VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pranav Y (1BM22CS204)

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

Prof. Swathi Sridharan
Assistant Professor
Department of Computer Science and Engineering

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019

Sep-2024 to Jan-2025

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 (23CS5PCAIN)" carried out by Pranav Y **(1BM22CS204),** 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. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Prof. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HoD Department of CSE, BMSCE

Index

Github Link: https://github.com/PranavY204/AI

<u>Program 1</u>

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:

Lab-1 Tic-Tac-Toe bot

→ <u>Pseudocode</u>:

- Function to determine winner of a given board

```
def winner (board):
    d1 = set([board[i][i] for i in range(3)])
    d2 = set([board[2-i][i] for i in range(3)])
    rows = [set([board[i][j] for j in range(3)])
                for i in range(3)]
    col = [set([board[j][i] for j in range(3)])
                for i in range(3)]
    if d1 == {'x'} or d2 == {'x'} or
        {'x'} in rows or {'x'} in cols:
        winner = 'x'
    elif d1 == {'o'} or d2 == {'o'} or {'o'} in
        rows or {'o'} in cols:
        winner = 'o'
    else:
        return None
```

~~Determine of score for~~

- To determine next move of bot:

```
def checkMove(board, move, bot):
    for move in possibleMoves(board, player)
    boardcopy = applyMove(board)
        if winner(boardcopy) == bot:
            return move
        elif winner(boardcopy) == player:
            set condition return
        else:
            return checkMove(boardcopy, move, bot)
    if terminal(board):
        return None
```

Code:

```
"""
Tic Tac Toe Player
"""

# import math
import copy

X = "X"
O = "O"
EMPTY = None

def player(board):
    """
    Returns player who has the next turn on a board.
    """
    nx = sum([i.count("X") for i in board])
    no = sum([i.count("O") for i in board])
    if nx > no:
        return "O"
    else:
        return "X"

    raise NotImplementedError


def actions(board):
    """
    Returns set of all possible actions (i, j) available on the board.
    """
    ls = set()
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == EMPTY:
                ls.add((i, j))
    return ls
    raise NotImplementedError


def result(board, action):
    """
    Returns the board that results from making move (i, j) on the board.
    """
    ls = actions(board)
    if action not in ls:
        raise Exception("Invalid action")
    boardcopy = copy.deepcopy(board)
    if player(board) == "X":
        boardcopy[action[0]][action[1]] = "X"
    elif player(board) == "O":
        boardcopy[action[0]][action[1]] = "O"

    return boardcopy

    raise NotImplementedError
```

```
def winner(board):
    """
    Returns the winner of the game, if there is one.
    """
    diagonal1 = set([board[i][i] for i in range(3)])
    diagonal2 = set([board[i][2 - i] for i in range(3)])
    rows = [set([board[i][0], board[i][1], board[i][2]]) for i in range(3)]
    columns = [set([board[0][i], board[1][i], board[2][i]]) for i in range(3)]
    if diagonal1 == {"X"} or diagonal2 == {"X"} or {"X"} in rows or {"X"} in columns:
        return "X"
    elif diagonal1 == {"O"} or diagonal2 == {"O"} or {"O"} in rows or {"O"} in columns:
        return "O"
    return None
    raise NotImplementedError


def terminal(board):
    """
    Returns True if game is over, False otherwise.
    """
    if winner(board) is not None or (
        not any(EMPTY in sublist for sublist in board) and winner(board) is None
    ):
        return True
    else:
        return False
    raise NotImplementedError


def utility(board):
    """
    Returns 1 if X has won the game, -1 if O has won, 0 otherwise.
    """
    if winner(board) == "X":
        return 1
    elif winner(board) == "O":
        return -1
    else:
        return 0
    raise NotImplementedError


def minimax(board):
    """
    Returns the optimal action for the current player on the board.
    """
    if terminal(board):
        return None

    def max_value(board):
        if terminal(board):
            return (utility(board), None)
```

