INDEX

Name Naaid Qaj (1BM21CS079) Std Sec reg slip

Roll No. Subject Ai Lab Copy (SB) School/College

School/College Tel. No. Parents Tel. No. 8025079711

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1	16/11/23	Lab 1 (Prattle Question)		
2	17/11/23	Lab 2 (Tic Tac Toe)		
3	24/11/23	Lab 3 (8-puzzle (BFS))		
4	8/12/23	Lab 4 (8-puzzle (IDFS) & A*)		
5	22/12/23	Lab 5 (Vacuum cleaner, KB & KBR)		
6	19/1/24	Lab 6 (Unification, FOL to CNF, Forward Chaining)		
7				

29/4

17/11/23

Program 1: Implement Tic Tac Toe

```
board = [ ]
for i in range (9):
    board.append (' ')

def printing OX ( letter , pos )
    board (pos) = letter

def empty (pos)
    if ( board (pos) == ' '):
        return True
else
        return False

# printy Tic Tac Toe

def printboard ( board )
    for i in range (3)
        print ( board [3 * i] + '|' + board (3*i+1) +
        '|' + board [3*i+2]
        if ( i <= 1 )
            print ( '__|__|__')

# Winning
def winning (board, l)
    if ( board [0] == l and board [1] = l and board (2) =l
    or ( board [3] == l and board [4] == l and board (5) = l
    or (board [6] = l and board (7) = l and board (8] == l)
    or ( board (0] <= l and board [3] = = l and board (6])
or (board (1] == l and board [4] and board (7)) or
    board [2] = l and board [5] = l and board (8] == l)
or (board [0] = = l and board (4] = = l and board (8] = = l)
Or
```

(board[2] == 1 and board[4] == 1 and board[6] == 1)

# initially
def Moves():
run = True
while (run):
move = int(input("Select position for X:"))
try:
if (move >= 0 and move < 9):
if emptyloc(move):
run = False
~~board~~ printing OX('X', move)
else:
print("Space Occupied")
else:
print("Invalid position")
except:
print("Please type number again")
print board(board)
if (win(board, 'X') == True):
return "winner is X"
if (boardFull(board)):
return "no winner"
run: True
while (run):
move = int(input("Select position for O"))
try:
if (move >= 0 and move < 9):
if (emptyloc(move)):
run False
printing OX('O', move)
else:
print("Space Occupied")

```
except:
    print("Please type number again")
if (int(board[0]) == 'null')
    return "user is 0"

def BoardFull(board):
    if " " in board:
        return False
    else:
        return True
if num != 7:
    while (!BoardFull(board)):
        Move().
```

Algorithm

① Take a list named as board for elements, keep it empty

② Defining a function putting OX to put the letters in the position

③ Check if the board is empty i.e the position, then return true.

④ Make a function print board to print the board

⑤ Define a function to check for all the winning conditions

⑥ Define the function moves to input
   i. → X → check if board is empty with empty function
   ii. Put it on the board ~~to~~ put it
   - Check the winning condition winning ( )
   - Check if the board is full

⑦ Check for O → check for empty
   put in function by putting

⑧ Check again if the board is full

⑨ Main function to call Moves ( ) till board becomes full.

17/11

```
Would you like to go first or second? (1/2)
1
   |   |
---+---+---
   |   |
---+---+---
   |   |


Player move: (0-8)
4
   |   |
---+---+---
   | O |
---+---+---
   |   |


 X |   |
---+---+---
   | O |
---+---+---
   |   |


Player move: (0-8)
2
 X |   | O
---+---+---
   | O |
---+---+---
   |   |


 X |   | O
---+---+---
   | O |
---+---+---
 X |   |


Player move: (0-8)
3
 X |   | O
---+---+---
 O | O |
---+---+---
 X |   |
```

```
 X |   | O
---+---+---
 O | O |
---+---+---
 X |   |


 X |   | O
---+---+---
 O | O | X
---+---+---
 X |   |


Player move: (0-8)
7
 X |   | O
---+---+---
 O | O | X
---+---+---
 X | O |


 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O |


Player move: (0-8)
8
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | O
The game was a draw.
Continue playing? (y/n)
n
```

