B.M.S. COLLEGE OF ENGINEERING BENGALURU
Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

Submitted in partial fulfillment for the 6th Semester Laboratory

Bachelor of Technology in
Computer Science and Engineering

Submitted by:

Jigar D Patel
1BM21CS081

Department of Computer Science and Engineering B.M.S.
College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
Mar-June 2021

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by Jigar D Patel (1BM21CS081) during the 5th Semester September-January2021.

Signature of the Faculty Incharge:

Dr. Asha G R
Assistant Professor
Department of Computer Science and Engineering B.M.S.
College of Engineering, Bangalore

Table of Contents

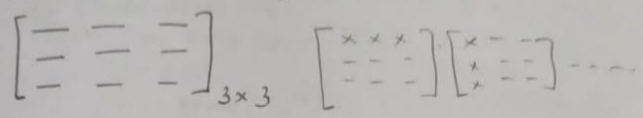
17/11/23

Program 1 : Implement Tic-Tac-Toe Game

Algo

→ create a empty global 3×3 array

→ Make the board of 3×3 tic-tac-to

→ 

→ Then we

```
import random

def create_board(self):                          } To create a board
    for i in Range(3):
        row = []
    for i in Range(3):
        row.append('_')
    self.board.append(row)

def get_random_player(self):                     } First Random Player
    return random.randint(0,1)

def fix_spot(self, row, col, player):            } To get rows & cols
    self.board[row][col] = player

def winning_condition(self, player):             } winning condition.
    #checking rows
    for i in range(n):
        win = True
        for j in Range(n):
            if self.board[i][j] != player:
                win = False
                break
        if win:
            return win
    #checking colums
    for i in range(n):
        win = True
        for j in range(n):
            if self.board[j][i] != player
                win = False
```

```
if win:
    return win

# checking diogonal
win = True
for i in range(n):
    if self.board[i][i] != player:
        win = False
        break
if win:
    return win

def swop_player_turn(self, player):
    return 'X' if player == 'O' else 'O'
```

Algorithm

1. make a board and initialize the value.
2. Import the Random Library in order to get the Random values
3. Once the Random player is selected, enter the Row and columns to Mark 'X' or 'O'
4. After that, Make a function to check the winning condition.
   - (a) First we will check all the row condition
   - (b) Then we will check all the Column condition
   - (c) we will check all the diogonal condition
5. If the the Mark is present consecutive in winning condition, we mark as 'win' or Restart the Match.

Must do better

17-11-23

Code

```
import random
class TicTacToe:
    def __init__(self):
        self.board=[]
    def create_board(self):
        for i in range(3):
            row=[]
            for .append('_')
        self.baxd.append(row)

    def get_random_first_player(self):
        return random.randint(0,1)
    def fix_spot(self,row,col,player):
        self.board[row][col]=player
    def is_player_win(self,player):
        win=None
        n=len(self.board)
        for i in range(n):
            win=True
            for j in range(n):
                if self.board[i][j]!=player:
                    win=False
                    break
            if win:
                return win
    for i in range(n):
        win=True
        for j in range(n):
            if self.board[j][i]!=player:
                win=False
                break
        if win:
            return win
```

```
        win = True
        for i in range (n):
            if self.board [i][i] != player
                win = false
                break
        if win:
            return w
        win = True
        for i in range (n):
            if self.board [i][n-1-i] != player:
                win = false
                break
        if win:
            return win
        return false
        for row in self.board:
            for item in row:
                if item == '_':
                    return false
        return True

    def swap_player_turn (self, player):
        return 'X' if player == 'O' else 'O'

    def show_board (self):
        for row in self.board:
            for item in row:
                print (item, end = " ")

            print ()

    def start (self):
        self.creak_board ()
        player = 'X' if self.get_random_first_player () ==
                    1 else 'O'
        while True:
            print (f" player {player} turn")
            Self.show_board()
            row, col = list (
```

```
        map(int, input("Enter row & column numbers to fix
            spot: ").split())
        print()
    self.fix_spot(row-1, col-1, player)
        if self.is_player_win(player):
            print(f"player {player} win the game!")
            break
        if self.is_board_filled():
            print("Match Draw!")
            break
        player = self.swap_player_turn(player)

print()
self.show_board()

tic_tac_toe = TicTacToe()
Tic_tac_toe.start()
```

