Q.1) Write a python code using if-else ladder to print age.

```
age = int(input('Enter a person's age:'))
if age <= 1 : print ('Infant')
elif age > 1 and age < 13 : print ('child')
elif age >= 13 and age < 20 : print ('teenager')
elif age >= 20 : print ('Adult')
```

O/P

```
Enter a person's age:
1
Infant
7
child
15
teenager
50
Adult
```

Q.2) Print the integers in reverse order.

```
num = int(input('Enter a integer : '))
reverse_num = 0
while num != 0 :
    digit = num % 10
    reverse_num = reverse_num * 10 + digit
    num //= 10
print ("Reverse number :" + str(reverse_num))
```

O/P

```
Enter a integer : 321
Reverse number : 123
```

Q.3> Program to print the pattern

```
1
1 2
1 2 3
1 2 3 4
```

```
s = int(input('Enter a number :'))
for i in range(1, s+1):
    for j in range(1, i+1):
        print(j, end=" ")
print("")
```

O/P

```
Enter a number :4
1
1 2
1 2 3
1 2 3 4
```

Q.4> Power of a number : x^n

```
x = int(input('Enter a number :'))
n = int(input('Enter a number :'))
print("The power is :", pow(x,n))
```

O/P :-

```
Enter a number: 2
Enter a number: 3
The power is : 8
```

Q.5:- Any Switch case program

```
num = int (input ('Enter a number:'))
match int (str (num) [0]):
case 1:
    print ('hi!')
case 2:
    print ('Hello!')
case 3:
    print ("Hey")
```

O/P

```
Enter a number: 3
Hey
```

8/10/24

17/11/2023

Prg 1 => Tic Tac Toe

```
board = [' '
 for x in
 range(10)]

def insertLetter(letter, pos):
    board[pos] = letter

def spaceIsFree(pos):
    return board[pos] == ' '

def printBoard(board):
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')

def isWinner(bo, le):
    return (bo[7] == le and bo[8] == le and bo[9] == le) or
           (bo[4] == le and bo[5] == le and
            bo[6] == le) or
           (bo[1] == le and bo[2] == le and bo[3] == le) or
           (bo[1] == le and bo[4] == le and bo[7] == le):
```

```
bo[2] == le and bo[5] == le and bo[8] == le or (
bo[3] == le and bo[6] == le and bo[9] == le or (
bo[1] == le and bo[5] == le and bo[9] == le or (
bo[3] == le and bo[5] == le and bo[7] == le)

def playerMove():
    run = True
    while run:
        move = input('Please select a position to place an
                                        'X' (1-9): ')
        try:
            move = int(move)
            if move > 0 and move < 10:
                if spaceIsFree(move):
                    run = False
                    insertLetter('X', move)
                else:
                    print('Sorry, this space is occupied!')
            else:
                print('Please type a number within the range!')
        except:
            print('Please type a number!')

def compMove():
    possibleMoves = [x for x, letter in enumerate(board)
        if letter == ' ' and x != 0]
    move = 0
```

```
for let in ['O', 'X']:
    for i in possibleMoves:
        boardCopy = board[:]
        boardCopy[i] = let
        if isWinner(boardCopy, let):
            move = i
            return move
cornersOpen = []
for i in possibleMoves:
    if i in [1,3,7,9]:
        cornersOpen.append(i)

if len(cornersOpen) > 0:
    move = selectRandom(cornersOpen)
    return move

if 5 in possibleMoves:
    move = 5
    return move

edgesOpen = []
for i in possibleMoves:
    if i in [2,4,6,8]:
        edgesOpen.append(i)

if len(edgesOpen) > 0:
    move = selectRandom(edgesOpen)
    return move
```

```
import random
ln = len(li)
r = random.randrange(0, ln)
return li[r]

def isBoardFull(board):
    if board.count(' ') > 1:
        return False
    else:
        return True

def main():
    print('Welcome to Tic Tac Toe!')
    printBoard(board)

    while not (isBoardFull(board)):
        if not (isWinner(board, 'O')):
            playerMove()
            printBoard(board)

        else:
            print('Sorry, O's won this time!')
            break
    if not (isWinner(board, 'X')):
        move = compMove()
        if move == 0:
            print('Tie Game!')

        else:
            insertLetter('O', move)
```

```
        print('Computer plays at ...')
        printBoard(board)

    else:
        print('X's won this time! Good Job!')
        breaks

    if isBoardFull(board):
        print('Tie Game!')

while True:
    answer = input('Do you want to play again? (y/
    if answer.lower() == 'y' or answer.lower == 'yes':
        board = [' ' for x in range((10)]
        print('--------')
        main()
    else:
        breaks
```

