

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



Lab REPORT on

Artificial Intelligence

Submitted by

Harsh Ghiya (1BM21CS073)

Under the Guidance of
Prof. Swarthy Sridharan
Assistant Professor, BMSCE

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Nov-2023 to Feb-2024

**B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the lab work entitled “**Artificial Intelligence**” carried out by **Harsh Ghiya (1BM21CS073)** who are bona fide students of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visveswaraiah Technological University, Belgaum during the year 2023-2024. The lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence lab (**22CS5PCAIN**) work prescribed for the said degree.

Swathi Sridharan
Assistant Professor
Dept. of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Prof.& Head
Dept. of CSE
BMSCE, Bengaluru

17/11/23

1.

Lab - 3

Write a program for Tic-Tac-Toe game.

Algo :-

- Create an empty global 3×3 array.
- Make the board of 3×3 tic-tac-toe.
- Then we will

import random

tic = [1, 2, 3, 4, 5, 6, 7, 8, 9]

def printboard(tic)

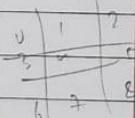
print(tic[0] + ' | ' + tic[1] + ' | ' + tic[2])

print(" - - - - -")

print(tic[3] + ' | ' + tic[4] + ' | ' + tic[5])

print(" - - - - -")

print(tic[6] + ' | ' + tic[7] + ' | ' + tic[8])



def winner(tic, pos)

if (tic[0] == tic[1] and

tic[0] == tic[2] or

tic[1] == tic[4] and tic[1] == tic[7] or

return true

elif if

tic[pos-0] == tic[pos-3] and

tic[pos-0] == tic[pos-6]

return true

elif if

tic[pos//3+1] == tic[pos//3+2] and

tic[pos//3+1] == tic[pos//3+3]

return true

return false.

```
def update_user (trc):  
    n = int(input("Enter the value on board(i)"))
```

```
    while (num not in trc):  
        num = int(input("Enter a no.on the board"))  
        trc [num-1] = z '0'
```

```
def update_comp (trc):  
    for i in trc:  
        if ('i' != 'x' and i != 'o'):  
            trc [i-1] = 'x'  
        if (winner (trc, i-1) == true):  
            return  
        else:  
            trc [i-1] = i
```

```
for i in trc:  
    if ('i' != 'x' and i != 'o'):  
        trc [i-1] = 'o'  
    if (winner (trc, i-1) == false):  
        return  
    else:  
        trc [i-1] = 1
```

```
num = random.randint (a)  
while (num not in trc):  
    num = random.randint(a)  
    trc [num-1] = z 'x'
```

Algorithm:-

- ① make a board and initialize the value.
- ② ~~make~~ ~~import~~ import the random library in order to get the random values.
- ③ Make a function for checking the winner:
 - ① check both the diagonals whether they have same values.
 - ② Then check the columns & rows if all the are same then return true.
Else return false.
- ④ Make a update user function:
In this function user can input the value. at my array tic
- ⑤ make a update-comp function
In this function comp can input its value depending upon the board.

Output:

1	2	3
4	5	6
7	8	9

~~Ques 11~~

1	2	0
0	X	6
7	8	X

1	2	3
4	X	6
7	8	9

X	2	0
0	X	6
7	8	X

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

Bafna Gold
Date: _____ Page: _____

2
Lab - 8

write an algorithm for 8-puzzle problem:-

Algorithm:-

- 1 Start with an empty queue
queue → []
- 2 Enque with initial state.
queue.append (cure)
initial.
- 3 while the queue is not empty, dequeue the first state from q the queue (cure)
def solve_puzzle (initial-state, goal-state)
queue > deque ([initial-state, []])
4. check if the current state is the target state, if it is return "success".
def checkcurrent : (initial-state, goal-state)
if (initial-state == goal-state)
print ("success");
5. Generate all possible moves from current state by determining the direction of the empty spot & considering all possible direction.
def generate (jedi current-state) :
6. moves = []
blank-row, blank-column = find-blank(board)
possible_moves = [(1,0), (-1,0), (0,1), (0,-1)]
6. for each possible moves, calculate the resulting state

7. check if the resulting state have already been visited. if it has not, enqueue the resulting state.