O/P

Player X turn

_ _ _
_ _ _
_ _ _

Enter row and column numbers to fix spot: 1 1

Player O turn

X _ _
_ _ _
_ _ _

Enter row and colum to fix spot: 2 1

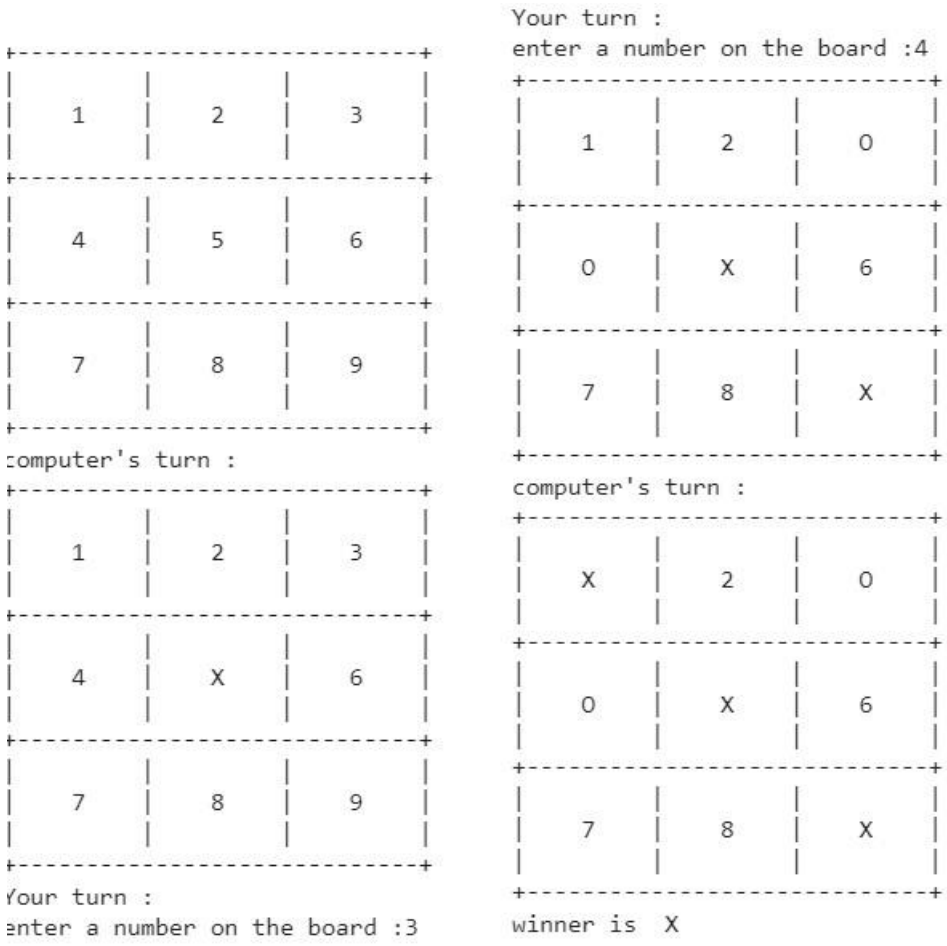
Player X turn

X _ _
O _ _
_ _ _

Enter row & column numbers to fix spot: 1 2

Play O turn	Player X turn	Play O turn	player X turn	player X
X X _	X X O	X X O	X X O	X X O
O _ _	O _ _	O X _	O X _	O X _
_ _ _	_ _ _	_ _ _	_ _ O	_ _ O

Output:


+-----+-----+-----+
| 1 | 2 | 3 |
+-----+-----+-----+
| 4 | 5 | 6 |
+-----+-----+-----+
| 7 | 8 | 9 |
+-----+-----+-----+
computer's turn :
+-----+-----+-----+
| 1 | 2 | 3 |
+-----+-----+-----+
| 4 | X | 6 |
+-----+-----+-----+
| 7 | 8 | 9 |
+-----+-----+-----+
Your turn :
enter a number on the board :3
Your turn :
enter a number on the board :4
+-----+-----+-----+
| 1 | 2 | 0 |
+-----+-----+-----+
| 0 | X | 6 |
+-----+-----+-----+
| 7 | 8 | X |
+-----+-----+-----+
computer's turn :
+-----+-----+-----+
| X | 2 | 0 |
+-----+-----+-----+
| 0 | X | 6 |
+-----+-----+-----+
| 7 | 8 | X |
+-----+-----+-----+
winner is X

⇒ 8-Puzzle problem using A* Algorithm:

1) Node (data, level): initialize node with puzzle
   Start & level

2) def puzzle():
   - accept (): ← Accept start & goal state.
   - f (start, goal) ← calculate [f = h+g]
   - process () ← process A* algorithm

3) Heuristic
   h (start, goal): Manhanlton dist. b/w current & goal state.