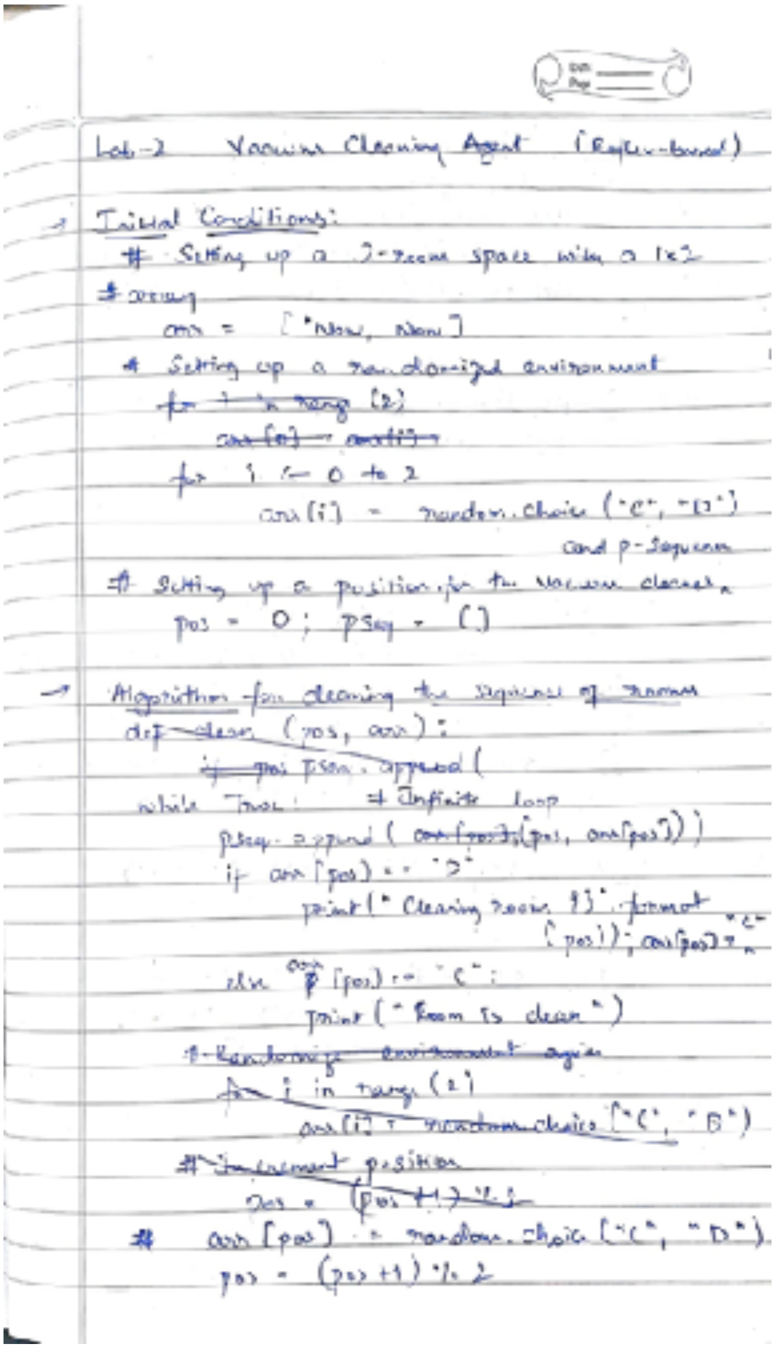
```
        value = float("-inf")
        action = None
        for move in actions(board):
            # v = max(v, min_value(result(board, action)))
            v, act = min_value(result(board, move))
            if v > value:
                value = v
                action = move
                if v == 1:
                    return (v, action)

        return (v, action)

    def min_value(board):
        if terminal(board):
            return utility(board), None

        value = float("inf")
        action = None
        for move in actions(board):
            # v = max(v, min_value(result(board, action)))
            v, act = max_value(result(board, move))
            if v < value:
                value = v
                action = move
                if value == -1:
                    return (value, action)

        return (value, action)

    if player(board) == "X":
        return max_value(board)[1]
    else:
        return min_value(board)[1]

    raise NotImplementedError


def display(board):
    for i in range(3):
        print(f"{board[i][0]}\t{board[i][1]}\t{board[i][2]}", end="\n\n")


def main():
    board = [[None for _ in range(3)] for _ in range(3)]
    while not terminal(board):
        current_player = player(board)
        if current_player == O:
            move = int(input("Enter possible cell to enter in (0 - 8): "))
            action = (move // 3, move % 3)
            if action in actions(board):
                board[action[0]][action[1]] = O
            else:
                print("Enter a valid move")
                move = int(input("Enter possible cell to enter in (0 - 8): "))
```

```
            action = [move // 3, move % 3]
        else:
            move = minimax(board)
            board[move[0]][move[1]] = X

        display(board)
        if winner(board) is not None:
            print(f"{winner(board)} wins!")


main()
```

```
None    X       None

None    None    None

None    None    None

Enter possible cell to enter in (0 - 8): 3
None    X       None

O       None    None

None    None    None

X       X       None

O       None    None

None    None    None

Enter possible cell to enter in (0 - 8): 2
X       X       O

O       None    None

None    None    None

X       X       O

O       X       None

None    None    None

Enter possible cell to enter in (0 - 8): 8
X       X       O

O       X       None

None    None    O

X       X       O

O       X       None

None    X       O

X wins!
```

Vacuum Cleaner

Algorithm:



Lab-2 Vacuum Cleaning Agent (Reflex-based)

Initial Conditions:

\# Setting up a 2-room space with a 1x2

\# array

arr = ['None, None]

\# Setting up a randomized environment

~~for i in range (2)~~

~~arr[i] = arr[i]~~

for i ← 0 to 2

arr[i] = random.choice ("C", "D")

and p-sequence

\# Setting up a position for the vacuum cleaner

pos = 0; pSeq = []

Algorithm for cleaning the sequence of rooms

def ~~clean~~ (pos, arr):

~~if pos~~ pSeq.append(

while True: # Infinite loop

pSeq.append(~~arr[pos]~~ (pos, arr[pos]))

if arr[pos] == "D"

print("Cleaning room {}".format (pos)); arr[pos] = "C"

else arr[pos] == "C":

print ("Room is clean")

~~# Randomize environment again~~

~~for i in range (2)~~

~~arr[i] = random.choice ["C", "D"]~~

\# ~~Increment~~ position

~~pos = (pos+1) % 2~~

\# arr[pos] = random.choice ("C", "D")

pos = (pos+1) % 2

Code:

import random

class Cleaner:

    def __init__(self):

```python
        self.env = [[None, None], [None, None]]
        for i in range(2):
            for j in range(2):
                self.env[i][j] = random.choice(("C", "D"))
        self.pos = [0, 0]
        self.pseq = []
        self.clean()

    def display(self):
        print("Current percept seq: ", self.pseq)
        print("Current env: ")
        print(self.env[0], self.env[1], sep="\n")

    def clean(self):
        moves = [[0, 1], [1, 0], [0, -1], [-1, 0]]
        next_move_idx = 0
        while True:
            self.pseq.append((self.pos, self.env[self.pos[0]][self.pos[1]]))
            if self.env[self.pos[0]][self.pos[1]] == "D":
                print(f"Room {self.pos} is dirty, cleaning...")
            else:
                print("Room is clean...")
                print("Moving...")
            self.env[self.pos[0]][self.pos[1]]= random.choice(("C", "D"))
            self.pos = [self.pos[0] + moves[next_move_idx][0], self.pos[1] + moves[next_move_idx][1]]
            next_move_idx = (next_move_idx + 1)%len(moves)
            self.display()
c = Cleaner()
```

Output:

```
Current percept seq:  []
Current env:  ['D', 'D']
Room 0 is dirty, cleaning...
Current percept seq:  [(0, 'D')]
Current env:  ['C', 'D']
Current percept seq:  [(0, 'D')]
Current env:  ['D', 'D']
Room 1 is dirty, cleaning...
Current percept seq:  [(0, 'D'), (1, 'D')]
Current env:  ['D', 'C']
Current percept seq:  [(0, 'D'), (1, 'D')]
Current env:  ['D', 'D']
Room 0 is dirty, cleaning...
Current percept seq:  [(0, 'D'), (1, 'D'), (0, 'D')]
Current env:  ['C', 'D']
Current percept seq:  [(0, 'D'), (1, 'D'), (0, 'D')]
Current env:  ['C', 'D']
Room 1 is dirty, cleaning...
Current percept seq:  [(0, 'D'), (1, 'D'), (0, 'D'), (1, 'D')]
Current env:  ['C', 'C']
Current percept seq:  [(0, 'D'), (1, 'D'), (0, 'D'), (1, 'D')]
Current env:  ['C', 'D']
Room is clean...
Moving right...
```