8. if the target stat. has not been found, continue with the next iteration with loop.

iii) ~~selected target is not feasible after selecting all possible states without expanding.~~

~~front
queue~~

⇒ 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 $0 \leq new_row \leq 3$ and $0 \leq new_col \leq 3$:

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]

new_board[blank_row][blank_col]

moves.append(new_board)

return moves

```
def sol_puzzle(initial_state, goal_state):
```

visited = set()

queue = deque([(initial_state, [])])

while queue:

current_state, path = queue.popleft()

visited.add(tuple(map(tuple, current_state)))

if current_state == goal_state:

return path

possible_moves = generate_moves(current_state)

```

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(" --- ")
    else:
        print("No solution")

```

~~premises do not relate over~~

```

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

```

```

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

```

Solution-path = solve-puzzle(initial, goal)
print-steps(solution-path)

Output:

Steps to reach the goal:

1 2 3
4 5
6 7 8

1 2 5
3 4
6 7 8

2 3
1 4 5
6 7 8

1 2
3 4 5
6 7 8

2 3
1 4 5
6 7 8

1 2
3 4 5
6 7 8

2 3 5
1 4
6 7 8

1 2
3 4 5
6 7 8

2 5
1 3 4
6 7 8

Q
S.M.II

2 5
1 3 4
6 7 8

1 2 5
3 4
6 7 8

Output:

```
→ 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
```

8/12/2023

Lab-5

→ 8. Puzzle problem using Iterative Deepening Search.

Algorithm:

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

(2) ~~Puzzle~~ def puzzle:

→ accept $\langle \rangle \leftarrow$ start & goal state

→ dls($\langle \rangle$, goal, depth) \leftarrow perform dls

(3) ~~DLS~~ def DLS ($\langle \rangle$, goal, depth) :

If (Current-state == goal) \Rightarrow

return solution

else :

generate child node, recursively call with
increase level.

(4) ~~DLS~~ def dls ($\langle \rangle$, start, goal)

→ Start with depth : 0

→ Repeat until goal is found :

→ Perform DLS with current depth

→ If solⁿ.found \Rightarrow \leftarrow exit
increment depth.

(5) → Create puzzle instance -

→ Call dls (start, goal)

→ Code Iterative-Deeping-Search :-

→ def iddfs(puzzle, goal, get-moves)
import itertools

```
def ddfs(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 = ddfs(route + [move], depth)  
            if next-route:  
                return next-route  
    for depth in itertools.count():  
        route = ddfs([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(generate(state, i, b))

. return pos-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 = iddfs(initial, goal, possible-moves)

if route:

 print("Success! It is possible to solve 8 puzzle problem")

 print("Path:", route)

else:

 print("Failed to find a solution")

Output :

Success!! It is possible to solve 8 puzzle problem

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

92/12

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.

4 → 8 puzzle problem using A* algorithm:

(1) ~~def~~ Node(data, level): initialize node with puzzle state & level.

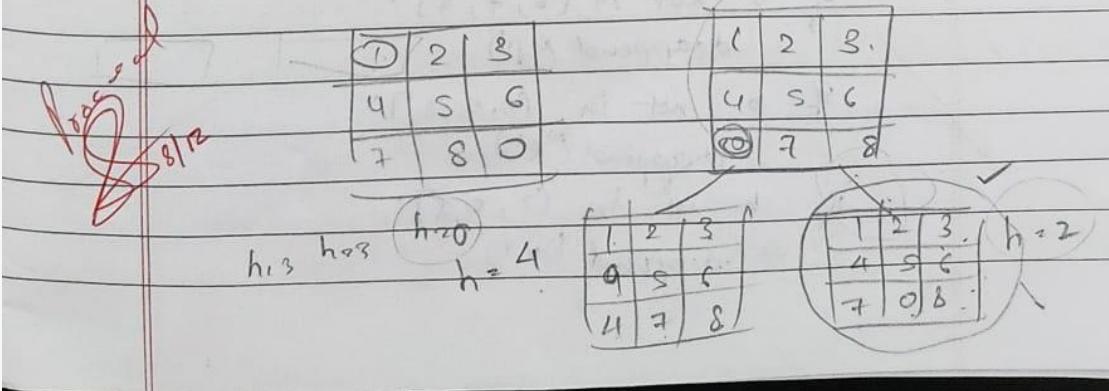
(2) def puzzle():
• accept(): ← Accept start & goal state.
• f(start, goal) ← calculate $f = h + g$
• process() ← perform A* algorithm.

(3) Heuristic
 $h(\text{start}, \text{goal})$: Manhattan dist. b/w current & goal state

(4) A* Algorithm:

- initialize start node, & add to open list
- loop until goal state come.
- select node have lowest f from ~~open list~~ open list
- Display state & check goal.
- generate child node & calculate f
- Add current node to closed list & remove from open list
- explore list by f
- display final move.

(5) ~~execute~~ → Create puzzle instance
→ call process for the execute.



~~92/12~~

→ Code of A* & puzzle using A* algorithm:

Class node :

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

def generate_child(self):

$x, y = self.find(self.data, '-')$

val-list = $[x, y-1], [x, y+1], [x-1, y], [x+1, y]$

children = []

for i in val-list:

child = self.shuffle(self.data, $x, y, i[0], i[1]$)

if child is not None

child-node = Node(child, self.level + 1)

children.append(child-node)

return children.

def shuffle(self, puzz, x1, y1, x2, y2):

if $x2 \geq 0$ and $y2 \geq 0$ and $y2 < \text{len}(\text{puzz.data})$ and $x2 < \text{len}(\text{puzz.data}[y2])$

$y2 \leq \text{len}(\text{puzz.data})$

