VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Manav Kumar (1BM22CS348)

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
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 **Manav Kumar (1BM22CS348),** 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.

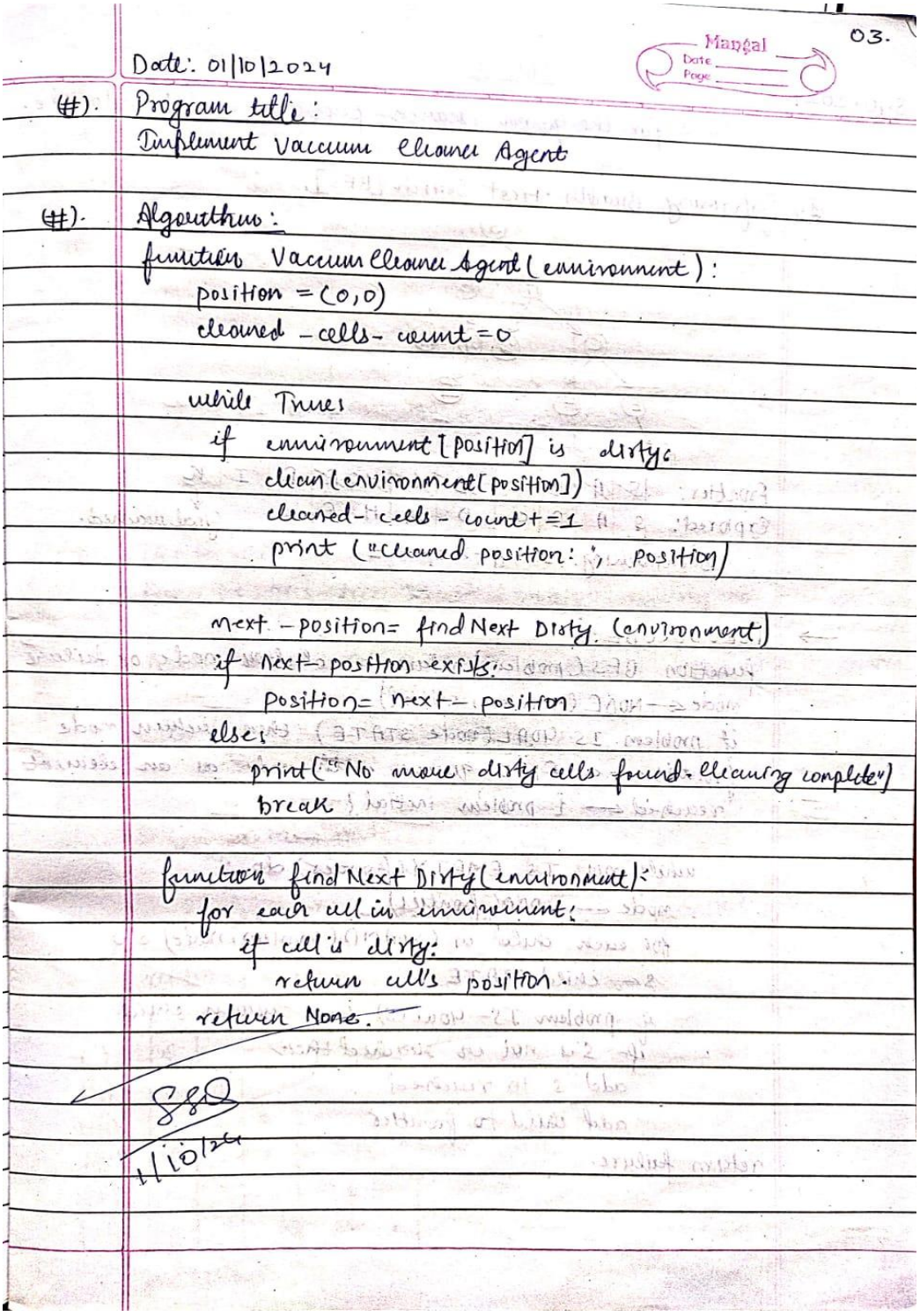
Namratha M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Manav-Kumar123/MANAV_1BM22CS348_AI

1/10/2024 **Implement Vacuum Cleaner Agent**

ALGORITHM AND STATE SPACE TREE:



03.