Program 2
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

8 puzzle using DFS

Algorithm:

Lab-3 8-puzzle problem using DFS and MD

→ 1. ~~Using DFS~~ Initial Setup
We assume initial array and final array (both 3×3) are defined with random integers from 0-8 placed (0 = empty space)

→ Algorithm using DFS
~~def dfs (start, end):~~
~~stack = []~~
~~stack.append (hash(start))~~
~~while stack[-1] != end:~~
~~valid-moves = getMoves(stack[-1])~~
~~Stack = [] for move in valid-moves:~~

```
def dfs (start, end, zeroPos):
    stack = [] ; visited = []
    stack.append (start)
    while stack[-1] != end:
        valid-moves = getMoves (stack[-1])
        for move in valid-moves:
            new-board = board.copy()
            swap (new-board[zeroPos],
                  new-board[zeroPos + move])
            if new-board not in visited:
                stack.append (new-board)
                visited.append (new-board)
            else:
                while stack.pop() != new-board:
                    continue
            continue
    return stack[-1]
```

Eg:

Start =

2	3	4
5	7	1
8	6	0

2+2+ +2+1+3+ 2+ + 0

MD = 19

Possible Moves =

(1)

2	3	4
5	7	1
8	0	6

MD = 19

(2)

2	3	4
5	7	0
8	6	1

MD = 19

Go with (1)

Possible Moves =

2	3	4
5	7	1
0	8	6

MD = 17

2	3	4
5	0	1
8	7	6

MD = 17

X

2	3	4
5	7	1
8	6	0

Go with (1)

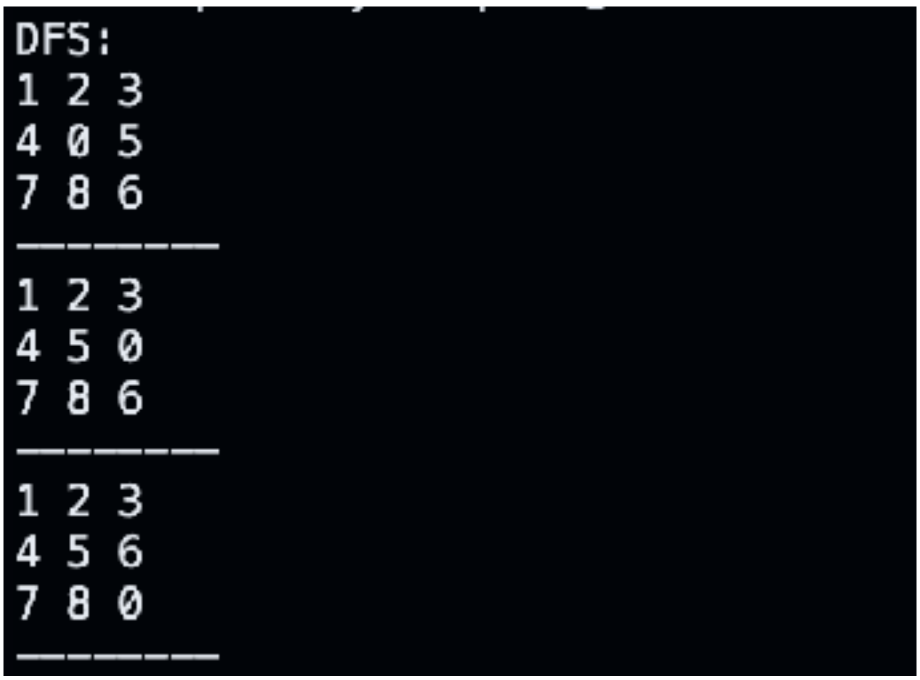
⋮

Code:

```
import heapq

class Puzzle:
    def __init__(self):
        self.board = [
        [1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]
    ]
        self.end = [
        [2, 8, 1],
        [0, 4, 3],
        [7, 6, 5]
    ]
    def getMoves(self, board):
        zero_pos = self.zero_index(board)
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        valid_moves = []
        for move in moves:
            if 0 <= zero_pos[0] + move[0] < 3 and 0 <= zero_pos[1] + move[1] < 3:
                valid_moves.append(move)
        return valid_moves

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

    def bhash(self, board):
       return tuple(map(tuple, board))

    def display(self, board):
        for ls in board:
            print(*ls)

    def manhattan_distance(self, state):
        """Calculate the total Manhattan distance of the state."""
        distance = 0
        for i in range(9):
            old = self.get_index(i, state)
```

```
            final = self.get_index(i, self.end)
            distance += (abs(final[0] - old[0]) + abs(final[1] - old[1]))
        return distance

    def get_index(self, el, board):
        for i in range(3):
            for j in range(3):
                if board[i][j] == el:
                    return [i, j]

    def misplaced(self, state):
        misplaced = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != self.end[i][j]:
                    misplaced += 1
        return misplaced


      def dfs(self):
        stack = []
        visited = []
        stack.append(self.board)
        visited.append(self.bhash(self.board))
        while stack:
            top = stack[-1]
            if self.bhash(top) == self.bhash(self.end):
                break
            valid_moves = self.getMoves(top)
            added = False
            # print(zeroPos, valid_moves)
            for move in valid_moves:
                new_board = [row[:] for row in top]
                zeroPos = self.zero_index(new_board)
                newPos = [zeroPos[0] + move[0], zeroPos[1] + move[1]]
                new_board[newPos[0]][newPos[1]], new_board[zeroPos[0]][zeroPos[1]] =
new_board[zeroPos[0]][zeroPos[1]], new_board[newPos[0]][newPos[1]]
                if self.bhash(new_board) not in visited:
                    stack.append(new_board)
                    visited.append(self.bhash(new_board))
                    added = True
                    break
```