```
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 find(self, puz, x):
    for i in range(0, len(self.data)):
        for j in range(0, len(self.data)):
            if puz[i][j] == x:
                return i, j
```

class Puzzler:

```
def __init__(self, size):
    self.n = size
    self.open = []
    self.closed = []
```

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

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

def h:

```

def m(self, start, goal):
    temp = 0
    for i in range(0, self.n):
        for j in range(0, self.n):
            if (start[i][j] != goal[j][i]) and
               start[i][j] != 0:
                temp = temp + 1
    return temp

```

```

def process(self):
    print("Enter the start state matrix (n x n)")
    start = self.accept()
    print("Enter the goal state matrix (n x n)")
    goal = self.accept()

```

```

start = Node(start, 0, 0)
start.fval = self.f(start, goal)
self.open.append(start)
print("n x n")

```

while True:

```

cur = self.open[0]
print(" ")
print("n x n")

```

```

for i in cur.data:
    for j in i:
        print(j, end=" ")

```

```

self.open.sort(key=lambda x: x.fval)

```

$p^{43} \sim p^{33} \log(3)$

Output

Enter the start 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 -

~~8/21/12~~

False)

Output:

▶ 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 _

22/12/2023

Lab-6

5

→ Implement the vacuum cleaner agent.

Algorithm:

1 → Vaccumn-Cleaning-Agent class :-

1 → __init__(self, room1, room2) : → constructor.

→ initialize the cleaning agent with 2 rooms
and set the current room.

2 → def clean_rooms(self) : Initiates the cleaning process
each rooms , if it calls the 'clean-room' function
to each room & then it calls the
'move_to_next-function' :

3 → def clean_room(self, room) :

if (room == 'clean') :

return (clean)

elif (room == 'dirty') :

initiates the cleaning process!

room == 'clean'

return (clean).

4 → def move_to_next_room:

Count = 0 :

→ Switches the current room to adjacent room.

if Count <= 2 :

else :
end

Move_to_next_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 = input("Enter initial location").upper()
```

```
vacuum = VacuumCleaner(initial_vacuum_location)
```

```
room_A_state = input("Enter state for Room A").lower()
```

```
room_B_state = input("Enter state for Room B :").lower()
```

room = A

'A' : room_A_state;

'B' : room_B_state.

print ("initial state: ")

print ("Vacuum cleaner is in Room A [vacuum.location] ")

print ("Room A: 2 rooms [A'] 3 ")

print ("Room B: 1 room [B'] 3 ")

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 == 'A':

 vacuum.move-right()

 current_room = 'A'

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

if cleaned_room == ~~old~~ 'clean':

 rooms [current_room] = 'clean'

print ("After cleaning completed.")

print ("Final state: ")

print ("Vacuum cleaner is in room of vacuum location")

print ("Room A: 2 rooms [A'] 3 ")

print ("Room B: 1 room [B'] 3 ")

Starvation Cleaning ()

Output:

Ever initial location of vacuum cleaner (A/B) : A
Ever state for Room A (clean/dirty) : dirty
Ever state for Room B (clean/dirty) : dirty

needed.)