O/P:

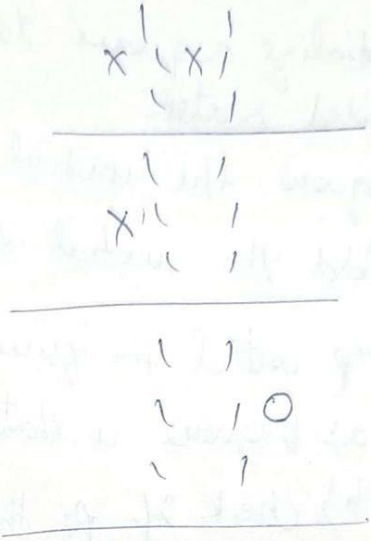
kaggle - down

Output:

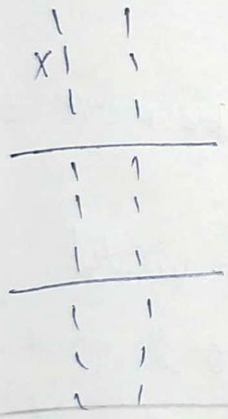
Do you want to play again? (Y/N) Y

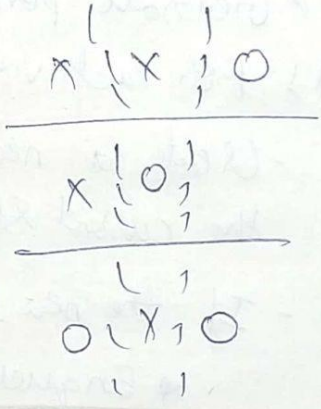
① Welcome to Tic Tac Toe!

Please select a position x : 2



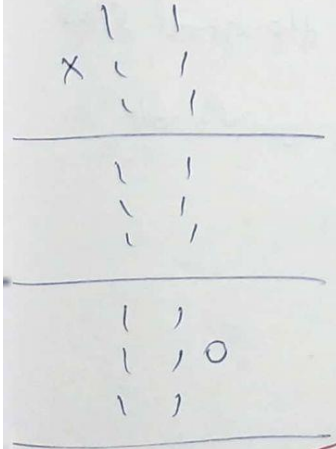
Please select a position to place 'X' (1-9) : 1 Compute place : 3

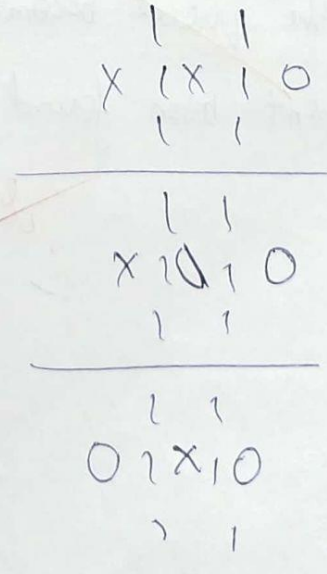




Compute placed an 'O' in position 9:

Please select a position : 4





Sathy O's won this time!

17.11.24

24/11/2023

8-Puzzle

Algorithm:-

1.) Initialize a queue to store states and a set of track visited states.

2.) Enqueue the initial state on the puzzle onto the queue

3.) Add the initial state to the visited set.

4.) Loop until the queue is empty.

a.) Dequeue a state from the queue.

b.) Check if the dequeued state is the goal state. If yes, the puzzle is solved.

c.) Generate possible moves from the current state.

d.) For each valid move:

- Create a new state by applying the move to the current state.
- If the new state has not been visited.
  - Enqueue the new state onto the queue.
  - Add the new state to the visited set.