- 8 puzzle problem using BFS

```
from queue import Queue

def print_board(board)
    for row in board
        print(row)
    print()

def get_blank_position(board):
    for i in range
```

Algorithm :-

1) First of all, we consider a queue i.e import it.
   - from queue import queue

2) Create a board with the help of function & print rows
```
def p_board(board):
    for x in board:
        print(x)
    print()
```

3) We check empty place
```
def i in range(3):
    def get_blank_position(board):
        for i in range(3):
            for j in range(3):
                if board[i][j] == 0:
```

return 1,1

4/ To check if move is valid
def valid-move(x, y):
return $0 <= x < 3$ and $0 <= y < 3$

5/ funtion for swapping
def swap(board, x1, y1, x2, y2):
board[x1][y1], board[x2][y2] =
board[x2][y2], board[x1][y1]

6/ Main function
# creating a set for the visited state
visited = set()
# A queue for BFS traversal
q = queue()
If queue is not empty, add to visited board
if current state is equal to goal state
return moves
- get the blank position
we can have the possible moves is
moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
for move in moves:
new-x, new-y = blank x + move(0)
blank-y + move(1)

- check if the move is valid, swap
~~note~~ a new state.

- if board not visited, add to queue,
add to visited, increase moves.

- Code:

```
from queue import Queue

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

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def swap(board, x1, y1, x2, y2):
    board[x1][y1], board[x2][y2] =
        board[x2][y2], board[x1][y1]

def solve_puzzle(initial_state, goal_state):
    visited = set()
    queue = Queue()
    queue.put((initial_state, 0))

while not queue.empty():
    current_state, move_count = queue.get()
    visited.add(tuple(map(tuple, current
    if current_state == goal_state:
        return move_count

    blank_x, blank_y = get_blank_pos(current_state

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

    for move in moves:
```

new_x, new_y = blank_x + move[0], blank_y + move[1]

if is_valid_move(new_x, new_y):
new_state = [row.copy() for row in current_state]
swap(new_state, blank_x, blank_y, new_x, new_y)

if tuple(map(tuple, new_state)) not in visited:
queue.put((new_state, move_count + 1))
visited.add(tuple(map(tuple, new_state)))

return None

initial_state = [

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

]

goal_state = [
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]
]
move_count = solve_puzzle(initial_state, goal_state)

if move_count is not None:
print("Minimum number of moves:", move_count)
else:

```
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

success
```

8 Puzzle using IDDFS

- Algorithm

→ defining the goal state
goal_state = [1, 2, 3]
[8, 0, 4]
[7, 6, 5]
return state == goal_state

def get_actions(state):
→ define the possible actions to move the blank tile
actions=[] → create an empty list
blank_row, blank_col = find_blank(state)

→ possible action
if blank_row > 0:
action.append("up")
blank_row < 2:
action.append("down")
blank_col > 0:
action.append("left")
blank_col < 2:
actions.append("right")

return algor

→ If find_blank(state):
Find the row and column index of the blank tile (0)
For i in range(3):
for j in range(3)
if state[i][j] == 0:
return i, j

Apply the action to move the blank tile
def apply_action (state, action):
new_state = [row.copy() for row in state]
blank_row, blank_col = find_blank (state)

if action == "up":
new_state [blank_row] [blank_col], new_state [blank_row] [blank_col] = new_state [blank_row - 1] [blank_col], 0

elif action == "down":
new_state [blank_row] [blank_col], new_state [blank_row + 1] [blank_col] = new_state [blank_row + 1] [blank_col], 0

elif action == "left":
new_state [blank_row] [blank_col], new_state [blank_row] [blank_col - 1] = new_state [blank_row] [blank_col - 1], 0

elif action = "right":
new_state [blank_row] [blank_col], new_state [blank_row] [blank_col + 1] =
new_state [blank_row] [blank_col + 1] = 0

return new_state

def depth_limited_search (state, parent=None, action=None, depth=0, depth_limit=0):