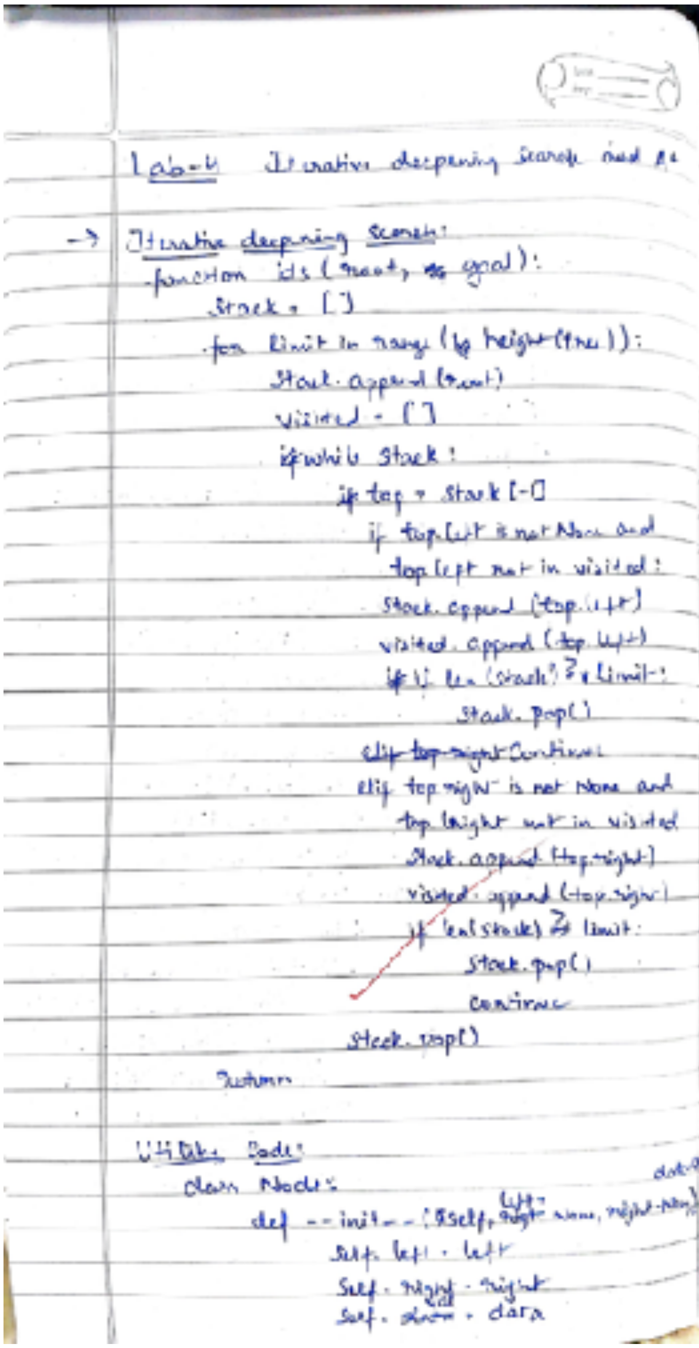
```
            if not added:
                stack.pop()
        while stack:
            self.display(stack.pop(0))
            print("--------")


c = Puzzle()
print('DFS: ')
c.dfs()
# print(c.zero_index("123405678"))
```



```
DFS:
1 2 3
4 0 5
7 8 6
--------
1 2 3
4 5 0
7 8 6
--------
1 2 3
4 5 6
7 8 0
--------
```

Output:

Iterative Deepening Search



Lab-4 Iterative deepening search and ...

→ Iterative deepening Search:

```
function ids(root, goal):
    stack = []
    for limit in range(height(tree)):
        stack.append(root)
        visited = []
        while stack:
            top = stack[-1]
            if top.left is not None and
                top.left not in visited:
                stack.append(top.left)
                visited.append(top.left)
                if len(stack) > limit:
                    stack.pop()
            elif top.right continue
            elif top.right is not None and
                top.right not in visited
                stack.append(top.right)
                visited.append(top.right)
                if len(stack) > limit:
                    stack.pop()
                    continue
            stack.pop()
    return
```

Utility Code:

```
class Node:
    def __init__(self, left=None, right=None, data)
        self.left = left
        self.right = right
        self.data = data
```

Running algorithm on given graph:

Goal State = F

Limit = 0
Explored: Y (In order)

Limit = 1
Explored - Y P X (In order)

Limit = 2
Explored - Y P R S X F H (In order)

Goal state found, therefore end.

Code:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
```

```
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

def iddfs(root, goal):
    for d in range(100000):
        res = dls(root, goal, d)
        if res:
            return "Found"
    return "Not Found"

def dls(root, goal, depth):
    if depth == 0:
        if root.value == goal:
            return True
        return False
    for child in root.children:
        if dls(child, goal, depth - 1):
            return True
    return False

root = TreeNode('Y')
node2 = TreeNode('P')
node3 = TreeNode('X')
node4 = TreeNode('R')
node5 = TreeNode('S')
node6 = TreeNode('F')
node7 = TreeNode('H')

root.add_child(node2)
root.add_child(node3)
node2.add_child(node4)
node2.add_child(node5)
node3.add_child(node6)
```

```
node3.add_child(node7)

print(iddfs(root, 'F'))
```

Output

```
Found
```

Program 3

Implement A* search algorithm

Algorithm:

→ A* algorithm on 8-puzzle problem:

Taking f(x) as Manhattan Distance

g(x) as No. of misplaced t[iles]

- Manhattan distance:

```
def getMD(boardFr, endgoal):
    ~~for i in range (9):~~
        ~~old = get_index(i, board)~~
        ~~end = get index (i, end)~~
    total_MD = 0
    for i in range (9):
        old = get_index(i, board)
        end = get_index (i, end)
        total_md += (abs(end[0] - old[0])
                     + abs(end[1] - old[1]))
    return total_MD
```

- No. of misplaced tiles:

```
def misplaced (board, end):
    total = 0
    for i in range (3)
        for j in range (3)
            if board[i][j] != end[i][j]:
                total += 1
    return total
```

Using A* with the following start and end states:

Initial :

1	2	3
8	0	4
7	6	5

Goal :

2	8	1
0	4	3
7	6	5

h(x) + f(x) = ?

g(x) = g(x) = 6

f(x) = 2 + 1 + 1 + 2 + 1 + 1 + 0

First Move:

Possible States:

①

1	0	3
8	2	4
7	6	5

f(x) = 10

g(x) = 6

h(x) = 16

②

1	2	3
8	6	4
7	0	5

f(x) = 11

g(x) = 7

h(x) = 19

③

1	2	3
0	8	4
7	6	5

f(x) = 6

g(x) = 5

h(x) = 11

Code:

```
import heapq

def manhattan(curr, goal):
    ans = 0
    for i in range(3):
        for j in range(3):
            for k in range(3):
                for l in range(3):
                    if goal[i][j] == curr[k][l]:
                        ans += abs(i - k) + abs(j - l)
    return ans

def astar(start, goal):
    open_set = []
    heapq.heappush(open_set, (manhattan(start, goal), start))
    close_set = set()
    gscore = {}
    gscore[tuple(map(tuple, start))] = 0
    parent = {}

    while open_set:
        _, curr = heapq.heappop(open_set)
        if curr == goal:
            return path(parent, curr)
        close_set.add(tuple(map(tuple, curr)))
        for neighbour in neighbours(curr):
            if tuple(map(tuple, neighbour)) in close_set:
                continue
            new_g = gscore[tuple(map(tuple, curr))] + 1
            if tuple(map(tuple, neighbour)) not in gscore or new_g < gscore[tuple(map(tuple, neighbour))]:
                parent[tuple(map(tuple, neighbour))] = curr
                gscore[tuple(map(tuple, neighbour))] = new_g
                heapq.heappush(open_set, (new_g + manhattan(neighbour, goal), neighbour))
    return "No solution"
```

```
def neighbours(curr):
    n = []
    x, y = 0, 0
    directions = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    for i in range(3):
        for j in range(3):
            if curr[i][j] == 0:
                x, y = i, j
                break
    for dx, dy in directions:
        if 0 <= x + dx < 3 and 0 <= y + dy < 3:
            new_state = [row.copy() for row in curr]
            new_state[x][y], new_state[x + dx][y + dy] = new_state[x + dx][y + dy], new_state[x][y]
            n.append(new_state)
    return n

def path(parent, curr):
    fol = [curr]
    while tuple(map(tuple, curr)) in parent:
        curr = parent[tuple(map(tuple, curr))]
        fol.append(curr)
    return list(reversed(fol))

start = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
goal = [[2, 8, 1], [0, 4, 3], [7, 6, 5]]
result = astar(start, goal)
if result != "No solution":
    for ind,state in enumerate(result):
        print(f"Step: {ind}")
        for row in state:
            print(row)
        print()
    print("Goal Reached")
else:
```

print(result)

Output:

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

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

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

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

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

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

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

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

Step: 9
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]

Goal Reached
```

Program 4
Implement Hill Climbing search algorithm

Algorithm:

→ Hill Climbing Algorithm:

Initial Setup:

Consider a mathematical expression which we have to attempt to maximise

```
def Cost(x):
    return math.cos(x)

def hill_climbing(initial_sol = 0, step = 0.01,
                  max_steps = 1000):
    current = initial_sol
    best = current
    for _ in range(max_steps):
        if neighbours = [current + step,
                         current - step]
        neighbours.sort(key = lambda x:
                        cost(x))
        current = neighbours[-1]
        if ~~best~~ cost(best) > cost(current):
            best = current
            ~~current = best~~
        else:
            break
    return best
```

→ Initialization Code:

```
initial_sol = random.uniform(-10, 10)
steps = random.choice(0.1, 0.001)
print(hill_climbing(initial_sol, steps))
```

Hill Climbing output:

cost function: math.sin(x)

final Solution: -4.71634

Code:

```
import math
import random
def cost(x):
    return math.sin(x)

def hill_climbing(initial_sol=0, steps=0.01, max_steps=1000):
    print(f"Initial sol: {initial_sol}; steps: {steps}")
    current = initial_sol
    best = current
    for _ in range(max_steps):
        neighbor = [current + steps, current - steps]
        # print(neighbor)
        neighbor.sort(key=lambda x : cost(x))
        current = neighbor[-1]
        if cost(best) < cost(current):
            best = current
        else:
            break
        print(f"Current: {current}, Best: {best}")
    return best
initial_sol = random.uniform(-10, 10)
steps = random.choice((00.01, 0.001))
print(hill_climbing(initial_sol, steps))
```

Output:

```
Current: 7.9808723301777995, Best: 7.9808723301777995
Current: 7.9708723301778, Best: 7.9708723301778
Current: 7.9608723301778, Best: 7.9608723301778
Current: 7.9508723301778, Best: 7.9508723301778
Current: 7.9408723301778, Best: 7.9408723301778
Current: 7.930872330177801, Best: 7.930872330177801
Current: 7.920872330177801, Best: 7.920872330177801
Current: 7.910872330177801, Best: 7.910872330177801
Current: 7.900872330177801, Best: 7.900872330177801
Current: 7.890872330177801, Best: 7.890872330177801
Current: 7.880872330177802, Best: 7.880872330177802
Current: 7.870872330177802, Best: 7.870872330177802
Current: 7.860872330177802, Best: 7.860872330177802
Current: 7.850872330177802, Best: 7.850872330177802
7.850872330177802
```

Program 5

Simulated Annealing

Algorithm:

Lab 5: Simulated Annealing algorithm

General Algorithm:

```
def annealing (init_sol, temp, cooling, final_temp)
    current ← init_sol
    new ← current
    while temp > final_temp:
        ~~neighbour ← getNeighbor (current)~~
        ~~if current cost (neighbour~~
        new = getNeighbor (current)
        if deltaE ← cost (new) − cost (current)
        if deltaE < 0 :
            current = new
        else if random() < e^(-deltaE/temp) :
            current = new
        temp *= cooling
    return current
```

→ Utility functions:

```
def objective_function (x)
    ## Define any mathematical objective
    ## function
    return x^4 + 5 + sin(x 5^4 − 1 + x)

def cost (sol)
    return objective_function(sol)
```

→ ~~Initialization Code:~~
~~initial_sol ←~~

```
def get_neighbor (sol):
    return sol + random (-1, 1)
# Adding a random perturbation to solution
```