4) A* algorithm :-
   1) Initialize start node & add to open list
   2) Loop until & goal state come.
      → Select node have lowest 'f' from open list
      → Display state & check goal
      → generate child node & calculate f
      → And current node to closed list (→ visited) & remove from open list (→ yet to explore)
      → Explore list by 'f'
   3) Display find move

5) · Create puzzle instance
   · call process for the execute.

```
if (mats[i][j] and
    (mats[i][j] != final[i][j]):
    count += 2

return count

def newNodes(mats, empty, new_empty, levels, Parent
             final) -> nodes:

    new_mats = copy.deepcopy(mats)

    x1 = empty_posi[0]
    y1 = empty posi[1]
    x2 = new posi[0]
    y2 = new posi[1]
    new mats[x1][y1], new mats[x1][y1] = new_mats[x2][y2]
                                         new mats[x1][y1]
    costs = calculate cost(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty, cost
                      levels)

    return new_nodes

def printMatrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d" %(mats[i][j], end = " "))
        print()

child = newNodes(minimum mats)

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

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

#Code

```
import copy
from heapq import heappush, heappop
n=5
rows = [1,0,-1,0]
cols = [0,-1,0,1]
class priorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, key):
        heappush(self.heap, key)

    def pop(self):
        return heappop(self.heap)

    def empty(self):
        if not self.heap:
            return True
        else
            False:

class node:
    def __init__(self, parent, mats, empty, level):
            self.parent = parent
            self.mats = mats
            self.empty = empty
            self.costs = costs
            self.level = level

    def calculate costs(mats, final) -> int:
            cost = 0
                for i in range(n):
                    for j in range(n):
```

empty_tile_posi = [1,2]

solve (initial, empty_tile, final)

o/p

1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 6

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4

Enter the start state matrix

1 2 3
4 5 6

Enter the goal state matrix

1 2 3 4
5 6

|
|
\'/

1 2 3 4
5 6
_ 7 8

|
|
\'/

1 2 3 4
5 6
7 _ 8

|
|
\'/

1 2 3
4 5 6
7 8

* Iterative deepening

```
def id_dfs(puzzle, goal, get_moves):
    import itertools
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move],
                                 depth - 1)
                if next_route:
                    return next_route
    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route

def possible_moves(state)
    b = state.index(0)
    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 = []
    for i in d:
        pos_moves.append(gen(state,i,b))
    return pos_moves
```

```
def generate(stat, m, b):
    temp = stat.copy()
    if m == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if m == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if m == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-2]
    if m == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+2]
        return temp.

initial = [1,2,3,0,4,5,7,5,8]
goal = [1,2,3,4,5,6,7,8,0]
route = id_dfs(initial, goal, possible_moves)

if route:
    print("Success!! It is possible to solve").
    print("Path:", route)
else:
    print("Failed to find")
```

O/P

Success!! It is possible to solve 8 puzzle problem

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

Output:

```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

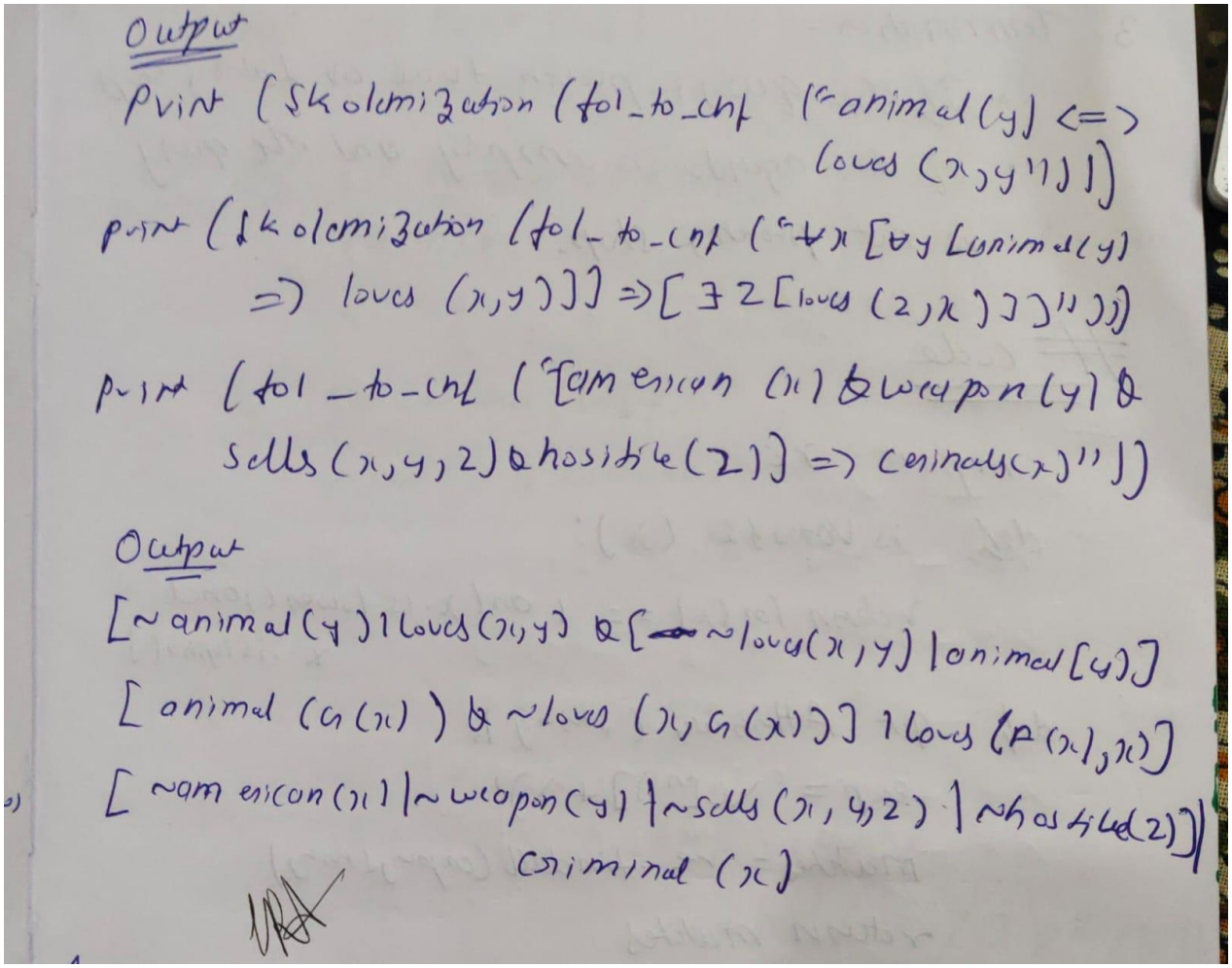
```
[ ] Start coding or generate with AI.
```