- If the state is the goal state:
return the action
elif depth = depth_limit:
return none

else get possible actions
new_state ← application (state, a)
result ← d_l_s (new_state, state a, depth+1,
depth_limit)

if result ≠ not None:
return result
return ← None

IDDFS
D_l = 0
while True:
result = d_l_s (initial_state, d_l = d_l)
if result ≠ None or None:
return result
d_l = +1

Prof 8/12

Code:

```
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')
    for move
```

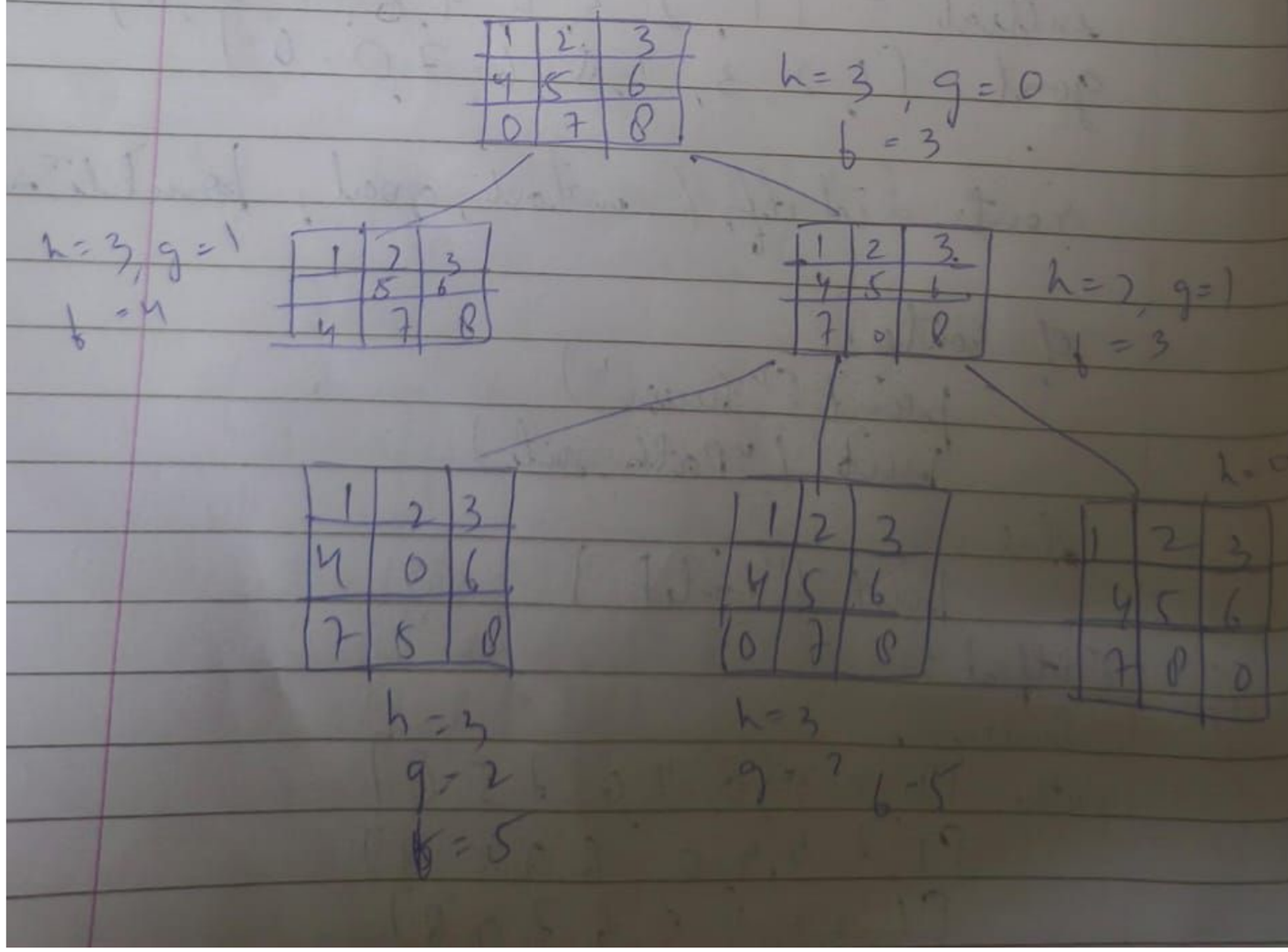
```
for i in d:
    for - moves. append (generate (state, i, b))
return for_moves

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

return temp

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

route = id dfs ( initial, goal, possible_moves)

if route:
    print ("Success")
    print ("Path", route)
else:
    print ("Failed")
```