→ Initialization code:

```
initial_solution = random (-10, 10)
# initial solution b/w -10, 10
initial_temperature = 10
final_temperature = 0.1
cooling_factor = 0.99
print (annealing (initial_solution, initial_temp, cooling, final_temperature))
```

Code:

```
import random
import math

class Annealing:
  def __init__(self) -> None:
    self.initial_sol = random.uniform(-10, 10)
    self.temp = 10
    self.cooling = 0.99
    self.final = 0.01
    self.annealing()
  def cost(self, x):
    # return x**2
    return x**4 + 5*math.sin(5*math.pi*x)
  def getNeighbors(self, sol):
    return sol + random.uniform(-1, 1)
  def annealing(self):
    current = self.initial_sol
    new = current
    best = current
    while self.temp > self.final:
      new = self.getNeighbors(current)
      dE = self.cost(new) - self.cost(current)
      print(f"Temp diff: {((self.temp - self.final)*100)/self.temp:.2f}%; Current sol: {current}; New sol: {new}; Best: {best}")
      if dE < 0 or random.random() < math.exp(-dE/self.temp):
        current = new
      if self.cost(new) < self.cost(best):
        best = new
      self.temp *= self.cooling
    print(f"Final solution: {best}")

c = Annealing()
```

Output:

```
Temp diff: 19.28%; Current sol: -0.5002454071369002; New sol: 0.4364917884841042; Best: 0.3017868187198611
Temp diff: 18.47%; Current sol: -0.5002454071369002; New sol: 0.32168017051042397; Best: 0.3017868187198611
Temp diff: 17.64%; Current sol: -0.5002454071369002; New sol: -0.8803105099726305; Best: 0.3017868187198611
Temp diff: 16.81%; Current sol: -0.5002454071369002; New sol: -0.057881220427036695; Best: 0.3017868187198611
Temp diff: 15.97%; Current sol: -0.5002454071369002; New sol: -1.0888795194246808; Best: 0.3017868187198611
Temp diff: 15.12%; Current sol: -0.5002454071369002; New sol: 0.33629024192800117; Best: 0.3017868187198611
Temp diff: 14.27%; Current sol: -0.5002454071369002; New sol: -1.2232165646664688; Best: 0.3017868187198611
Temp diff: 13.40%; Current sol: -0.5002454071369002; New sol: 0.15430243856813264; Best: 0.3017868187198611
Temp diff: 12.53%; Current sol: -0.5002454071369002; New sol: -0.0662229695212678; Best: 0.3017868187198611
Temp diff: 11.64%; Current sol: -0.5002454071369002; New sol: 0.4842211037532054; Best: 0.3017868187198611
Temp diff: 10.75%; Current sol: -0.5002454071369002; New sol: -0.9684694626478614; Best: 0.3017868187198611
Temp diff: 9.85%; Current sol: -0.5002454071369002; New sol: -0.424334393511431; Best: 0.3017868187198611
Temp diff: 8.94%; Current sol: -0.5002454071369002; New sol: 0.2996205263624432; Best: 0.3017868187198611
Temp diff: 8.02%; Current sol: 0.2996205263624432; New sol: -0.4090653806467661; Best: 0.2996205263624432
Temp diff: 7.09%; Current sol: 0.2996205263624432; New sol: 0.2590605338320269; Best: 0.2996205263624432
Temp diff: 6.15%; Current sol: 0.2996205263624432; New sol: 1.0364699436308205; Best: 0.2996205263624432
Temp diff: 5.20%; Current sol: 0.2996205263624432; New sol: -0.256078698773591; Best: 0.2996205263624432
Temp diff: 4.24%; Current sol: 0.2996205263624432; New sol: 1.0421712775865477; Best: 0.2996205263624432
Temp diff: 3.28%; Current sol: 0.2996205263624432; New sol: 0.3999848075527662; Best: 0.2996205263624432
Temp diff: 2.30%; Current sol: 0.2996205263624432; New sol: 0.24764703780765895; Best: 0.2996205263624432
Temp diff: 1.31%; Current sol: 0.2996205263624432; New sol: 0.18455253675984933; Best: 0.2996205263624432
Temp diff: 0.32%; Current sol: 0.2996205263624432; New sol: 0.5943383719613244; Best: 0.2996205263624432
Final solution: 0.2996205263624432
```

Program 6

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

Algorithm:

Lab-7 Propositional Logic problems

KB:

$R_1 \Rightarrow$ Alice mother of Bob

$R_2 \rightarrow$ Bob father of Charlie

$R_3 \Rightarrow$ Father is a parent

$R_4 \Rightarrow$ Mother is a parent

$R_5 \Rightarrow$ All parents have children

$R_6 \Rightarrow$ If someone is parent, children are sibling

$R_7 \Rightarrow$ Alice ~~are~~ married to David

Hypothesis:

Charlie is sibling of Bob

Entailment:

Let R_8 : Bob is parent

$R_2, R_5 \models R_8$

Let R_9 : Alice is parent

$R_1, R_5 \models R_9$

Let R_{10} : Bob ~~has no siblings~~ has a child Charlie

$R_8, R_5, R_2 \models R_{10}$

Let R_{11} : Bob is sibling to Charlie

$R_{10}, R_8, R_6 \models R_{11}$

$\therefore$ We can prove hypothesis as true

$KB \models R_{11}$

Code:

```
#Create a knowledge base using propositional logic and show that the given query entails the
knowledge base or not.
import itertools

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Evaluate the expressions for each combination
```

```
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Replace True/False with string "True"/"False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Convert boolean values of a, b, c to "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

    # Additional output for combinations where both KB and query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)" # Knowledge Base
query = "a or b" # Query to evaluate
```

