VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

ARTIFICIAL INTELLIGENCE

Submitted by

YASHASVINI M R(1BM21CS252)

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 **YASHASVINI M R(1BM21CS252),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester Nov -2023 to Feb-2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Dr Kayarvizhy N
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Course Outcome

CO1	Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO2	Analyse Searching and Inferencing Techniques.
CO3	Design a reasoning system for a given requirement.
CO4	Conduct practical experiments for demonstrating agents, searching and inferencing.

1. Implement Tic –Tac –Toe Game

```
import math

def print_board(board):
    for i in range(len(board)):
        for j in range(len(board[i])):
            print(board[i][j], end='')
            if j < len(board[i]) - 1:
                print('|', end='')
        print()
        if i < len(board) - 1:
            print('-'*5)
    print()

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
    return None

def get_empty_cells(board):
    # Returns a list of empty cells in the board
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == '
']

def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner:
        return 10 - depth if winner == 'X' else -10 + depth
    elif not get_empty_cells(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'X'
```

```
            score = minimax(board, depth + 1, False)
            board[i][j] = ' '
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'O'
            score = minimax(board, depth + 1, True)
            board[i][j] = ' '
            best_score = min(score, best_score)
        return best_score

def best_move(board):
    best_score = -math.inf
    move = None
    for i, j in get_empty_cells(board):
        board[i][j] = 'X'
        score = minimax(board, 0, False)
        board[i][j] = ' '
        if score > best_score:
            best_score = score
            move = (i, j)
    return move

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

    while not check_winner(board) and get_empty_cells(board):
        user_move = input("Enter your move (row and column separated by a
space): ")
        x, y = map(int, user_move.split())
        if board[x][y] == ' ':
            board[x][y] = 'O'
            print_board(board)
        else:
            print("Invalid move. Try again.")
            continue

        if not get_empty_cells(board):
            break

        computer_move = best_move(board)
```

```python
        board[computer_move[0]][computer_move[1]] = 'X'
        print("Computer's move:")
        print_board(board)

    winner = check_winner(board)
    if winner:
        print(f"Player {winner} wins!")
    else:
        print("It's a tie!")

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

OUTPUT

```
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |

Enter your move (row and column separated by a space): 2 2
 | |
-----
 | |
-----
 | |O

Computer's move:
 | |
-----
 |X|
-----
 | |O

Enter your move (row and column separated by a space): 1 2
 | |
-----
 |X|O
-----
 | |O
```

```
Computer's move:
 | |X
-----
 |X|O
-----
 | |O

Enter your move (row and column separated by a space): 2 0
 | |X
-----
 |X|O
-----
O| |O

Computer's move:
 | |X
-----
 |X|O
-----
O|X|O

Enter your move (row and column separated by a space): 1 1
Invalid move. Try again.
Enter your move (row and column separated by a space): 0 1
 |O|X
-----
 |X|O
-----
O|X|O
```

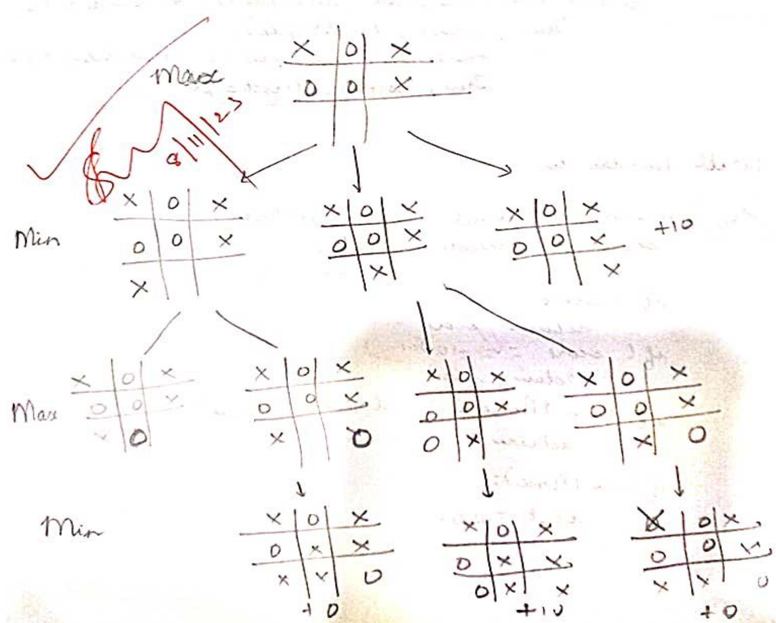
```
Computer's move:
X|O|X
-----
 |X|O
-----
O|X|O