Convert FOL into CNF program :-

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

def getPredicates(string):
    expr = '[a-zA-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 == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    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])

ANTS.pop(0)}({a1[0] if len(a1) else match[1]}))}
    return statement.

import re
def fol_to_cnf(fol):
    statement = fol.replace("<=>", "-")
    while '-' in statement:
        i = statement.index('-')
        new_statment = '[' + statement[:i] + '=>' +
            statment[i+1:] + ']&[' + statement
            [i+1:] + '=>' + statment[:i] + ']'
        statment = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^\]]+)\]'
    statement = re.findall(expr, statement)
```

```
for i, s in enumerate (statements):
    if '[' in s & ']' not in s:
        statements[i] += ']'

for s in statements:
    statement = statement.replace(s, tol_to_ch(s))

new_statement
while '~∀' in statement:
    i = statement.index('~∀')
    statement = list(statement)
    statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'

while '~∃' in statement:
    i = statement.index('~∃')
    s = list(statement)
    s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
    statement = ''.join(s)

for s in statements:
    statement = statement.replace(s, tol_to_ch(s))

exp = '~\[[^]]+\]'

statements = re.findall(expr, statement)

for s in statements:
    statement = statement.replace(s, DeMorgan(s))

return statement
```



Output:

```
In [3]: print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
        print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
        print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

```
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

Aim:- Creak a knowledge bar consisting of first order Logic statements and prove the given query using forward reasoning.

Algorithm

1) Initialize the Agenda:
   - Add the query to the agend.
2) while the Agenda is not empty:
   a. pop a statement from the agenda
   b. if the statment is already known, continue to the next statment.
   c. If the statment is a fact, add it to the set of known facts.
   d. If the statment is a rule, apply the rule to generate new statments & add them to the agenda.

3. Termination:-

- If the query is proved true or false, stop
- If the agenda is empty and the query is not proved, stop.

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]
```

```
    def getResult(self):
        return self.result

    def getConstant(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]

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])

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 display(self):
    print("All facts: ")
    for fact in set([st.expression for fact in
                     self.facts])):
        print(f"\t{i+1}. {fact}")
```

O/P

```
kb_ = KB()
kb_.tell('king(x) & greedy(x) => evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

Query evil(x):

1. evil(John)

Output:

In [ ]:

```
kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

```
Querying evil(x):
        1. evil(John)
```

29/12/23

Lab-7

① Knowledge Base Entailment

→ 'KB' logically entails 'S' If all the models that evaluate KB to True also evaluate S to True.

→ Denoted by KB |= S

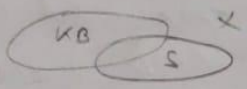
eg:- KB: A ∨ B, ¬C ∨ A S: A ∧ C

eg:-

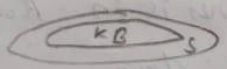
A	B	C	KB	S
F	F	F	F	F
F	F	T	F	F
F	T	F	T	F
F	T	T	F	F
T	F	F	T	F
T	F	T	T	T
T	T	F	T	F
T	T	T	T	T

KB |= S

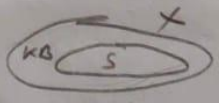
→ KB |= S



KB ⊭ S



KB |= S



KB ⊭ S

Algo

① So, first will define knowledge base query in code

② then we will define the query_entails in code to satisfies the conditions

③ So, the KB entails 'S', if and only if the KB is 'true' & 'S' is also true.