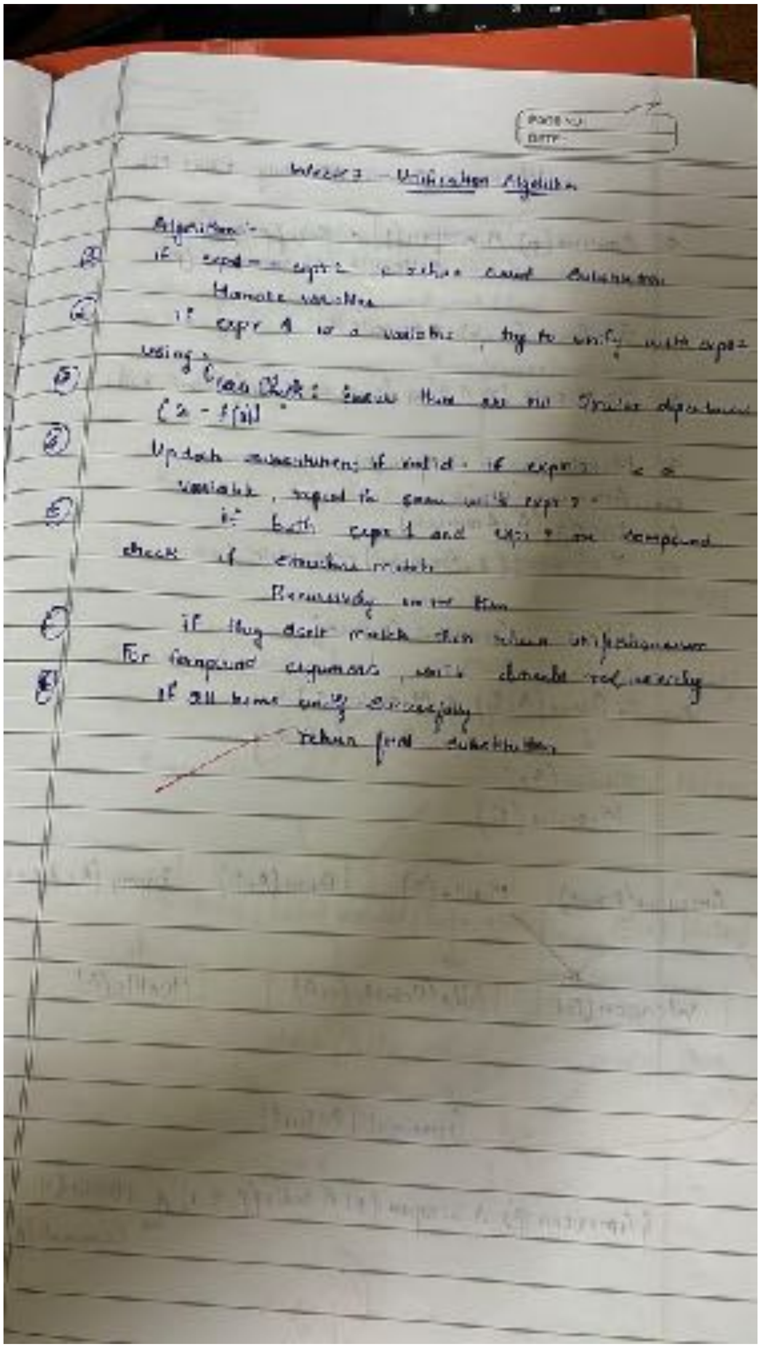
# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)

```
a       b       c       KB                  Query
False False False False                    False
False False True  False                    False
False True  False False                    True
False True  True  True                     True
True  False False True                     True
True  False True  False                    True
True  True  False True                     True
True  True  True  True                     True

Combirations where both KB and Query are True:
a       b       c       KB                  Query
False True  True  True                     True
True  False False True                     True
True  True  False True                     True
True  True  True  True                     True
```

Program 7

Implement unification in first order logic

Algorithm:


Week 7 - Unification Algorithm

Algorithm:-

1. If expr1 == expr2 : return current substitution
   Handle variables
2. if expr 1 is a variable, try to unify with expr2 using:
3. Occurs Check: Ensure there are no circular dependencies (x = f(x))
4. Update substitution if valid. If expr2 is a variable, repeat the same with expr 1.
5. if both expr 1 and expr 2 are compound check if structures match
   Recursively unify them
6. if they don't match then return unification failure
   For compound arguments, unify elements recursively
7. if all terms unify successfully
   return final substitution

Code:

```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list): # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst: # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst: # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x): # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst

def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {} # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y: # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower(): # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower(): # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list): # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"
```

```
        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]: # If the predicates/functions are different
            return "FAILURE"

        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]): # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else: # If x and y are different constants or non-unifiable structures
        return "FAILURE"

def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
```

```
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term

    return parse_term(input_str)

# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)
```

```
        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

if __name___== "_main_":
    main()
```

Output:

```
Enter the first expression (e.g., p(x, f(y))): p(x, f(y)))
Enter the second expression (e.g., p(a, f(z))): p(a, f(z)))
Expression 1: ['p', '(x', ['f', '(y)))']]
Expression 2: ['p', '(a', ['f', '(z)))']]
Result: Unification Successful
Substitutions: {'(x': '(a', '(y)))': '(z)))'}
```

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

Lab - 9 First Order Logic Forward Chaining

Considering Statement

As per law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all missiles were sold to it by Robert, who is American

Prove Robert is a criminal.

Deduced Statements:

American (x) ∧ Weapon(y) ∧ Sold (z, y, x) ∧ Hostile(z) ⟹ Criminal (x)

Enemy (A, America)

~~Has (mis~~

∀ ~~x Missile(x) → W~~

∃ x Missiles (x) ∧ Owns (A, x)

∀ x Missiles (x) ∧ Owns (A, x) ⟹ Sold (A, x, Robert)

American (Robert)

We can deduce that missiles are weapons

∀ x Missile (x) ⟹ Weapon (x)

We can also deduce that an enemy of America is hostile

∀ x Enemy (x, America) ⟹ Hostile (x)

∴ From First Level of atomic Statements

American (Robert) Missiles (y) Owns (A, y) Enemy (A, America)

Weapons (y) Hostile (A)

Sold (A, y, Robert)

Criminal (Robert)

Algorithm:

~~forward chain~~ (KB, α)

Assuming KB represents all current knowledge

α represents atomic query to be validated

while new != {}

for rule in KB do

($p_1 \wedge p_2 \ldots \wedge p_n \to q$) ← Standardize variables (rule)

~~for θ in~~ ~~Reset~~

($\theta, p_1 \wedge p_2 \ldots \wedge p_n$) = ($\theta, p_1' \wedge p_2' \ldots p_n'$)

for p_1', p_2' ... in KB

q' = Subst(θ, q)

if q' can not unify some sentence in KB or new

new += q'

ϕ = Unify (q', α)

if not ϕ return False

else return θφ

~~return~~ KB += new

return False

Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be
    made """ # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and
'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
```

```
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

Output

```
No more inferences can be made.
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
```

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:

General Algorithm:

- Consider all atomic sentences in KB
- ~~Generate~~ Add sentences to KB s.t all can be inferred one layer from current level of sentences and required atomic sentences already in KB
- If not find sentence not in KB, return False, else rules used.