Enter your move (row and column separated by a space): 1 0
X|O|X
-----
O|X|O
-----
O|X|O

It's a tie!
```

OBSERVATION

Tic - Tac - Toe

The tic-tac-toe game makes use of the Minimax algorithm.

- Consider a player - X as the maximizer. This player tries to maximize to win by choosing a move which gives higher score. The score will be chosen in such a way that it ensures faster victory of X using lesser moves.
-



15-11-2023

Minimax Algorithm

```
(In general)
import math

def minimax(curDepth, nodeIndex, maxTurn, scores,
            targetDepth):

    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth+1, nodeIndex*2,
               False, scores, targetDepth),
               minimax(curDepth+1, nodeIndex*2+1,
               False, scores, targetDepth))
    else:
        return min(minimax(curDepth+1, nodeIndex*2,
               True, scores, targetDepth),
               minimax(curDepth+1, nodeIndex*2+1
               True, scores, targetDepth))
```

With tic-tac-toe :

```
def minimax(board, depth, isMax):
    score = evaluate(board)

    if (score == 10):
        return score
    if (score == -10):
        return score
    if (isMovesLeft(board) == False):
        return 0
    if (isMax):
        best = -1000
        for i in range(3):
            for j in range(3):
                if (board[i][j] == '_'):
                    board[i][j] = player
                    best = max(best, minimax(board,
                                   depth+1, not isMax))

                    board[i][j] = '_'
        return best
    # minimize
    else:
        best = 1000
        for i in range(3):
            for j in range(3):
                if (board[i][j] == '_'):
                    board[i][j] = opponent
                    best = min(best, minimax(board,
                           depth+1, not isMax))
                board[i][j] = '_'
        return best
```

AI- approaches

Rule - Based approach : Rules are given by us humans.

Machine - learning approach : Learns from the dataset

15/11/23

2. Implement vaccum cleaner agent

```
def printInformation(location):
    print("Location " + location + " is Dirty.")
    print("Cost for CLEANING " + location + ": 1")
    print("Location " + location + " has been Cleaned.")


def vacuumCleaner(goalState, currentState, location):
    # printing necessary data
    print("Goal State Required:", goalState)
    print("Vacuum is placed in Location " + location)

    # cleaning locations
    totalCost = 0

    while (currentState != goalState):
        if (location == "A"):
            # cleaning
            if (currentState["A"] == 1):
                currentState["A"] = 0
                totalCost += 1
                printInformation("A")
            # moving
            elif (currentState["B"] == 1 ):
                print("Moving right to the location B.\nCost for moving 
RIGHT: 1")
                location = "B"
                totalCost += 1

        elif (location == "B"):
            # cleaning
            if (currentState["B"] == 1):
                currentState["B"] = 0
                totalCost += 1
                printInformation("B")
            # moving
            elif (currentState["A"] == 1):
                print("Moving left to the location A.\nCost for moving LEFT: 
1")
                location = "A"
                totalCost += 1

    print("GOAL STATE:", currentState)
    return totalCost
```

```
# declaring dictionaries
goalState = {"A": 0, "B": 0}
currentState = {"A": -1, "B": -1}

# taking input from user
location = input("Enter Location of Vacuum (A/B): ");
currentState["A"] = int(input("Enter status of A (0/1): "))
currentState["B"] = int(input("Enter status of B (0/1): "))

# calling function
totalCost = vacuumCleaner(goalState, currentState, location)
print("Performance Measurement:", totalCost)
```

OUTPUT