④ As we have seen in the Truth table & Logically graph.

⑤ So, base on this we say the 'KB' is Logically entails S' or not.

~~from~~

code

```
from sympy import Symbols, And, Not, Implies, Satisfiable
def create_KB():
    p = Symbols('p')
    q = Symbols('q')
    r = Symbols('r')                    } define

    knowledge_base = And(
        Implies(p, q),                  } p => q
        Implies(q, r)                   } q => r
        Not(r)                          } ~r
    )                                   } KB
    return KB

def query_entails(KB, query):                condition for entails 'S'
    entailment = satisfiable(And(knowledge_base, Not(query)))
    return not entailment

if __name__ == "__main__":
    KB = create_KB()
    query = Symbols('not p')
    result = query_entails(KB, query)        } computation

    print("Knowledge Base:", KB)
    print("query:", query)
    print("query entails KB:", result)
```

O/P

KB : ~r & (Implies(P, q)) & (Implies(q, r))
Query: (not, P)
Query entails KB: False.

Output:

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

```
[ ]  Start coding or generate with AI.
```

② Knowledge Base Resolution

→ A Sentence is in conjunctive Normal form (CNF) if it is a conjuction of clause, Each clause being a disjunction literals

$$\underbrace{(A \vee B)}_{\text{clause}} \wedge \underbrace{(C \vee D \vee \neg E)}_{\text{clause}} \wedge \underbrace{(E \vee G)}_{\text{clause}}$$

eg:- CNF conversion

$a \Leftrightarrow B \quad (a \Rightarrow B) \wedge (B \Rightarrow a)$

$a \Rightarrow B \quad \neg a \vee B$

$\neg(a \wedge B) \quad \neg a \vee \neg B$

$\neg(a \vee B) \quad \neg a \wedge \neg B$

$(A \wedge B) \Rightarrow C$

$\neg(A \wedge B) \vee C$

$(\neg A \vee \neg B) \vee C$

$(\neg A \vee \neg B \vee C)$

⇒ If a literal appears in one clause and its Negation in the other one, the two clauses can be merge & that literal can be discarded

⇒ we found an empty clause (KB does entails)

code

```
def negate_literal(literal)
    if literal[0] == '~':
        return literal[1:]
    else
        return '~' + literal

def resolve(c1, c2):
    resolve_clause = set(c1) | set(c2)
```

```
for literal in (1):
    if negate_literal (literal) in (2):
        resolved_clause.remove (literal)
        resolved_clause.remove (negate_
                                literal (literal))
    return tuple (resolved_clause)

def resolution (knowledge_base):
while true:
    new clause = set()
    for i, c1 in enumerate (knowledge_base)
        for j, c2 in enumerate (knowledge base)
            if i != j:
                new clause = resolve (c1, c2)
            if len (new_clause) == 0 & new_
                clause not knowledge_
                base:
            new clause.add (new_clause)
    if not new clause break.
    Knowledge_base != new_clause
return knowledge_base
if __name__ == "__main__":
    KB = {('p', 'q'), ('~p', 'r'), ('~r', '~r')}
    print ("Resolved KB", result)
```