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]]
```

* 8 puzzle problem using A*

(1) Create initial and goal state for problem.
(2) We calculate value at each step
f value = h value + path cost
(no. of steps)
(3) Choose the direction where f value is least.
(4) We reach to the goal state when f value = 0.
(5) $f(h) = h(x) + g(x)$
heuristic, Goal st, path cost
6) Now check the heuristic cost of each of the child node and expand only those who have least heuristic value.
7) This will go on until we get h and g become answer



Code

```
Class Node
def __init__(self, data, level, fvalue):
    self.data = data
    self.level = level
    self.fval = fval
def generate_child(self)
    x, y = self.find(self.data, '-1')
    valllist = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]

Children = []
for i in valllist:
child = self.shuffle(self.data, x, y, i[0], i[1])
if child is not None:
    childNode = Node(child, self, level+1, 0)
    Children.append(childNode)
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
  0.append(ij)
  temp.append(0)
return temp
class Puzzle
  def __init__(self, size)
  self.m = size
  self.n = size
  self.open()
  self.goal()
def solve(self)
  temp = []
for i in range(0, self.m)
  temp = input().split(" ")
temp.append(temp)
return temp
def f(self, start, goal)
return self.h(start, goal)
  def h(self, start, goal):
temp = 0
  for i in range(0, self.m)
    for j in range(0, self.m):
      if start[i][j] != goal[i][j]
                & start[i][j] != '_':
        temp += 1 return temp
```

O/P

Enter the start state

1 2 3
4 5 6
- 7 8

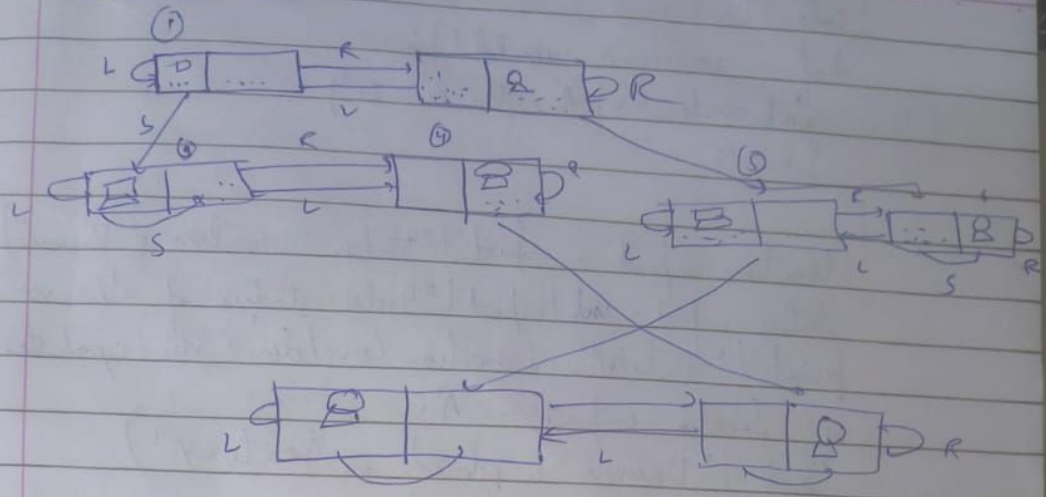
Enter the goal state

1 2 3
4 5 6
7 8 -

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