```
Enter Location of Vacuum (A/B): B
Enter status of A (0/1): 1
Enter status of B (0/1): 1
Goal State Required: {'A': 0, 'B': 0}
Vacuum is placed in Location B
Location B is Dirty.
Cost for CLEANING B: 1
Location B has been Cleaned.
Moving left to the location A.
Cost for moving LEFT: 1
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
GOAL STATE: {'A': 0, 'B': 0}
Performance Measurement: 3
```

OBSERVATION

Vacuum World Program

- Takes input of initial location, and status of location A and B.
- If ~~location~~ it is in location 'A' and it is dirty, it cleans or if it is clean and location 'B' is dirty is moves to location 'B' and cleans till it reaches goal state {'A': 0, 'B': 0}
- Cost is one each for moving and cleaning.
- Performance measure is total cost.

Program:

```
def PrintInformation(location):
    print ("Location " + location + " is Dirty.")
    print ("Cost for cleaning " + location + ": 1")
    print (" Location " + location + " has been Cleaned.")
def vacuumCleaner(goalState, currentState, location):
    print( "Goal State required:", goalState)
    print ("Vacuum is placed in Location " + location)

    total cost = 0

    while (currentState != goalState):
        if (location == 'A'):
            if (currentState ["A"] == 1):
                currentState ["A"] = 0
                total Cost += 1
                print Information ("A")

            elif (currentState ["B"] == 1):
                print ("Moving right to the location B.\n
    Cost for moving right : 1")
            location = "B"
            total Cost += 1

        elif (location == "B"):
            if (currentState ["B"] == 1):
                currentState ["B"] = 0
                total Cost += 1
                print Information ("B")
            elif (currentState ["A"] == 1):
                print ("Moving left to the location A.\n
    Cost for moving left : 1")
                location = "A"
                total Cost += 1
print ("Goal State:", Current State)
return total Cost

goalState = {"A": 0, "B": 0}
currentState = {"A": -1, "B": -1}

location = input( "Enter location of Vacuum(A/B): ").
currentState ["A"] = int (input ("Enter state of A(0/1): ")
currentState ["B"] = int (input (" Enter status of B(0/1) : ")

total Cost = vacuum Cleaner( goalState, currentState, location)
print (" Performance Measurement:", total Cost)
```

Output

```
Enter location of Vacuum (A/B) : B
Enter State of A(0/1) : 1
Enter state of B(0/1) : 1.
Goal State required: {'A': 0, 'B': 0}
Vacuum is placed in Location B
Location B is dirty
Cost for Cleaning B: 1
Location B is Cleaned.
Moving left to the Location A.
Cost for moving Left : 1
```

3. Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm

```
def bfs(src, target):
    queue = []
    queue.append(src)
    visited = set()

    while queue:
        source = queue.pop(0)
        visited.add(tuple(source))  # Store visited states as tuples for
faster lookup

        print(source[0], '|', source[1], '|', source[2])
        print(source[3], '|', source[4], '|', source[5])
        print(source[6], '|', source[7], '|', source[8])
        print("-----------")

        if source == target:
            print("Success")
            return

        poss_moves_to_do = possible_moves(source, visited)
        for move in poss_moves_to_do:
            queue.append(move)

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []

    # Add possible directions to move based on the position of the empty
cell
    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')

    pos_moves_it_can = []

    for i in d:
```

```
        pos_moves_it_can.append(gen(state, i, b))

    # Return possible moves that have not been visited yet
    return [move_it_can for move_it_can in pos_moves_it_can if
tuple(move_it_can) not in visited_states]

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

# Taking input for initial and goal states
print("Enter the initial state of the puzzle (use numbers 0-8 separated by
spaces):")
src = list(map(int, input().split()))

print("Enter the goal state of the puzzle (use numbers 0-8 separated by
spaces):")
target = list(map(int, input().split()))