Initial State:

Vacuum Cleaner 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

)

Final State:

Vacuum Cleaner is in Room B

Room A: clean

Room B: Clean

~~Ques 4)~~

~~Ques 1/2~~

Output:

```
② 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.

29/12/2023

G

Lab - 7

⇒ Q1) Create a knowledge base using propositional logic & show that the given query entails the knowledge base or not.

P

Algorithm:

Knowledge Base class:

→ Initialization :

def __init__(self):

 self.clauses = []

→ Create an instance of Knowledgebase class with an empty list

o) Adding a clauses:

Append a new clause to the list of clauses in the knowledge base.

o) Resolving clauses:

Combine two clauses by resolving

SUBSTUTION

(Q2) → Negate the Query:-

Obtain the negation of query.

Combine with knowledge base:

o) Check satisfiability :

def satisfiability () :-

 c check the conjunction rule by

Code(1) from simpy import symbols

```
def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')
```

```
knowledge_base = And(Implies(p,q), Implies(q))
                  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)
```

~~```
print("query entails knowledge base", result)
```~~

Output:-

knowledge\_base : or & (Implies(p,q)) & (Implies(q,r))

Query = p

Query entails knowledge base : False.

## Output:

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

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

7.2)

## \* Knowledge based resolution:

Algorithm:-

Initialization:-

- 1) Create an instance of knowledge base class with an empty list of clauses:-

self.clauses = []

- 2) Adding a clause:-

append a new clause to the list of clauses in the knowledge base.

- 3) Resolving clauses:-

combine 2 clauses by common literals  
(Resolving literals)  
(eliminating complementary literals)

negate the query & adds it to the knowledge base

- 4) Repeatedly resolve the pairs of clauses in the knowledge base until a contradiction found or no new resolution are possible.

query)

```
def resolve(self, clause_a, clause_b):
 return literal for literal in clause_a + clause_b
 if ('not' + literal) not in clause_a or
 literal not in clause_b
```

Code (2) 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.remove(literal)  
 resolved\_clause.add(negate\_literal(literal))  
return tuple(resolved\_clause)

def resolution (knowledge-base):

while True:

new\_clauses = set()

for i, q in enumerate(knowledge-base):  
 for j, l in enumerate(knowledge-base):

if i != j:

new\_clause = resolve(c1, c2)

if len(new\_clause) > 0 & new\_clause not in knowledge-base:  
 new\_clauses.add(new\_clause)

)

if not new\_clause  
break

knowledge-base ! = 'new-clause'  
return knowledge-base

7b -- name -- => "Main" :  
 $KB = \{ (P, S1), (\neg P, Q), (\neg Q, R) \}$   
 $result = resolution(KB)$   
 $print("original KB", KB)$   
 $print("resolved KB", result)$

Output:-

Enter statement :- negation.

The statement is not entailed by knowledge-base

~~Book~~

```

rules = 'PvQ ~PvR ~QvR' #P=vQ, 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.

```

8

Lab - 8

## (1) Unification :- Algorithm:

Eg: knows (John, x)    knows (John, Jane)  
 { 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 term is available

If term1 occurs in term2  
 return FAIL

(c) else if term2 is a variable.

If term2 occurs in term1  
 Return FAIL

Else

return of (term1 / term2)

(d) else return FAIL

Step 2 : If predicate(term1)  $\neq$  predicate(term2)  
 return FAIL

Step 3 : number of arguments  $\neq$   
 return FAIL

Step 4 : set (SUBST) to NIL

Step 5: For  $i = 1$  to the number of elements in  $L_1$

(a) call unify ( $i^{\text{th}} \text{ term}_1$ ,  $i^{\text{th}} \text{ term}_2$ )  
put result into  $S$

(b)  $S = \text{FAIL}$

return FAIL

(c) if  $S \neq \text{NIL}$

(a) Apply  $S$  to the remainder of both  
 $L_1$  and  $L_2$

(b) ~~set~~  $\text{SUBST} = \text{APPEND}(S, \text{SUBST})$

Step 6: Return SUBST.

