-RESOLVE (Ci, Cj)

if resolvent contains the empty clause then return true

new ← new ∪ resolvents

if new ⊆ clause then return false

clause ← clause ∪ new

KB

P

query

q

P ∧ Q → R

S ∨ T → Q

T

S

=> KB ∨ ¬R

convert to CNF

output:

```
Enter rule: (pvq)^(p^~r)
Enter the Query: p
**********Truth Table Reference**********
kb alpha
**********
False True
----------
True True
----------
False True
----------
True True
----------
False False
----------
False False
----------
False False
----------
False False
----------
The Knowledge Base entails query
>
```

Program 7:

Creation of Knowledge Base using prepositional logic and prove the query using resolution(22/1/24)

```
kb = []


def CLEAR():
    global kb
```

```
    kb = []


def TELL(sentence):
    global kb
    # If the sentence is a clause, insert directly.
    if isClause(sentence):
        kb.append(sentence)
    # If not, convert to CNF, and then insert clauses one by one.
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        # Insert clauses one by one when there are multiple clauses
        if isAndList(sentenceCNF):
            for s in sentenceCNF[1:]:
                kb.append(s)
        else:
            kb.append(sentenceCNF)


def ASK(sentence):
    global kb

    # Negate the sentence, and convert it to CNF accordingly.
    if isClause(sentence):
        neg = negation(sentence)
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        neg = convertCNF(negation(sentenceCNF))

    # Insert individual clauses that we need to ask to ask_list.
    ask_list = []
    if isAndList(neg):
        for n in neg[1:]:
            nCNF = makeCNF(n)
```

```
            if type(nCNF).__name__== 'list':
                ask_list.insert(0, nCNF)
            else:
                ask_list.insert(0, nCNF)
        else:
            ask_list = [neg]
        clauses = ask_list + kb[:]
        while True:
            new_clauses = []
            for c1 in clauses:
                for c2 in clauses:
                    if c1 is not c2:
                        resolved = resolve(c1, c2)
                        if resolved == False:
                            continue
                        if resolved == []:
                            return True
                        new_clauses.append(resolved)

            if len(new_clauses) == 0:
                return False

            new_in_clauses = True
            for n in new_clauses:
                if n not in clauses:
                    new_in_clauses = False
                    clauses.append(n)

            if new_in_clauses:
                return False
        return False


    def resolve(arg_one, arg_two):
        resolved = False

        s1 = make_sentence(arg_one)
        s2 = make_sentence(arg_two)

        resolve_s1 = None
```

```
resolve_s2 = None

# Two for loops that iterate through the two clauses.
for i in s1:
    if isNotList(i):
        a1 = i[1]
        a1_not = True
    else:
        a1 = i
        a1_not = False

    for j in s2:
        if isNotList(j):
            a2 = j[1]
            a2_not = True
        else:
            a2 = j
            a2_not = False

        # cancel out two literals such as 'a' $ ['not', 'a']
        if a1 == a2:
            if a1_not != a2_not:
                # Return False if resolution already happend
                # but contradiction still exists.
                if resolved:
                    return False
                else:
                    resolved = True
                    resolve_s1 = i
                    resolve_s2 = j
                    break
                # Return False if not resolution happened
if not resolved:
    return False

# Remove the literals that are canceled
s1.remove(resolve_s1)
s2.remove(resolve_s2)

# # Remove duplicates
```

```
    result = clear_duplicate(s1 + s2)

    # Format the result.
    if len(result) == 1:
        return result[0]
    elif len(result) > 1:
        result.insert(0, 'or')

    return result


def make_sentence(arg):
    if isLiteral(arg) or isNotList(arg):
        return [arg]
    if isOrList(arg):
        return clear_duplicate(arg[1:])
    return


def clear_duplicate(arg):
    result = []
    for i in range(0, len(arg)):
        if arg[i] not in arg[i+1:]:
            result.append(arg[i])
    return result


def isClause(sentence):
    if isLiteral(sentence):
        return True
    if isNotList(sentence):
        if isLiteral(sentence[1]):
            return True
        else:
            return False
    if isOrList(sentence):
        for i in range(1, len(sentence)):
            if len(sentence[i]) > 2:
                return False
            elif not isClause(sentence[i]):
```

```
                return False
        return True
    return False


def isCNF(sentence):
    if isClause(sentence):
        return True
    elif isAndList(sentence):
        for s in sentence[1:]:
            if not isClause(s):
                return False
        return True
    return False


def negation(sentence):
    if isLiteral(sentence):
        return ['not', sentence]
    if isNotList(sentence):
        return sentence[1]

    # DeMorgan:
    if isAndList(sentence):
        result = ['or']
        for i in sentence[1:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', sentence])
        return result
    if isOrList(sentence):
        result = ['and']
        for i in sentence[:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', i])
        return result
    return None
```

```
def convertCNF(sentence):
    while not isCNF(sentence):
        if sentence is None:
            return None
        sentence = makeCNF(sentence)
    return sentence


def makeCNF(sentence):
   if isLiteral(sentence):
        return sentence

    if (type(sentence).__name__== 'list'):
        operand = sentence[0]
        if isNotList(sentence):
            if isLiteral(sentence[1]):
                return sentence
            cnf = makeCNF(sentence[1])
            if cnf[0] == 'not':
                return makeCNF(cnf[1])
            if cnf[0] == 'or':
                result = ['and']
                for i in range(1, len(cnf)):
                    result.append(makeCNF(['not', cnf[i]]))
                return result
            if cnf[0] == 'and':
                result = ['or']
                for i in range(1, len(cnf)):
                    result.append(makeCNF(['not', cnf[i]]))
                return result
            return "False: not"

        if operand == 'implies' and len(sentence) == 3:
            return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])

        if operand == 'biconditional' and len(sentence) == 3:
            s1 = makeCNF(['implies', sentence[1], sentence[2]])
            s2 = makeCNF(['implies', sentence[2], sentence[1]])
```

```
    return makeCNF(['and', s1, s2])

if isAndList(sentence):
    result = ['and']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isAndList(cnf):
            for i in range(1, len(cnf)):
                result.append(makeCNF(cnf[i]))
            continue
        result.append(makeCNF(cnf))
    return result

if isOrList(sentence):
    result1 = ['or']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isOrList(cnf):
            for i in range(1, len(cnf)):
                result1.append(makeCNF(cnf[i]))
            continue
        result1.append(makeCNF(cnf))
        # Associativity:
    while True:
        result2 = ['and']
        and_clause = None
        for r in result1:
            if isAndList(r):
                and_clause = r
                break

        # Finish when there's no more 'and' lists
        # inside of 'or' lists
        if not and_clause:
            return result1

        result1.remove(and_clause)
```

```
            for i in range(1, len(and_clause)):
                temp = ['or', and_clause[i]]
                for o in result1[1:]:
                    temp.append(makeCNF(o))
                result2.append(makeCNF(temp))
            result1 = makeCNF(result2)
        return None
    return None


def isLiteral(item):
    if type(item).__name__ == 'str':
        return True
    return False


def isNotList(item):
    if type(item).__name__ == 'list':
        if len(item) == 2:
            if item[0] == 'not':
                return True
    return False


def isAndList(item):
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'and':
                return True
    return False


def isOrList(item):
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'or':
                return True
    return False
```

```
CLEAR()

TELL('p')
TELL(['implies', ['and', 'p', 'q'], 'r'])
TELL(['implies', ['or', 's', 't'], 'q'])
TELL('t')
print(ASK('r'))
```