bfs(src, target)
```

OUTPUT

```
Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 0 4 5 6 7 8
Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 4 5 0 6 7 8
1 | 2 | 3
0 | 4 | 5
6 | 7 | 8
-----------
0 | 2 | 3
1 | 4 | 5
6 | 7 | 8
-----------
1 | 2 | 3
6 | 4 | 5
0 | 7 | 8
-----------
1 | 2 | 3
4 | 0 | 5
6 | 7 | 8
-----------
2 | 0 | 3
1 | 4 | 5
6 | 7 | 8
-----------
1 | 2 | 3
6 | 4 | 5
7 | 0 | 8
-----------
1 | 0 | 3
4 | 2 | 5
6 | 7 | 8
-----------
1 | 2 | 3
4 | 7 | 5
6 | 0 | 8
-----------
1 | 2 | 3
4 | 5 | 0
6 | 7 | 8
-----------
Success
```

OBSERVATION

8 - Puzzle Problem Using - BFS

- Uses queue to hold all states that have to be / been visited
- When visited, pops them to a list called exp.
- The 8 - Puzzle problem requires the initial state to reach the goal state by moving the tiles to blank.
- Possible moves are 'd', 'u', 'r', 'l'.

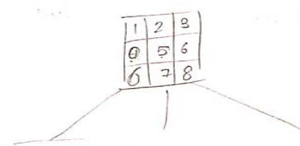
```
def bfs(src, target):
    queue = []
    queue.append(src)
    exp = []
    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print(source[0], '|', source[1], '|', source[2])
        print(source[3], '|', source[4], '|', source[5])
        print(source[6], '|', source[7], '|', source[8])
        print("..------")
        if source == target:
            print("Success")
            return
        poss_moves_to_do = possible_moves(source, exp
        for move in poss_moves_to_do:
            if move not in exp and move not in queue
                queue.append(move)

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []
    if b not in [0, 1, 2]:
        d.append('u')        //
    if b not in [6, 7, 8]:
        d.append('d')
```

```
 if b not in [0,3,6]:
      d.append('l')
 if b not in [2,5,8]:
      d.append('r')

 poss_moves_it_can = []
 for pos i in d:
      pos_moves_it_can.append(gen(state, i, b))
 return [move_it_can for move_it_can in pos_moves_
   it_can if move_it_can not in visited_states]

def gen(states, 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]
                                  c; b- b. a
    if m == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1] //swap

    if m == 'r':
        temp[b+1], temp[b], tem[b-1

    return temp
print (" Enter the initial state of the puzzle( use
  numbers 0-8 separated by spaces) : ")
src = list( map( int, input (). split ()))
print (" Enter the goal state of the puzzle ( use
numbers 0-8 separated by spaces ),")
target = list ( map( int , input (). split ()))
 bfs ( src , target)
```

Output
Enter the initial state of puzzle(use numbers 0-8
separated by spaces):
1 2 3 0 4 5 6 7 8

Enter the goal state of the puzzle(use numbers 0-8
separated by spaces):
1 2 3 4 5 0 6 7 8

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

1 | 2 | 3
4 | 7 | 5
6 | 0 | 8
- - - - -
1 | 2 | 3
4 | 5 | 0
6 | 7 | 8
- - - -
Success
```



4. Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be solved using this algorithm

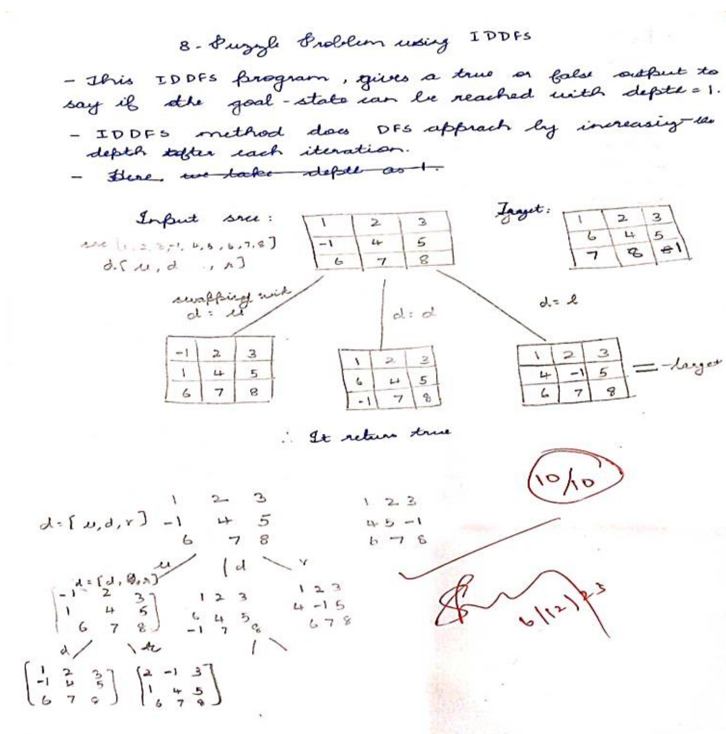
```
def dfs(src,target,limit,visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False
    visited_states.append(src)
    moves = possible_moves(src,visited_states)
    for move in moves:
        if dfs(move, target, limit-1, visited_states):
            return True
    return False

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

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

def iddfs(src,target,depth):
    for i in range(depth):
```

```
        visited_states = []
        if dfs(src,target,i+1,visited_states):
            return True, i+1
    return False

print("Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):")
src = list(map(int, input().split()))

print("Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):")
target = list(map(int, input().split()))
depth = 8
iddfs(src, target, depth)
```