O/P

Enter statement = ~negation

The statement not entailed by knowledge_base.

```
rules = 'PvQ ~PvR ~QvR' #P=>Q, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
main(rules, goal)
```

```
Step    |Clause |Derivation
-------------------------------
 1.     | PvQ   | Given.
 2.     | ~PvR  | Given.
 3.     | ~QvR  | Given.
 4.     | ~R    | Negated conclusion.
 5.     | QvR   | Resolved from PvQ and ~PvR.
 6.     | PvR   | Resolved from PvQ and ~QvR.
 7.     | ~P    | Resolved from ~PvR and ~R.
 8.     | ~Q    | Resolved from ~QvR and ~R.
 9.     | Q     | Resolved from ~R and QvR.
 10.    | P     | Resolved from ~R and PvR.
 11.    | R     | Resolved from QvR and ~Q.
 12.    |       | Resolved R and ~R to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

24/11/23 Lab-4 VKAze-11-23

8-puzzle Problem

Algo

We will use ~~branch and bound~~ this technique

① 3 types of nodes involved in branch and bound

a. Live node is a node whose children node are currently being explored

b. E-node is a node currently being expanded

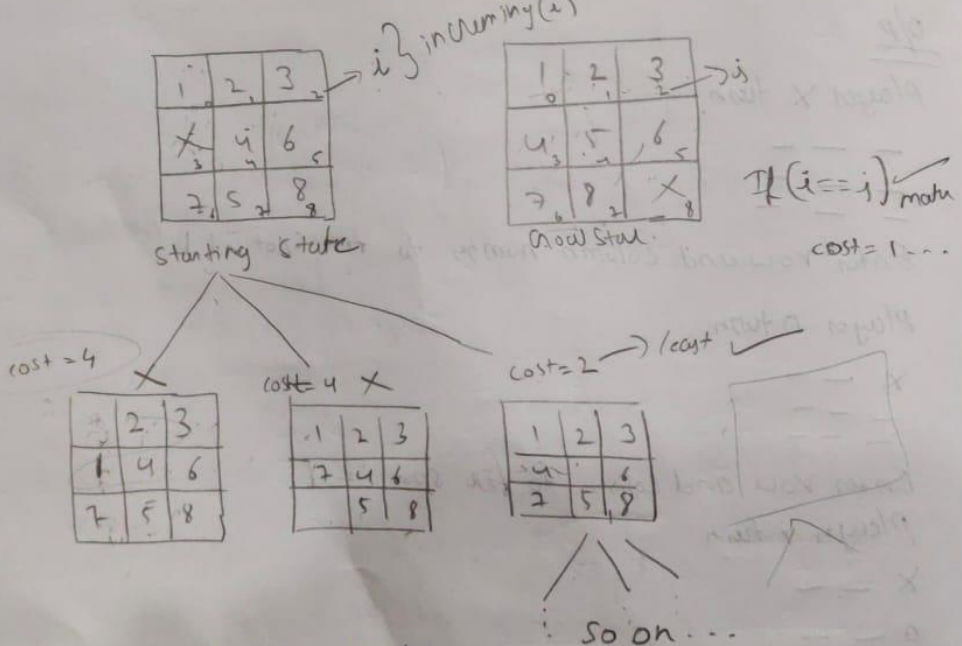
c. Dead node is a node that is not to be expanded

② cost function; use to determine the next E-node → have least cost

$$c(x) + g(x) + h(x)$$

where $g(x)$ = cost of reaching the current node from the root.

$h(x)$ = cost of reaching an answer node from x



③ Once we get the match for starting state to Goal state, we stop the function and Agent will solve the 8-puzzle problem

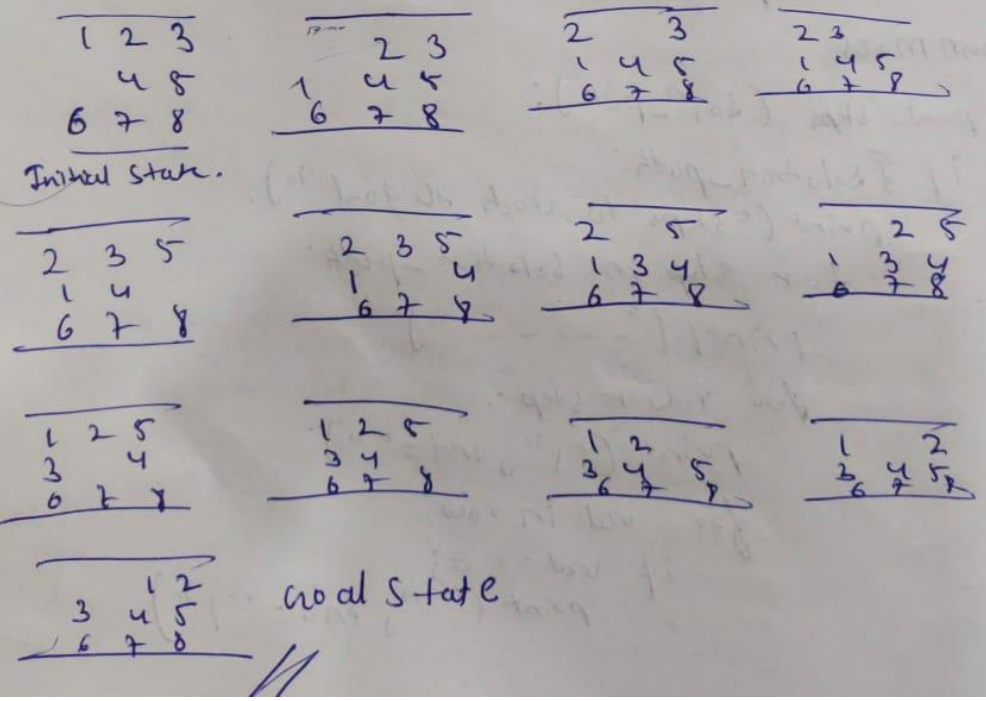
code