```
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 _




  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

1 2 3
4 5 6
7 8 _
```

Vacuum Cleaner



→ Initialise a cleaning agent with 2 rooms & set the current room
init (self, room1, room2)

→ clean_room (self) : initiates the cleaning policies of each room, it calls the clean room function

→ clean_room (room)
if (room == 'clean')
return ('clean')
elif (room == 'dirty')
# initiate cleaning
room = = 'clean'
return ('clean')

→ → If move_to_next_room:
count = 0;
Initially the current room is the room
if count = = 2
then move to next room

code:

```
def vacuum_world():
    goal_state = {'A': 0, 'B': 0}
    cost = 0

    location_input = input("Enter location of Vacuum")
    status_input = int(input("Enter status of other room"))
    print("Initial location condition" + str(goal_state))
    if location_input == 'A':
        print("Vacuum is placed in location A")
        if status_input == 1:
            print("location A in 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 in dirty")
        print("Moving right to 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("cost" + str(cost))
        print("location B is already clean")

    if status_input == 0:
        print("location A is already clean")
    if status_input_complement == 1:
        print("location B is Dirty")
```

```
print("Moving RIGHT to 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(cost)
print("location B is already clean")
if status_input_complement == 1 # if A is dirty
print("location A is Dirty")
print("moving LEFT to location A")
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()
```

```
0 indicates clean and 1 indicates dirty
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
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.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Knowledge Based entailment

Algorithm

1. Creating a knowledge base
   define symbols
   p = Symbols('p')
   q = Symbols('q')
   r = Symbols('r')

   k_b = And(
   Implies(p, q), # If p then q
   Implies(q, r), # If q then r
   Not(r)
)
   return knowledge_base

Proceed

def query_entails(knowledge_base, query):
entailment = satisfiable(And(knowledge_base, Not(query)))
return not entailment

Code:

```
from sympy import symbols, And, Not, Implies
    satisfiable

def create_knowledge_base():
    P = symbols('P')
    q = symbols('q')
    r = symbols('r')

knowledge_base = And(
                    Implies(P, q),
                    Implies(q, r),
                    Not(r)
                )
return knowledge_base

def query_entails(knowledge_base, query):
    entailment = satisfiable(And(knowledge_base, Not(query)))
return not entailment

if __name__ == "__main__":
    KB = create_knowledge_base()
    query = symbols('P')
    result = query_entails(KB, query)
    print("Knowledge Base:", KB)
    print("Query:", query)
```

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

- knowledge based Resolution

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

def resolve(c1, c2):
    resolved_clause = set(c1) | set(c2)

    for literal in c1:
        if negate_literal(literal) in c2:
            resolved_clause = resolve(literal)
            resolved_clause.remove(negate_literal
                                   (literal))

def resolution(knowledge_base):