5.) If the queue becomes empty and the goal state hasn't been found, the puzzle is unsolvable.

```
def print_grid(src):
    state = src.copy()
    state[state.index(-1)] = '_'
    print(f"""
    {state[0]} {state[1]} {state[2]}
    {state[3]} {state[4]} {state[5]}
    {state[6]} {state[7]} {state[8]}
    """)

def h(state, target):
    # Manhattan distance
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3
        dist += abs(x1 - x2) + abs(y1 - y2)
    return dist

def astar(src, target):
    # A* algorithm
    states = [src]
    g = 0
    visited_states = set()

    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.add(tuple(state))
            print_grid(state)
```

```
    if state == target:
        print("Success")
    return

    moves += [move for move in possible_moves(state, vi
        if moves not in moves]
costs = [g+h(move, target) for move in moves]
states = [moves[i] for i in range(len(moves)) if cost
        == min(costs)]
g += 1
print("No success")

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if 0 > b-3 >= 0:
        d += 'u'
    if 0 > b+3 >= 0:
        d += 'd'
    if b not in [2, 5, 8]:
        d += 'r'
    if b not in [0, 3, 6]:
        d += 'l'

    pos_moves = []
    for move in d:
        pos_moves.append(gen(state, move, b))
return [move for
```

```
def get_state(state, direction, b):
    temp = state.copy()
    if direction == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if direction == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if direction == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    if direction == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    return temp
```

```
src = [1, 2, 5, 3, 4, -1, 6, 7, 8]
target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]
astar(src, target)
```

O/P:

Level : 0

1	2	5
3	4	_
6	7	8

Level : 1

1	2	3
3	4	5
6	7	8

Level: 2

1 _ 2

3 4 5

6 7 8

Level : 3

_ 1 2

3 4 5

6 7 8

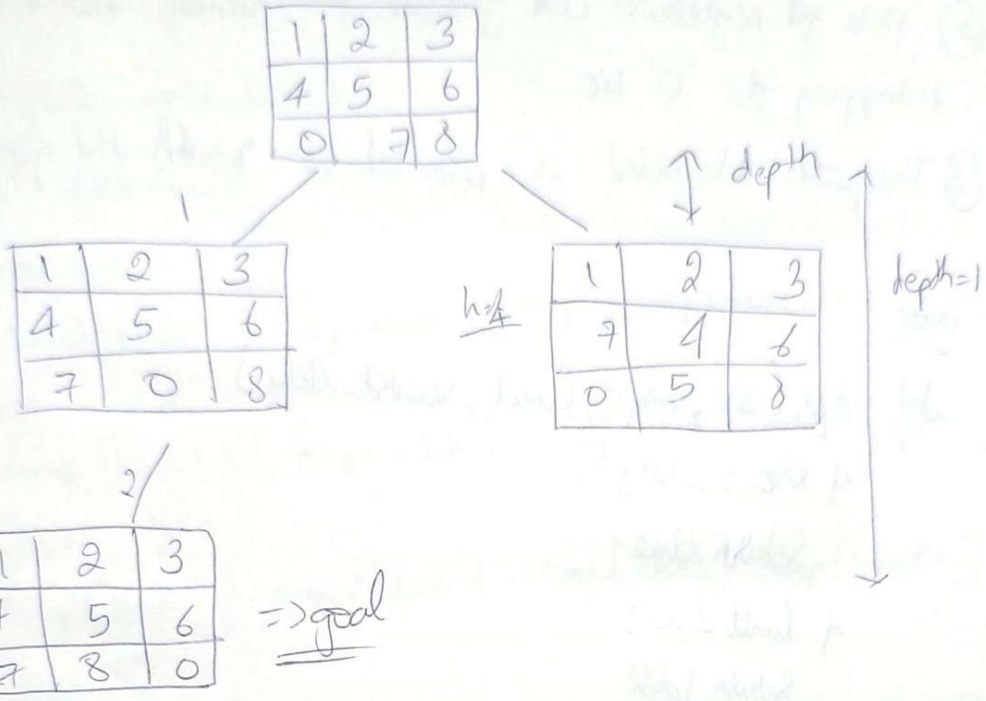
Success

30-11-22

8/12/2022

8-Puzzle iterative deepning Search Algorithm

Algorithm :-



=> Algorithm:

① Initialize the initial state = [] and goal state for the 8 puzzle

goal_state = [1,2,3,4,5,6,7,8,0] # 0 is the blank space

② Set the depth = 1 and expand the initial state

The depth_limited search (depth) is performed

```
if node state = goal
    return node
else
    for neighbour in get node (state)
        child = puzzlenode (neighbour_node)
        result = depth_limited_search (depth-1)
        if result = True:
            return result
```

③ After one iteration ... the depth by 1 and perform depth_limited_search again

④ Here get_neighbours will generate the possible moves by swapping the '0' tile.

⑤ The path traversed is printed to reach the goal state

Code:-

```
def dfs(src, target, limit, visited_states):
    if src == target:
        return true
    if limit <= 0:
        return false
    visited_states.append(src)
    moves = possible_moves(src, visited_states);
    return True
return false

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if b not in [0, 1, 2]:
        d += 'u'
    if b not in [6, 7, 8]:
        d += 'd'
    if b not in [2, 5, 8]:
        d += 'r'
    if b not in [0, 3, 6]:
        d += 'l'
pos_moves = []
```

```
    for move in d:
        pos_moves.append(gen(state, move, b))
    return [move for move in pos_moves if move not in visited_states]

def gen(state, move, blank):
    temp = state.copy()
    if move == 'u':
        temp[blank-3], temp[blank] = temp[blank], temp[blank-3]
    if move == 'd':
        temp[blank+3], temp[blank] = temp[blank], temp[blank+3]
    if move == 'r':
        temp[blank+1], temp[blank] = temp[blank], temp[blank+1]
    if move == 'l':
        temp[blank-1], temp[blank] = temp[blank], temp[blank-1]
    return temp

def idfs(src, target, depth):
    for i in range(depth):
        visited_states = []
        if dfs(src, target, i+1, visited_states):
            print(True)
            return
    print(False)

def get_user_input():
    print("Enter input:")
    puzzle = []
    for i in range(3):
        row = []
```

```
    for j in range(3):
        value = int(input("Enter values "))
        row.append(value)
    puzzle.append(row)
return puzzle

initial_puzzle = get_user_input()
solution = iterative_deepening_search(initial_puzzle)
if solution is not None:
    print("Solution found:")
    for row in solution:
        print(row)
```

O/P:-

Enter initial puzzle state:
Enter value for position (1,1): 1
Enter value for position (1,2): 2
Enter value for position (1,3): 3
Enter value for position (2,1): 4
Enter value for position (2,2): 5
Enter value for position (2,3): 0
Enter value for position (3,1): 7
Enter value for position (3,2): 8
Enter value for position (3,3): 6
Checking depth=4

Moving up

1	2	0
4	5	3
7	8	6

Moving down

1	2	3
4	5	0
7	8	6

Moving left

1	0	2
4	5	3
7	8	6

Moving right

1	2	0
4	5	3
7	8	6

Moving down

1	2	3
4	5	6
7	8	0

Solution found:

1	2	3
4	5	6
7	8	9

=> Solving 8-puzzle using A* Algorithm

goal

1	2	3
4	5	6
7	8	0

h=0

1	2	3
0	4	6
7	5	8

h=4

h=3

1	2	3
4	0	6
7	5	8

h=5

0	2	3
1	4	6
7	5	8

h=4

1	2	3
7	4	6
0	5	8

=> Algorithm:-

① Create the initial state and goal state for the problem in A* the heuristics is considered, lowest heuristic node is considered in each step

f value = h value + path cost

② Initially expand the node, find the location of empty tile and generate the node Calculate the heuristic function value

$$f(x) = h(x) + g(x)$$

h(x): misplaced tile

g(x): depth from starting node [path cost]

③ Maintain two list namely 'open' and 'close'. The nodes (states) generated are stored in the open list, sort using the f(x) values.