Code :-

18 → unification

(1) import re

```
def getAttributes(expression):
 expression = expression.split("(\\"")[:-1]
 expression = " ".join(expression)
 expression = expression[:-1]
 expression = re.split("(\\?\\.\\?|\\?.\\?|\\?)", expression)
 return expression
```

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

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

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

if isConstant(exp1)  
 return P(exp2, exp1)

```
if isconstant(expr):
 return [(expr, exp1)]
```

```
if getInitialPredicate(exp1) != getInitialPredicate(exp2)
 print("Predicates do not match")
 return False.
```

```
attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
```

```
head1 = getFirstPart(exp1)
heads2 = getFirstPart(exp2)
```

```
if not initialSubstitution:
 return False
if attributeCount1 == 1:
 return initialSubstitution
```

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

```
exp1 = "knows (A, x)"
exp2 = "knows (y, y)"
```

```
substitution = unify(exp1, exp2)
print("so", "substitutions :")
print(substitution)
```

expr1 = "knows (A, x)"

expr2 = "knows (y, mother (y))"

substitution = unify (expr1, expr2)

print ("substitution: ")

print (substitution)

Output :-

① substitution :

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

substitution

[('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')]
```

9  
(2)

Convert First order logic to CNF conversion

Algorithm:-

Step1: Create a list of SKOLEM CONSTANTS.  $\rightarrow \{l, \delta, m\}$

Step2: find  $\forall, \exists$

• If the attributes are lower case, replace them with a skolem constant.

• Remove use skolem constant or function from the list

• If the attributes are both lowercase and uppercase replace the uppercase attribute with a skolem function

Step3: replace  $\Leftrightarrow$  with ' $\neg$ '  $P(\Rightarrow)Q$

transform  $\neg$  as  $\neg Q \equiv (\neg P \Rightarrow Q) \wedge (Q \Rightarrow \neg P)$ .

Step4: replace  $\Rightarrow$  with ' $\neg$ '  $P \Rightarrow Q$

transform  $\neg$  as  $\neg Q \equiv (\neg P \vee Q)$

g

Convert FOL to CNF

(B) Code

```
def getAttributes(string):
 expr = '\((\w+)\)+\)'
 matches = re.findall(expr, string)
 return [m for m in matches if m[0].isalpha()]
```

```
def getPredicate(string):
 expr = '[a-zA-Z]+(\w+[a-zA-Z]+)+\)'
 return re.findall(expr, string)
```

```
def DetArgon(sentence):
 string = ".join([str(sentence).copy(),"
 string += "string.replace('n', '')"
 flag = 'T' in string
```

```
for predicate in getPredicate(string)
 string = string.replace(predicate, f'n{predicate}')
```

```
s = list(string)
```

```
for i, c in enumerate(string):
 if c == '1':
 s[i] = '2'
 elif c == '2':
 s[i] = '1'
```