Output

Program 8:

Implement unification in first order logic(29/1/24)

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

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)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
```

```
        if val == old:
            attributes[index] = new
    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:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
```

```
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot
be unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    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 []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
```

```
Enter the first expression
knows(y,f(x))
Enter the second expression
knows(pri,p)
The substitutions are:
['pri / y', 'f(x) / p']
```

Program 9:

Convert a given first order logic statement into Conjunctive Normal Form (CNF).(29/1/24)

```
import re

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 DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'
    string = ''.join(s)
```

```
    string = string.replace('~~','')
    return f'[{string}]' if flag else string
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement
def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+ statement[i+1:] +
'=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.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] + 'V' + statement[i+1:]
            statement = statement[:br] + new_statement if br > 0 else new_statement
        while '~∀' in statement:
            i = statement.index('~∀')
            statement = list(statement)
            statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
            statement = ''.join(statement)
        while '~∃' in statement:
            i = statement.index('~∃')
            s = list(statement)
            s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
            statement = ''.join(s)
        statement = statement.replace('~[∀','[~∀')
        statement = statement.replace('~[∃','[~∃')
        expr = '(~[∀V∃].)'
        statements = re.findall(expr, statement)
        for s in statements:
            statement = statement.replace(s, fol_to_cnf(s))
        expr = '~\[[^]]+\]'
        statements = re.findall(expr, statement)
        for s in statements:
            statement = statement.replace(s, DeMorgan(s))
        return statement
    def main():
        print("Enter FOL:")
        fol = input()
        print("The CNF form of the given FOL is: ")
        print(Skolemization(fol_to_cnf(fol)))