④ The goal is reached when h(x)=0, implies that all the tile are in the correct position.

Code:-

```
def print_grid(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
        f"""
    { state[0] } { state[1] } { state[2] }
    { state[3] } { state[4] } { state[5] }
    { state[6] } { state[7] } { state[8] }
    """
    )

def h(state, target):
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3
        dist += abs(x1-x2) + abs(y1-y2)
    return dist

def astar(src, target):
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        print(f"level: {g}")
        moves = []
```

```
    if state == target:
        print("Success")
        return
    moves += [moves for move in possible_moves(
        state, visited_states) if move not in moves]
    costs = [g + h(move, target) for move in moves]
    states = [moves[i]
              for i in range(len(moves)) if costs[i] == min(cost)]
    g += 1
else: print("No Success")

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if 9 > b-3 >= 0:
        d += 'u'
    if 9 > b+3 >= 0:
        d += 'd'
    if b not in [2,5,8]:
        d += 'r'
    if b not in [0,3,6]:
        d += 'l'
    pos_moves = []
    for moves in d:
        pos_moves.append(gen(state, move, b))
    return [move for move in pos_moves if tuple(moved) not in
            visited_states]
```

```
def gen(state, direction, b):
    temp = state.copy()
    if direction == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if direction == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if direction == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    if direction == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    return temp
```

```
src = [1, 2, 5, 3, 4, _, 6, 7, 8]
target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]
astar(src, target)
```

O/P:-

Level: 0

1 2 5
3 4 _
6 7 8

Level: 1

1 2 _
3 4 5
6 7 8

Level: 2

1 _ 2
3 4 5
6 7 8

Level: 3

_ 1 2
3 4 5
6 7 8

8/12/23

Success

22/12/2023

Vaccum Cleaner Implementation

->Vaccum cleaner Agent :-

Objective :- Given MxN grid (floor) create an agent that moves around the grid until the entire grid is clean

① Moves the agent anyway you see fit until the floor is clean

② Agent can start at any tile on the floor.

Code:-

① Create your clean function

② Create a print function that shows current position of the vaccum cleaner at every move your agent makes.

③ Floor is represented in this manner '1' represents dirty and '0' represents clean.

Algorithm:-

Step 1:- Define Goal state and cost

- Initialize the goal state representing the cleanliness status of room A and B.
- Set the cost counter to track the agent actions.

Step 2:- Take User Input for Vacuum's Location and Room Status

- Prompt the user to input the location of the Vacuum and the status (clean/dirty) of that location.
- Input the complement status of the other room.

Step3: Execute Actions based on Location and Status

- Check the location of the vacuum and the status of that room.
- Clean the room if it's dirty and update the goal state
- Increment the cost based on the performed action.

Step 4:- Perform Actions based on Status of other room

- Check the status of other room based on user input
- Clean the other room if it's dirty and update the goal state
- Increment the cost for the actions taken.

Step 5 :- Display Final Results

- Display the final goal state indicating the cleanliness status of rooms A and B.
- Show the performance measurement, which is the accumulated cost of actions taken by the vacuum agent.

Code:-

```
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("Location B is already clean.")
    if status_input == '0':
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Moving Right to the location B.")
            cost += 1
```

```
        print("Cost for Suck" + str(cost))
        print("Location A has been cleaned.")
else:
    print(cost)
    print("Location A is already clean.")
    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("NO action " + str(cost))
        print("Location A is already clean.")
print("Goal State:")
print(goal_state)
print("Performance Measurement: " + str(cost))
vacuum_world()
```

o/p

O/P

Enter Location of Vacuum: A
Enter Status of A: 1
Enter Status of other room: 1
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 Right: 2
cost for Suck: 3
Location B has been cleaned.
Goal state:
{'A': 0, 'B': 0}
Performance Measurement: 3