```
return '+'[string] if flag - else string
```

def skolemization(sentence):

~~statement constants~~

statement = ''.join(1[1:-1] for sentence in sentence).copy()

matches = re.findall('([&lt;][&gt;])', statement)

for match in matches[1:-1]:

statement = statement.replace(match, '')

for s in statements:

statement = statement.replace(s, s[1:-1])

for predicate in getPredicates(statements):

attributes = getAttributes(predicates)

if ''.join(attributes) == 'lower':

statement = statement.replace(M[i], pop())

else:

aL = [a for a in attributes if a.islower()]

aU = [a for a in attributes if a not in aL]

import re

def fol\_to\_cnf(fol):

statement = fol.replace('<=>', '-')

while '-' in statement:

i = statement.index('-')

n\_w = '[' + statement[0:i] + ']' + statement[i+1:-1]

[i+1:-1] & ['] + statement[i+1:-1]

'' + statement[i+1:-1]

statement = new\_statement

Statement = statement.replace (" $\Rightarrow$ ", " $-$ ")

while ' $-$ ' in statement :

i = statement.index ('-')

br = statement.index ('[') if '[' in statement else 0

while ' $\neg A$ ' in statement :

i = statement.index (' $\neg A$ ')

statement = list(statement)

statement[i], statement[i+1] = ']'

statement

while ' $\neg \exists$ ' in statement :

i = statement.index (' $\neg \exists$ ')

s = list(statement)

statement = ''.join(s)

expr = '( $\neg [A \exists]$ ). )'

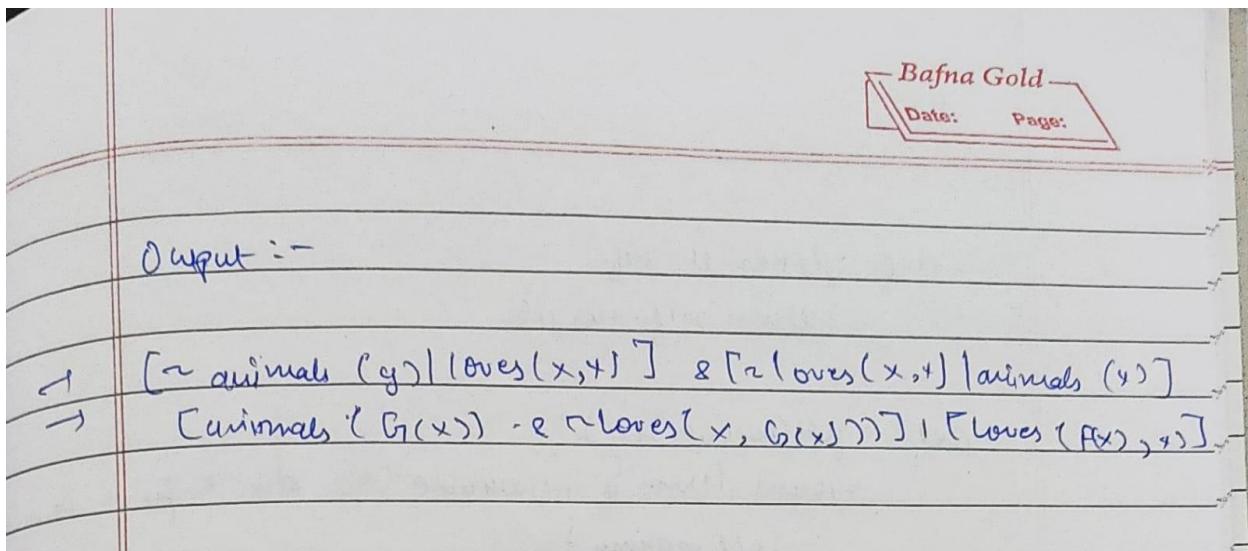
for s in statements :

statement = statement.replace (s, DeMorgan (s))

return statement

print (skolemization (fol\_to\_cnf ("animal (y)  $\Rightarrow$  loves (x<sub>1</sub>)"))

print (skolemization (fol\_to\_cnf (" $\forall x [\forall v [animal (v) \Rightarrow loves$ (  
z, x)]])"))



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)
```

~~Practise~~  
10  
(3) Create a knowledge consistency of for statement and prove the give query using back forward chaining, memory

Algorithm:-

- 1) Initialize the Agenda:-
  - add the query to the agenda.
- 2) while the agenda do is not empty :
  - a) pop a statement from the agenda.
  - b) if the statement is already known , continue to repeat statement
  - c) if the statement is a law , add it to the set of known fact.
  - d). If the statement is a rule , apply the rule to the generate ~~the~~ new statements & add them to the agenda.

Q) Code for KB consisting for + prove the given query using forward reasoning.

import re

def isVariable(x):

return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string)

expr = ' \((\wedge)\) + )'

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.getConstant(s))

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

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

Class Implications:

```
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.rhs

if val.predicate == fact.predicate:

for i, v, m in enumerate(val.getVariables()):

if v:

count[v] >= fact.getLifespan()

new\_lhs.append(fact)

for key in constants:

if Constants[key]:

attribute = attributes.replace(key, constant)

return fact(expr) if len(new-lhs) and all([t.  
getBelief() for t in new-lhs]) else  
None

Class KB

```
def __init__(self):
 self.facts = set()
 self.implications = set()
```

```
def tell(self, e):
 if '=>' in e:
 self.implications.add(implication(e))
 else:
 self.facts.add(fact(e))
```

```
for i in self.implications:
 res = i.evaluate(self.facts)
 if res:
 self.facts.add(res)
```

```
def display(self):
 print("All facts:")
 for i, f in enumerate(set([f.expression
 for f in self.facts])):
 print(f'{i+1}. {f}')
```

Kb -> KB1  
 Kb - tell('Ring(x) & greedy(x) => evil(x)')  
 Kb - tell('Ring(John)')  
 res - tell('greedy(John)')

kb - tell ('King (Richard)')  
kb:query ('evil (x)')

output

Every evil (+)  
1. evil (John)

2011/2/21  
cool!  
very cool!

## 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)
```

---