OUTPUT

```
Enter the initial state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 -1 4 5 6 7 8
Enter the goal state of the puzzle (use numbers 0-8 separated by spaces):
1 2 3 6 4 5 7 8 -1
(True, 3)
```

OBSERVATION



5. Implement A* search 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+1,0)
                children.append(child_node)
        return children

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

        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 <
len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):

        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp
```

```
    def find(self,puz,x):

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


class Puzzle:
    def __init__(self,size):

        self.n = size
        self.open = []
        self.closed = []

    def accept(self):

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

    def f(self,start,goal):

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

    def h(self,start,goal):

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


    def process(self):

        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()
```

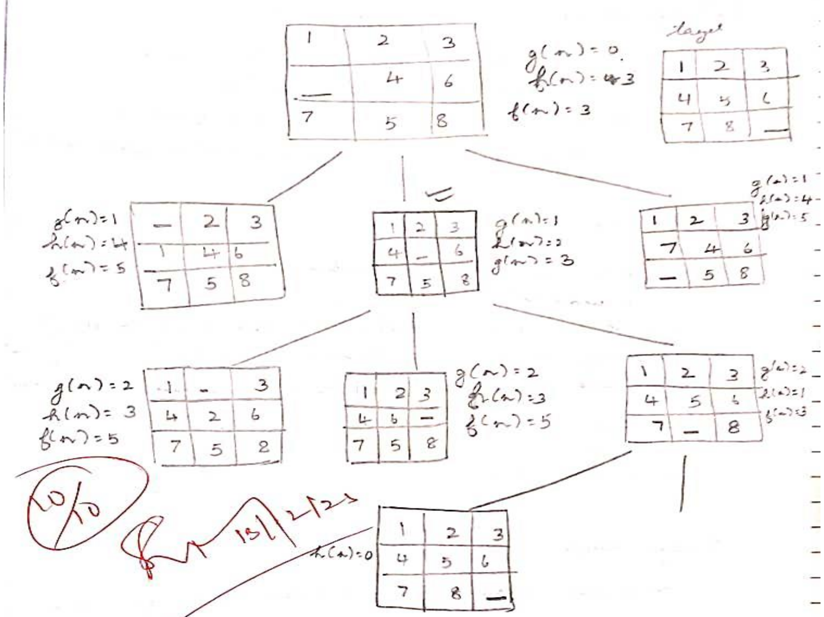
```
        start = Node(start,0,0)
        start.fval = self.f(start,goal)

        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")

            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]


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


puz = Puzzle(3)
puz.process()
```

OUTPUT

```
Enter the start state matrix

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

1 2 3
4 5 6
7 8 _



   |
   |
 \'/

1 2 3
_ 4 6
7 5 8

   |
   |
 \'/

1 2 3
4 _ 6
7 5 8

   |
   |
 \'/

1 2 3
4 5 6
7 _ 8

   |
   |
 \'/

1 2 3
4 5 6
7 8
```

OBSERVATION

8 - Puzzle Problem using A*



- In A* search algorithm, the node to explore is chosen based on it's value of f(n)

ie $f(n) = h(n) + g(n)$

where $h(n)$ = heuristic function, in this code, it is the number of misplaced tiles in that state from goal state.

and $g(n)$ = depth of the node.

6. Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not