```
from collections import deque
def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
            return i,j
def generate_moves(board):
    moves = []
    blank_row, blank_col = find_blank(board)
    possible_moves = [
        (1,0), (-1,0), (0,1), (0,-1)
    ]
    for dr,dc in possible_moves:
        new_row, new_col = blank_row + dr, blank_col + dc
        if 0<=new_row<3 and 0<=new_col<3:
            new_board = [row[:] for row in board]
            new_board[blank_row][blank_col], new_board[new_
            row][new_col] = new_board[new_row][new_col],
            new_board[blank_row][blank_col]
            moves.append(new_board)
    return moves
def print_steps(solution_path):
    if solution_path:
        print("steps to reach the goal:")
        for step in solution_path:
            print("------")
            for row in step:
                print("|", end="")
                for val in row:
                    if val == 0:
                        print(" ", end="|")
```

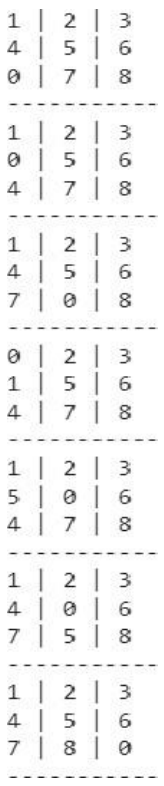
```
            else:
                print(val, end=" ")
        print()
    print("- - - - - -")
    print()
else:
    print("No solution exists!")

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

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

solution_path = solve_puzzle(initial, goal)
print_steps(solution_path)
```

O/P

Steps to reach the goal :



Success


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

Lab :- 8 19/1/23

Aim :- Implement unification in first order Logic

Eg :- Knows (John, x) Knows (John, Jane)

{x / Jane}

Step 1 :- If term 1 or term 2 is a variable or constant then:

a) Term 1 or Term 2 are identical return NIL.

b) Else if term 1 is available

if term 1 occurs in term 2

return Fail

else

return {(term 2 / term 1)}

c) else if term 2 is a variable

if term 2 occurs in term 1

return Fail

else

return {(term 1 / term 2)}

d) else return Fail

Step 2 :- If predicate (term 1) ≠ Predicate (term 2)

return Fail

Step 3 :- Number of arguments ≠

return Fail

Step 4 :- Set (subset) to NIL

Step 5 :- For i = 1 to the No. of elements in term 1

a) call Unify (ith term 1, ith term 2)

put result into S

Code for unification :-

```
impor re
def getAttributes (expression):
    expression = expression.split ("(") [1:]
    expression = "(".join (expression)
    expression = expression [:-2]
    expression = re.split ("?<! (.), (?! .\))", expression)
    return expression
def getInitialPredicate (expression):
    return expression.split ("(")[0]
def isConstant (char):
    return char.isupper () & len (char) == 1
def isVariable (char):
    return char.islower () & len (char) == 1
```

```
def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True

def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]

def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:])
    return newExpression
```

```
def unify(exp1, exp2):
    if exp1 == exp2
        return[]
    if isConstant(exp1) & isConstant(exp2):
        if exp1 != exp2:
            return False.
    if isConstant(exp2):
        return [(exp1,exp2)]
    if isConstant(exp2):
        return [(exp2,exp1)]

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

    if getInitialPredicate(exp1) != getInitialPredicate(exp2)
    Print("Predicates do not match. cannot be
                                    unified")
    return False

attribute Count 1 = len(getAttributes(exp1))
attribute count 2 = len(getAttribute(exp2))
if attribute count 1 != attribute count 2:
        return False
```

```
if not initial_substitution
    return False
if attribute_count_1 == 2:
    return initial_substitution

tail_1 = get_Remaining_Part(exp1)
tail_2 = get_Remaining_Part(exp2)

if initial_substitution != []:
    tail_1 = apply(tail_1, initial_substitution)
    tail_1 = apply(tail_1, initial_substitution)
    tail_2 = apply(tail_2, initial_substitution)

remaining_substitution = unify(tail_1, tail_2)
if not remaining_substitution
        return False.

initial_substitution.extend(remaining_substitution)
return initial_substitution
```

O/p

```
exp1 = "knows(A,x)"
exp2 = "knows(y, mother(y))"
substitution = unify(exp1, exp2)
print("Substitutions:")
print(substitution)
```

Substitutions :-

```
[('A', 'y'), ('mother(y)', 'x')]
```

Output:

```
exp1 = "knows(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('X', 'Richard')]
```