22/12/23

29/12/2023

Knowledge Based Entailment

```
valiable = {"Cloudy": 0, "Rainy": 1, "Sunny": 2}
priority = {"~": 3, "v": 1, "^": 2}

def _eval(i, val1, val2):
    if i == "^":
        return val2 and val1
    return val2 or val1

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 CheckEntailment():
    Kb = input("Enter Knowledge base:")
    query = input("Enter query:")
    combinations = [
        [T, T, T],
        [T, T, F],
        [T, F, T],
        [T, F, F],
        [F, T, T],
        [F, T, F],
        [F, F, T],
        [F, F, F],
    ]
    postfix_kb = toPostfix(Kb)
    postfix_q = toPostfix(query)
    for combination in combinations:
        eval_kb = evaluatePostfix(postfix_kb, combination)
        eval_q = evaluatePostfix(postfix_q, combinations)
        print(combination, ":Kb=", eval_kb, ":q=", eval_q)
```

```
if eval_KB == True:
    if eval_q == False:
        print ("Doesn't Entail!!")
        return False
print ("Entails!")

if __name__ == "__main__":
    CheckEntailment()
```

O/P :-

Enter Knowledge base:

(Cloudy v Rainy)^~(Sunny v Cloudy)

Enter query : Cloudy v Rainy v Sunny

Cloudy	Rainy	Sunny	KB	Query
F	F	F	F	F
F	F	T	F	T
F	T	F	T	T
F	T	T	F	T
T	F	F	F	T
T	F	T	T	T
T	T	F	T	T
T	T	T	T	T

Entails

29/12/2022

Knowledge based Resolution

```
def disjunctify(clauses):
    disjuncts = []
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
    return disjuncts

def getResolvant(ci, cj, di, dj):
    resolvant = list(ci) + list(cj)
    resolvant.remove(di)
    resolvant.remove(dj)
    return tuple(resolvant)

def resolve(ci, cj):
    for di in ci:
        for dj in cj:
            if di == '~'+dj or dj == '~'+di:
                return getResolvant(ci, cj, di, dj)

def checkResolution(clauses, query):
    clauses += [query if query.startswith('~') else '~'+query]
    proposition = '^'.join(['('+clause+')' for clause in clauses])
    print('Trying to prove {proposition} by contradiction...')
```

```
clauses = disjunctify(clauses)
resolved = False
new = set()

while not resolved:
    n = len(clauses)
    pairs = [(clauses[i], clauses[j]) for i in
        range(n) for j in range(i+1, n)]
    for (ci, cj) in pairs:
        resolvant = resolve(ci, cj)
        if not resolvant:
            resolved = True
            break
    new = new.union(set(resolvant))
if new.issubset(set(clauses)):
    break

for clause in new:
    if clause not in clauses:
        clauses.append(clause)
```

O/P:

Enter the clauses seperated by a space:

$(A \vee B) \wedge (C \vee D \vee j) \wedge (E \vee G)$

Enter the query: $A \vee B \vee C$

Trying to prove $((A \wedge (V) \wedge (B) \wedge (E') \wedge (V) \wedge$

by contradiction.

Knowledge base entails the query.

Knowledge based Unification

Algorithm :-

Unify (ψ1, ψ2)

Step 1 :- If ψ1 or ψ is a variable or constant, then :

a.) If ψ1 or ψ2 or identical, then return NIL.

b.) Else if ψ1 is a variable,

a. then if ψ1 occurs in ψ2, then return FAILURE

b. Else return {ψ2/ψ1}.

c.) Else if ψ2 is a variable

a. If ψ2 occurs in ψ1 then return FAILURE

b. Else return {(ψ1/ψ2)}

d.) Else return FAILURE.

Step 2 : If the initial Predicate symbol is ψ1 and ψ2 are not same, then return FAILURE.

Step 3 : If ψ1 and ψ2 have a different number of arguments, then return FAILURE.

Step 4 : Set Substitution set (Subst) to NIL.