    while True:
        new_clauses = 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
                        knowledge_base = new_clause.add(new_
                if i != j:
                    not new_clause
                    break
        knowledge_base != new_clause
        return knowledge_base
```

if __name__ == "__main__":
    kb = [['P', 'Q'], ['~P', 'R'], ['~Q', 'R']]
    result = resolution(kb)
    print("Original KB", kb)
    print("Resolved KB", result)

O/P:

enter statement = negation
The statement not entailed by knowledge base

Step	Clause	Derivation
1.	Rv~P	Given.
2.	Rv~Q	Given.
3.	~RvP	Given.
4.	~RvQ	Given.
5.	~R	Negated conclusion.
6.		Resolved Rv~P and ~RvP to Rv~R, which is in turn null.

A contradiction is found when ~R is assumed as true. Hence, R is true.

Unification

T_1 $\bar{\pi}$ T_2

Eg: Knows [John, x]) Knows (John, Jane)

S: {x / Jane}

Step 1: If term1 or term2 is a variable or constant, then:

a) term1 or term2 are identical
return NIL

b) Else if term1 is a variable
if term1 occurs in term2
return FAIL
else return {(term2) term1}

c) Else if term2 is a variable
if term2 occurs in term1
return FAIL
else
return {(term1 / term2)}

d) else return FAIL

Step 2: If predicate (term1) ≠ predicate (term2)
return FAIL

Step 3: Number of arguments ≠
return FAIL

Step 4: set (SUBST) ← NIL
For i = 1 to the number of elements in term1
a) call unify (ith term1, ith term2)
put result into S
b) S = FAIL
return FAIL

c) If S ≠ NIL
   a. Apply S to the remainder of both L' and L2
   b. SUBST = APPEND(S, SUBST)
d) Return SUBST

- Unification Code

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = re.split("(?<=\)),(?!.*\))", expression)
    return expression

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

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

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

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

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

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

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

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

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

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp1):
        return [(exp1, exp2)]

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

    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
    else:
```

return [(exp2, exp2)]

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

if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicate do not match. Can not be unified")
    return False

attributeCount1 = len(getAttributes(exp1))

attributeCount2 = len(getAttributes(exp2))

if attributeCount1 != attributeCount2:
    return False

head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)
initialSubstitution = unify(head1, head2)
if not InitialSubstitution:
    return False
if attributeCount1 == 1:
    return initialSubstitution

tail1 = getRemainingPart(exp1)
tail2 = getRemainingPart(exp2)

remainingSubstitution = unify(tail1, tail2)

if not remaining Substitution :
return False

initial Substitution, initial (remaining Substitution
return initial Substitution

exp 1 = "knows(A, x)"
exp 2 = "knows(y, y)"
substitution = unify (exp 1, exp 2)
print("Substitution :")
print (Substitutions)

Substitutions :
[('x', 'y'), ('y', 'x')]

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

FOL to CNF conversion

Step 1: Remove implication
replace "Or(Not(1 and (2)4), (and (3)))"

Step 2: apply demorgan law
formula replace ('Not(And', 'Or(Not)') replace
('And', 'Or(Not')

Step 3: Substitute quantifiers
formula.replace("ForAll(', 'All('), replace
('Exists(', 'Or(')

Step 1: create a list of SKOLEM_CONSTANTS
Step 2: Find ∀ ?
If the attributes are lower case, replace them
with a skolem constant
remove used skolem content or function from
list
If the attributes are both lowercase and uppercase,
replace the uppercase attribute with a skolem
function.
Step 3: replace <=> with '->'
formula = or Q = (P => Q) ∧ (Q => P)

Step 4: replace => with '->'
formula = or Q = ~P ∨ Q

Step 5: Apply demorgan law
replace ~
or ~A ~B ~Q if () was present)
replace ~(
or ~P(~Q if ((was present)
replace ~∨ with '∧'

FOL to CNF Code:

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

def getPredicates(string):
    expr = '[a-zA]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).split())
    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])
for predicate in getPredicates(statement):
    attributes = getAttributes(predicate)
    if '', join(attributes) in lower():
```

```
import re

def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
while '_' in statement:
    i = statement.index('_')
    new_statement = '(' + statement[:i]
+ '=>' + statement[i+1:] + ')&(' + statement
[i-1::-1] + ')'
    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, bd_to_dis(s))

while ':-' in statement:
    i = statement.index(':-')
    bk = statement.index('(') if '(' in
statement else 0
new_statement = '~' + statement[bk:i] + '|' +
statement[i+1:]