    main()
```

Forward chaining [Reasoning]

Procedure forward-chain(KB, f)

if there is a fact in KB that is identical to f then return add f to KB

for each $(l_1 \wedge l_2 \wedge \ldots \wedge l_n \Rightarrow d)$ in KB such that for some i, unify $(l_i, f) = \theta$ succeeds do

FIND-AND-INFER(KB, $[l_1, \ldots l_{i-1}, l_{i+1} \ldots l_n]$, d, θ)

Procedure FIND and INFER(KB, conditions, conclusion, θ)

if conditions = [] then

forward-chain [KB, subst(θ, conclusion)]

else for each f in KB such that UNIFY

(f, Close(θ, First(conditions))) = θ_1, do

Find and Infer(KB, REST conditions)

conclusion, compose(θ_1, θ_2))

end

O/P :

KB :

missile(x) => weapon(x)

missile(m_1)

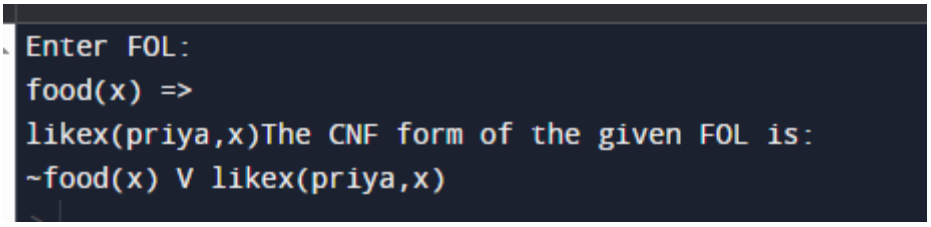
enemy(x, america) => hostile(x)

american(west)

enemy(china, america)

owns(china, m_1)

missile(x) & owns(china, x) => sells(west, x, china).



Program 10:

Forward Chaining (29/1/24)

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}')
def main():
    kb = KB()
    print("Enter KB: (enter e to exit)")
```

```
    while  True:
        t = input()
        if(t == 'e'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()
main()
```

Output

④ Convert a given first order logic statement into conjunctive Normal Form (CNF).

Algo:

(1). Eliminate biconditional & implications

Eliminate $\Rightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

(2). remove $\neg$ inwards:

$\neg(\forall x\ P) \equiv \exists x \neg P$

$\neg(\exists x\ P) \equiv \forall x \neg P$

$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$

$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$

$\neg\neg\alpha \equiv \alpha$.

(3). Standardize variables part by renaming them each quantifier should be a different variable.

(4). skolemize: Each existential variable is replaced by a stolen constant or universally quantified variable.

(5). Drop universal quantifiers

(6). Distribute $\wedge$ over $\vee$

OIP: Enter FOL:

Food(x) $\Rightarrow$ likes(Sai, x)

The CNF Form of the given FOL is

$\neg$Food(x) $\vee$ likes(Sai, x).

```
Enter KB: (enter e to exit)
missile(x) => weapon(x)
missile(m1)
enemy(x,america) => hostile(x)
american(west)
enemy(china,america)
owns(china,m1)
missile(x) & owns(china,x) => sells(west,x,china)
american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)
e
Enter Query:
criminal(x)
Querying criminal(x):
    1. criminal(west)
All facts:
    1. weapon(m1)
    2. criminal(west)
    3. missile(m1)
    4. owns(china,m1)
    5. american(west)
    6. sells(west,m1,china)
    7. hostile(china)
    8. enemy(china,america)
>
```

⑤ Implement unification in FOL

Algo: Function unify(x, y, θ) returns a substitution to make x & y identical.

If θ = failure then return false
else if x = y then return θ
else if variable?(x) then return unify-var(x, y, θ)
else if variable?(y) then return unify-var(y, x, θ)
else if compound?(x) & compound?(x, op.θ)
return unify(x.args, y.args, unify(x, op, y, op, θ))
else if list?(x) & list?(y) then return unify(x.Rest, y.Rest, unify(x.First, y.First, θ))
else return failure

Function unify var(var, x, θ) return a substitution

if {var/val} ∈ θ then return unify(val, x, θ)
else if {x/val} ∈ θ then unify(var, val, θ)
else if occur-check?(var, x) then return failure
else return add {var/x} to θ.

O/P:

Enter the first expression
knows(y, f(x))

Enter the second expression
knows(pre, p)

the substitutions are:-
['pre/y', 'f(x)/p'].