→ Resolution using FOL

Consider the following sentences:

1) food (x) → likes(John, x)
2) food (Apple)
3) food (Vegetables)
4) ¬ eats (y, z) ∧ killed (y) → food (z)
5) eats (Anil, Peanuts)
6) alive (Anil)
6) killed (x) ∨ alive (x)
7) ¬ killed (x) ∨ ¬ alive (x)

To prove α : likes (John, Peanuts)

Adding negation (¬ likes (John, Peanuts)) to KB and resolving

¬ likes (John, Peanuts) ; ¬ food (x) ∨ likes (John, x) ; {x / peanuts}

¬ food (Peanuts) ; ¬ eats (y, z) ∨ killed (y) ∨ food (z) ; {z / Peanuts}

¬ eats (y, Peanuts) ∨ killed (y) ∨ food (Peanuts) ; eat (Anil, Peanuts) ; {y / Anil}

killed (Anil) ∨ ~~food (Peanuts)~~ ; ¬ alive (x) ∨ ¬ killed (x) ; {x / anil}

¬ alive (Anil) ; alive (Anil)

{ }

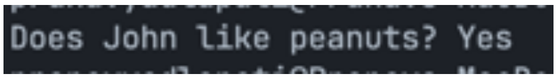
Results in empty set, therefore contradiction

∴ likes (John, Peanut) is true.

Code:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)", # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule: # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule: # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule: # Handle negation
            sub_pred = rule[4:] # Remove "not "
            return not resolve(sub_pred.strip())
        else: # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")
```

```
    # Default to False if no rule or fact applies
    return False

# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```



Program 10

Implement Alpha-Beta Pruning and Minimax Algorithm

Algorithm:

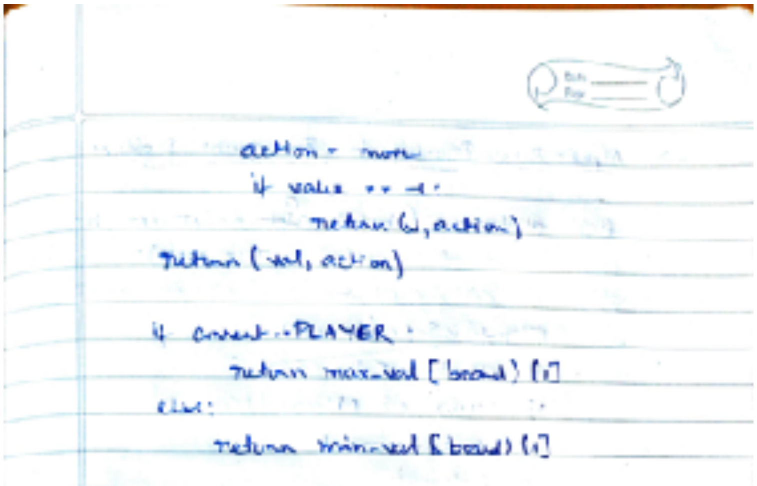
→ Alpha-Beta Pruning for 8-queens problem

```
def alabprune(board, alpha = -INT_MAX, b = INT_MAX):

    if not isSafe(board):
        return False
    if row == N:
        return board

    for col ← 0 to N-1
        if safe(board, row, col):
            board[row][col] = 1
            result = alabprune(board,
                row+1, a, b)
            if res:
                return res
        board[row][col] = 0
        alpha = max(alpha, row)
        if alpha >= beta:
            break
    return ~~return Null~~ null
    for col ← 0 to N-1
        if safe(board, row, col):
            board[row][col] = 1
    result = alabprune(board, row+1,
            a, b)
        if res:
            return res
    board[row][col] = 0
    a = max(a, row)
    if a >= b:
        break
```

Lab 10 Minimax and Alpha-Beta Pruning

→ Minimax for Tic-Tac-Toe algorithm

```
def minimax(board):
    if terminal(board):
        return None

    def max_val(board):
        if terminal(board):
            return utility(board), None
        val = -INT_MAX
        action = None
        for move in possible_moves(board):
            v, act = min_val(result(board, move))
            if v > val:
                val = v
                action = move
                if v == 1:
                    return v, action
        return (v, action)

    def min_val(board):
        if terminal(board):
            return utility(board), None
        val = INT_MAX
        action = None
        for move in possible_moves(board):
            v, act = max_val(result(board, move))
            if v < val:
                value = v
```



```
action = move
    if value == -1:
        return (-1, action)
return (val, action)

if current == PLAYER:
    return max_val(board)[1]
else:
    return min_val(board)[1]
```

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node: # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) # Maximize alpha
            if beta <= alpha: # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Iterate over children of the minimizer node
```

```
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval) # Minimize beta
            if beta <= alpha: # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level
    return current_level[0] # Return the root node, which is a maximizer
# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

if __name__== "_main_":
    main()
```

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```

Output:

Minimax Algorithm:

```
def minimax(board):
    """
    Returns the optimal action for the current player on the board.
    """
    if terminal(board):
        return None

    def max_value(board):
        if terminal(board):
            return (utility(board), None)

        value = float('-inf')
        action = None
        for move in actions(board):
            # v = max(v, min_value(result(board, action)))
            v, act = min_value(result(board, move))
            if v > value:
                value = v
                action = move
                if v == 1:
                    return (v, action)

        return (v, action)

    def min_value(board):
        if terminal(board):
            return utility(board), None

        value = float('inf')
        action = None
        for move in actions(board):
            # v = max(v, min_value(result(board, action)))
            v, act = max_value(result(board, move))
            if v < value:
                value = v
                action = move
                if value == -1:
                    return (value, action)

        return (value, action)
```