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 = '(~(∀|∃)[^)]*)'
for s in statement:
    statement = statement.replace(s, DeMorgan(s))
return statement
```

print (Skolemization (fol_to_cnf ("criminal(x)
=> lower(x, y)")))
print (Skolemization (fol_to_cnf ("∀x[∀y[
criminal(y) => lower(x,y)]]=>[∃z[lower(z,x)]]")))
print (fol_to_cnf ("(american(x) & weapon(y)
& sells(x,y,z) & hostile(z)] => criminal(x)")))

[~ criminal(y)] lower(x,y)]] & [~ lower(x,y)] criminal(y)]
[criminal(G(x)) & ~ lower(x, G(x))]] [lower(f(x),
x)]]
[~ american(x) | ~ weapon(y) | ~ sells(x,y,z)
| ~ hostile(z)] | criminal(x)

```
~bird(x)|~fly(x)
[~bird(A)|~fly(A)]
```

Forward Chaining

Algorithm

1) Initialize the Agenda
   - Add the query to the agenda

2) While the Agenda is not empty:
   a) Pop a statement from agenda
   b) If the statement is already known, continue to next statement
   c) if it is a fact, add it to the set of known facts
   d) If the statement is a rule, apply the rule to generate new statement and add them to agenda

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-zA]+)\([^&|]+\)'
    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)

        params = getAttributes(expression)[0].strip
                 ('()').split(',')
        return [predicate,params]
```

```
def getRoot(self):
    return self.root

def getConstant(self):
    return (None if isVariable(c) else c
    for c in self.params)

def getVariables(self):
    return [c if isVariable(c) else for c
    in self.params]

def substitute(self, constants):
    c = constants.copy()
    f = f"{self.predicate}({','.join([c
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 not predicate == fact.predicate
if v:
constants (0) = fact.get(constant
new_lhs.append(fact)

predicate, attributes = getPredicate&(self.expression)
(0), tuple(getAttributes(self.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
(getResult() for f in new_lhs) else None

class KB:

def __init__(self):
self.facts = set() self.facts = set()
self.implication = set()

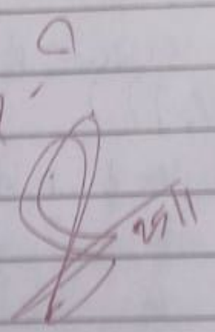
def tell(self, e)
if '=>' in e:
self.implication.add(implication(e))

else:
self.facts.add(Fact(e))
for r in self.implication:
res = r.evaluate(self.facts)

```
    if res:
        self.facts.add(res)

def query(self, e):
    facts = set(str(fact) for fact in self.facts)
    ind = 1
    print(f'Query: {e}')

def display(self):
    print("All facts:")

    for i, f in enumerate(set(str(fact)
    for fact in self.facts)):
    print(f'{i+1}: {f}')

kb = KB()
kb.tell('missile(x) => weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x, America) => hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono, America)')
kb.tell('owns(Nono, M1)')

kb.tell('missile(x) & owns(Nono, x) => sells(West, x,
kb.tell('american(x) & weapon(y) & sells(x, y, z) &
        hostile(z) => criminal(x)')
kb.query('criminal(x)')
kb.display()
```

Date __/__/__
Page

1, Ameen (Wed)
2, Seth (Went, M, None)
3, Mundu (M.)
4, Enemy (None, Ameen)
5, Council (meets)
6, weapon (M.)
7, Gun (None, M.)
8, Hotel (None)



```
Querying criminal(x):
        1. criminal(West)
All facts:
        1. enemy(Nono,America)
        2. missile(M1)
        3. sells(West,M1,Nono)
        4. weapon(M1)
        5. hostile(Nono)
        6. criminal(West)
        7. owns(Nono,M1)
        8. american(West)
```