```
[ ] exp1 = "knows(A,x)"
    exp2 = "knows(y,mother(y))"
    substitutions = unify(exp1, exp2)
    print("Substitutions:")
    print(substitutions)
```

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

Lab-6

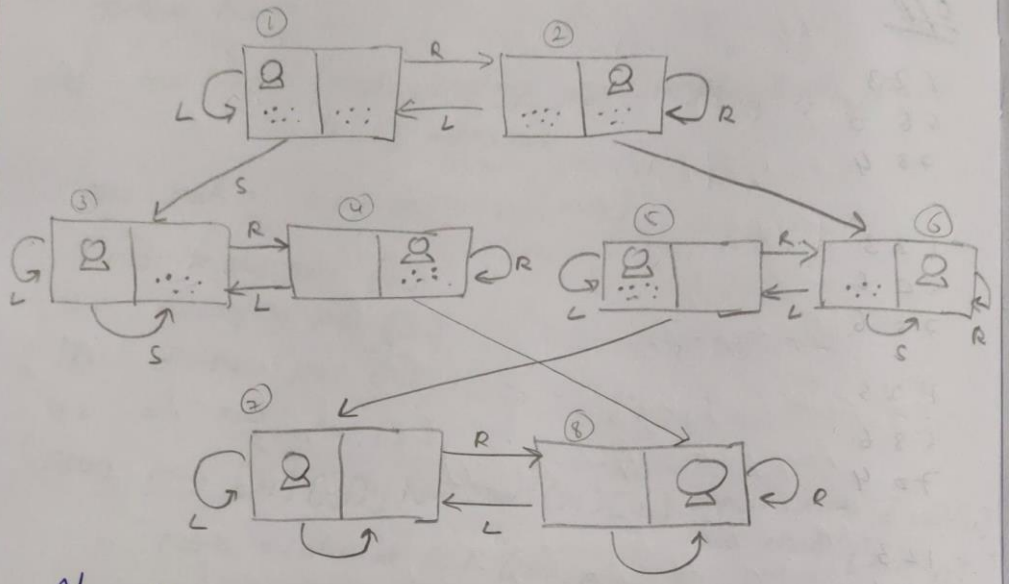
22/12/23

→ Implement the vacuum cleaning agent

$(n.2^n)$ => n = number of room (2)

$2 \cdot 2^2 = 8$ state

22-12-23



Algorithm

1) init (self, room1, room2): —> constructor
   → initialize the cleaning agent with 2 rooms & set the current room

2) clean_room (self): initiates the cleanly process of each room, it call the 'clean_room function' to each room & the it calls the 'move_to_next_function'.

3) def clean_room (self, room):

```
if (room == 'clean'):
    return (clean)
else if (room == 'dirty'):
    initiates the cleaning process
    room == 'clean'
    return 'clean'
```

4) move_to_next:

count = 0:

→ Switch the current room to adjacent room.

if count $n.2^n \leq 2$:

end

else:
move_to_new_room.

Code

```
class VacuumCleaner:
    def __init__(self, initial_location):
        self.location = initial_location

    def move_left(self):
        print("Moving Left")
        self.location = 'A'

    def move_right(self):
        print("Moving right")
        self.location = 'B'

    def suck(self, room):
        print("Sucking dirt in Room{room}")
        return 'clean'

def simulate_cleaning():
    initial_vacuum_location = ip("Enter initial location
        of the vacuum cleaner (A/B):").upper()
    Vacuum = VacuumCleaner(initial_vacuum_location)
    room_A_state = I/P("Enter state for Room A
        (clean/dirty):").lower()
    room_B_state = I/P("Enter state for Room B
        (clean/dirty):").lower()

    rooms = {
        'A': room_A_state,
        'B': room_B_state,
    }

    print("\n initial state:")
    print("Vacuum cleaner is in Room
    {vacuum.location}")
```

```
print(f"Room A: {rooms['A']}")
print(f"Room B: {rooms['B']}\n")

if rooms['A'] == 'clean' and rooms['B'] == 'clean':
    print("Both rooms are already clean. No
          cleaning needed.")
else:
    print("Starting the cleaning process")

    current_room = vacuum.location
    cleaned_room = vacuum.suck(current_room)

    if cleaned_room == 'clean':
        rooms[current_room] = 'clean'

    if current_room == 'A':
        vacuum.move_right()
        current_room = 'B'
    else:
        vacuum.move_left()
        current_room = 'A'

clean_room = vacuum.suck(current_room)
    if cleaned_room == 'clean':
        rooms[current_room] = 'clean'

    print("\nCleaning completed.")
    print("Final state:")
    print(f"Room A: {rooms['A']}")
    print(f"Room B: {rooms['B']}")

simulate_cleaning()
```

Op

Enter initial location of vacum cleaner (A/B): A
Enter state for Room A (clean/dirty): dirty
Enter state for Room B (clean/dirty : dirty

Initial state :

Vacum cleaner is in Room A
Room A: Dirty
Room B : dirty

Starting the cleaning process
Sucking dirt in Room A
moving right
Sucking dirt in Room B

cleaning completed
Finial State :
vacum cleaner is in Room B
Room A : clean
Room B : clean.

Output:

```
0 indicates clean and 1 indicates dirty
Enter Location of VacuumB
Enter status of B0
Enter status of other room1
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

```
[ ]  Start coding or generate with AI.
```