Step 5 : For i=1 to no of elements in ψ1.

a) Call Unify function

b.) If S = failure then return FAILURE.

c.) If S ≠ NIL then do,

a.) Apply S to the remainder of both L1, L2.

b.) SUBST = APPEND (S, Subst)

Step 6 : Return SUBST

Code:-

```
import re

def getAttributes(expr):
    expr = expr.split("(")[1:]
    expr = "(".join(expr)
    expr = expr[:-1]
    expr = re.split("(?<!\(),(?!\))", expr)

def getInitialPredicate(expr):
    return expr.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def getFirstPart(expr):
    attr = getAttributes(expr)
    return attr[0]

def getRemainingPart(expr):
    predicate = getInitialPredicate(expr)
    attr = getAttributes(expr)
    newExpr = predicate + "(" + ",".join(attr[1:]) + ")"
    return newExpr
```

```
def (unify (exp1, exp2)):
    if exp1 == exp2:
        return []

if is constant (exp1) and is constant (exp2):
    if exp1 != exp2
        return false

if is Constant (exp1):
    return [(exp1, exp2)]

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

if isVariable (exp2):
    if checkOccurs (exp2, exp1):
        return false
    else:
        return [(exp1, exp2)]

exp1 = input ("Expression 1:")
exp2 = input ("Expression 2:")
subs = unify (exp1, exp2)
print ("Substitutions:")
print (subs)
```

O/P:-

Expression1: Knows(x, john)

Expression2: Knows(Smith, y)

Substitutions:-

[('Smith', 'x'), ('john', 'y')]

Forward Reasoning

Algorithm:-

Step1:-

Initialization:-

Set up your initial Knowledge base with the relevant information.

Step2:-

Selection:-

Choose a rule or fact to start the reasoning process.

Step3:- Application:-

Use the selected rule or fact to derive new information through logical reasoning.

Step4:- Update:-

Incorporate the newly derived information into the Knowledge base.

Step5:- Iteration:-

Repeat Steps 2-4 until the goal is reached or until no further inferences can be made.

Output:-

Enter the number of statements in Knowledge Base

3

P(a)

q(b)

p(x) & q(x) => r(x)

Enter Query:

r(x)

Querying r(x):

1. r(a)

All facts:

1. P(a)
2. q(b)
3. r(a)

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 getPredicate(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 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}
        return Fact(f)
```

```
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 main():
    kb = KB()
    n = int(input("Enter no of statements in knowledge base:"))
    for _ in range(n):
        kb.tell(input())
    print("Enter query:")
    query = input()
    kb.query(query)
    kb.display()
main()
```

19.01.24

(10) First order logic statements into Conjunctive Normal form [CNF]

Algorithm:-

① User interaction:-

- Get FOL statement from the user

② Conversion to CNF:-

- Call "fol_to_cnf" to convert the FOL statement to CNF

③ Skolemization:-

- Call "Skolemization" to perform Skolemization on the CNF statement

④ Print Result:-

- Print the Skolemized CNF statement.

O/P:-

Enter FOL statement : $\forall x (P(x) \Rightarrow Q(x))$

FOL converted to CNF:

$[\sim P(A) \mid Q(A)]$

Code:-

```
import re

def getpredicates(string):
    expr = '[a-z~]+\([A-za-z,]+\)'
    return re.findall(expr, string)

def getAttributes(string):
    expr = '\([^)]+\)'
    return [m for m in str(matches) if m.isalpha()]

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

```
SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
statement = ''.join(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])
if ''.join(attributes).islower():
    statement = statement.replace(match[1], SKOLEM_pop())
else:
    v = [a for a in attributes if not a.islower()][0]
return statement

def fol_to_cnf(fol):
statement = fol.replace("<=>", "_")
while '_' in statement:
i = statement.index('_')
statement = new_statement
expr = '\[(A-Z)+\]'
statements = re.findall(statements):
if '[' in s and ']' not in s:
    statements[i] += ']'
    statements = statements.replace(s, fol_to_cnf(s))
return statement
```

```
for s in statements:
    statements = statements.replace(s, fol_to_cnf(s))
exp = '~\([^]]+\)'
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement

def main():
    statement = input("Enter FOL statement:")
    print("FOL converted to CNF:
        {skolemization(fol_to_cnf(statement))}

main()
```

23/1/24