```
def tell(kb, rule):
    kb.append(rule)

combinations = [(True, True, True), (True, True, False),
                (True, False, True), (True, False, False),
                (False, True, True), (False, True, False),
                (False, False, True), (False, False, False)]


def ask(kb, q):
    for c in combinations:
        s = r1(c)
        f = q(c)
        print(s, f)
        if s != f and s != False:
            return 'Does not entail'
    return 'Entails'


kb = []

rule_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or
x[1] and (x[0] and x[1]): ")
r1 = eval(rule_str)
tell(kb, r1)

query_str = input("Enter Query as a lambda function (e.g., lambda x: x[0]
and x[1] and (x[0] or x[1]): ")
q = eval(query_str)


result = ask(kb, q)
print(result)
```

OUTPUT 1

```
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x: (not x[1] or not x[0] or x[2]) and (not x[1] and
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x: x[2]
False True
False False
False True
False False
False True
False False
False True
False False
Entails
```

OUTPUT 2

```
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x: (x[0] or x[1]) and (not x[2] or x[0])
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x: x[0] and x[2]
True True
True False
Does not entail
```

OBSERVATION

20-12-23

Propositional Logic

- If ¬q is true iff P is false in m
- P ∧ q is true iff P and q are true in m
- P ∨ q is true iff either P or q is true
- P → q is true unless P is true and Q is false in m.
- P ⟺ q is true iff P and q are both true or both false

Sample Input output.

Enter Rule 1 as a lambda function (eg, lambda x: x[0] or x[1] and (x[0] and x[1]):
lambda x: (x[0] or x[1]) and (not x[2] or x[0]
Enter Query as a lambda function (eg. lambda x: x[0] and x[1] and (x[0] or x[1]): lambda x: x[0] and x[2]

True True
True False
Does not entail

Query entails KB only if KB → Query
Does not entail if KB = true & Query = false

10/10

20/12/23

7. Create a knowledge base using prepositional logic and prove the given query using resolution

```
import re

def main():
    rules = input("Enter the rules (space-separated): ")
    goal = input("Enter the goal: ")
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1


def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]


def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms


def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}',
f'{negate(goal)}v{goal}']
    return clause in contradictions
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
```

```
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']

                            if
contradiction(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and
{temp[j]} to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when
{negate(goal)} is assumed as true. Hence, {goal} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if
contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and
{temp[j]} to {temp[-1]}, which is in turn null. \
                            \nA contradiction is found when
{negate(goal)} is assumed as true. Hence, {goal} is true."
                            return steps
            for clause in clauses:
                if clause not in temp :
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and
{temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter the rules (space-separated): Rv~P Rv~Q ~RvP ~RvQ
Enter the goal: R

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.
```

OBSERVATION

27/12/23 Create a knowledge base using propositional logic and prove the given query using resolution.

- The query is concluded to be true, using resolution of the clauses.
- While resolving, if we get null — then the query is true. (contradiction is found)

Enter the kb:
R V ~P   R V ~Q   ~R V P   ~R V Q
Enter the Query:
R

10/10

Tracing:

rules = R V ~P   R V ~Q   ~R V P   ~R V Q
Given goal = R

steps = (R V ~P (all rules) rule, R goal)
then create a copy of the rules and add ~R to it.
- dictionary - steps.
- Rules first as given, and then negation of goal as negated conclusion.

temp [1] = R V ~P
temp [2] = R V ~Q
terms 1 = 'R' '~P'
terms 2 = 'R' '~Q'

If sign C in term R & ~P
here ~P not in terms 2
then ~P is also not in term 2
clause is empty now.
so i = i++, temp [3] = ~R V P

8. Implement unification in first order logic

```
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 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, exp1)]

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

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

```
    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return False

    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution


exp1 = input("Enter the first expression: ")
exp2 = input("Enter the second expression: ")


substitutions = unify(exp1, exp2)


print("Substitutions:")
print(substitutions)
```

OUTPUT

```
Enter the first expression: knows(f(x),y)
Enter the second expression: knows(J,John)
Substitutions:
[('J', 'f(x)'), ('John', 'y')]
```

OBSERVATION

10-1-24

Implementation of Unifications in First Order Logic

- Unification in first order is

- Starts with unify(exp1, exp2)

Input: knows(f(x), y) - first expression
knows(J, John) - second expression.

in unify((knows(f(x), y), (knows(~~f(x), y~~))) J, John

returns [] as exp1 ≠ exp2
returns as false for isConstant
returns false for is Variable.

then if checks id getInitialPredicate(exp1) &

- checks if the string before '(' is same in both expressions.

- Since it true, it returns True.

and goes to count the number of attributes in both the expression.

- Since it is same, it returns true.

Here exp1 = exp2 = 2

head 1 = getFirstPart(knows(f(x), y)) = f(x)
head 2 = getFirstPart(knows(J, John)) = J

Initial Substitution = unify(f(x), J)

i) exp1 ≠ exp2 return []
ii) f(x) && J ≠ constant, returns False
iii) is Constant (f(x)) - false
iv) is Constant (J) - true

∴ it returns [J, f(x)]

Initial Substitute = [J, f(x)]

now, tail 1 = getRemainingPart(knows (f(x), y))
tail 2 = getRemainingPart (knows (J, John))
Predicate = getInitialPredicate (knows (f(x), y))
= knows.
attributes = getAttributes (expression)
= (f(x), y)
new Express = knows (, y)

return (, y)
After doing for both.

tail 1 = (y)
tail 2 = (John)

initial Subn = [J, f(x)] – not empty.
tail 1 = apply(tail 1 (y), ~~initial subs~~ [J, f(x)])
tail 2 = apply(John , [J, f(x)])

apply (kn(y), [J, f(x)])
exp = y
exp = knows(y)
For subs in substitution
new = J
old = f(x)

exp = replace Attributes (y, f(x), J)
exp, old, new

attribute = getAttributes (kn(y)) = y
predicate = knows
[y]
return knows (y) y ≠ f(x)
return knows(y)
tail 1 = Knows (y)
tail 2 = knows (John)
remaining Substitution = unify (knows(y), knows(John))
Everything repeats.
is Variable (y)
if checkOccurs (John, y)
return false
hence return [(John, y)]

9. Convert a given first order logic statement into Conjunctive Normal Form (CNF)

```
import re

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

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

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            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).islower():
                statement =
statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
```

```
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU,
f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')
    return statement

def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] +
']&['+ statement[i+1:] + '=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else
new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃',
statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀','[~∀')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[∀|∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement
```

```python
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)"))
```

OUTPUT

```
[~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)
```

OBSERVATION

17-1-24 Convert First Order Logic to CNF

- Remove implication.
  $a \rightarrow b \equiv \neg a \vee b$
- Move ¬ inwards. $\neg \forall x P = \exists x \neg$
  eg: $\neg(a \vee b) \equiv \neg a \wedge \neg b$
- Rename variables, - give different variables for each statement
- Eliminate existential instantiate by elimination.
  replace ∃ by constant and replace the constant by a f (skolem function)
- Drop universal quantifier.
  - keep ∨ as it is, change where there is ∧
  ~~ust~~ ~~∃~~ seperate the literals) clauses.

Algorithm :
- Take input calls fol_to_cnf :
  replaces <=> with "–", replace that with => a
  a<=>b as a=>b & b=>a
- Then replace => with '–' change it to
  a-b as ~a∨b
- Converts ~∀x P(x) to ∃x P(x)
- Coverts ~∃x p(x) to ∀x P(x)
- ~~Then~~ If ~[statement] performs DeMorgan's law:
  replace '~~' to ''
  flag = [— used later.
  Removes ~[&]
  replaces | with & and & with |
- Skolemization – eliminate . ∃
  makes a copy of list for the statement
  matches [∀∃] in reverse

eg: animal(y)<=> loves(x,y)
[~animal(y)|loves (x,y)] & [~loves(x,y)|animal(y)]

17/1/24

10.Create a knowledge base consisting of first order logic statements and 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 = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
```

```
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f
in new_lhs]) else None

class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')
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,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

OUTPUT

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

OBSERVATION

24-01-234

Creating a knowledge base consists of first order logic statements and prove the given query using forward reasoning

tell('missile(x) => weapon(x)')
tell('missile(M1)')
tell('enemy(x, America) => hostile(x)')
tell('american(West)')
tell('enemy(Nono, America)')
tell('owns(Nono, M1)')
tell('missile(x) & owns(Nono, x) => sells(West, x, Nono)')
tell('american(x) & weapon(y) & sells(x, y, z) & hostile(z) => criminal(x)')

tell('criminal(x)'

tell('missile(x) => weapon(x)')
there is '=>' in s
lhs = missile(x)
rhs = weapon(x)

24/1/24