Mangal Date Page

Date: 01/10/2024

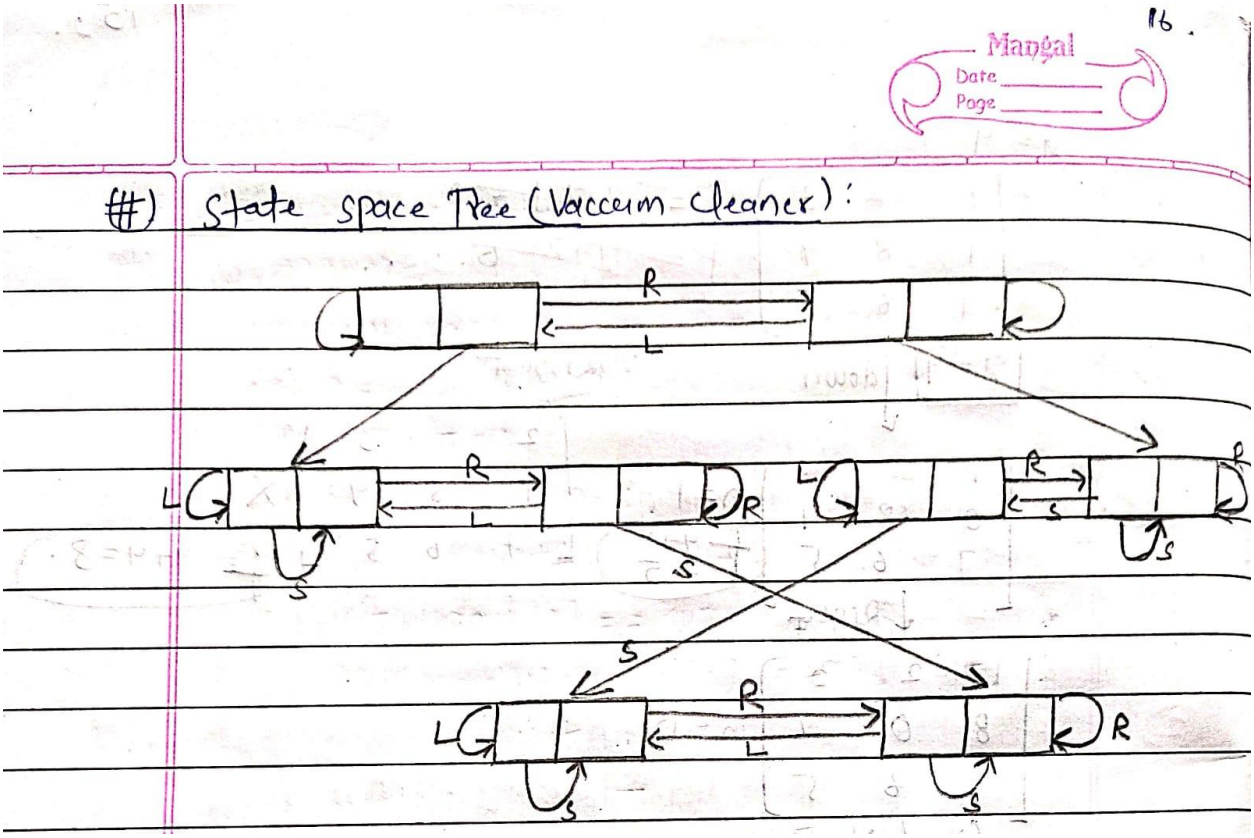
(#). Program title:

Implement Vaccuum Cleaner Agent

(#). Algorithm:

```
function VacuumCleanerAgent(environment):
    position = (0,0)
    cleaned_cells_count = 0

    while True:
        if environment[position] is dirty:
            clean(environment[position])
            cleaned_cells_count += 1
            print("cleaned position:", position)

        next_position = findNextDirty(environment)
        if next_position exists:
            position = next_position
        else:
            print("No more dirty cells found. cleaning completed")
            break

function findNextDirty(environment):
    for each cell in environment:
        if cell is dirty:
            return cell's position
    return None
```

SSD
1/10/24



CODE:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + " (0 for Clean, 1 for Dirty): ")
    status_input_complement = input("Enter status of other room: ")

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':

        print("Vacuum is placed in Location A")

        if status_input == '1':
            print("Location A is Dirty.")

            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")
```

```
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to the Location B.")
            cost += 1
            print("COST for moving RIGHT: " + str(cost))

            goal_state['B'] = '0'
            cost += 1



            print("COST for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B.")
            cost += 1
            print("COST for moving RIGHT: " + str(cost))

            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location B is already clean.")

else:
    print("Vacuum is placed in Location B")


    if status_input == '1':
        print("Location B is Dirty.")

        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING: " + str(cost))
        print("Location B has been Cleaned.")
```

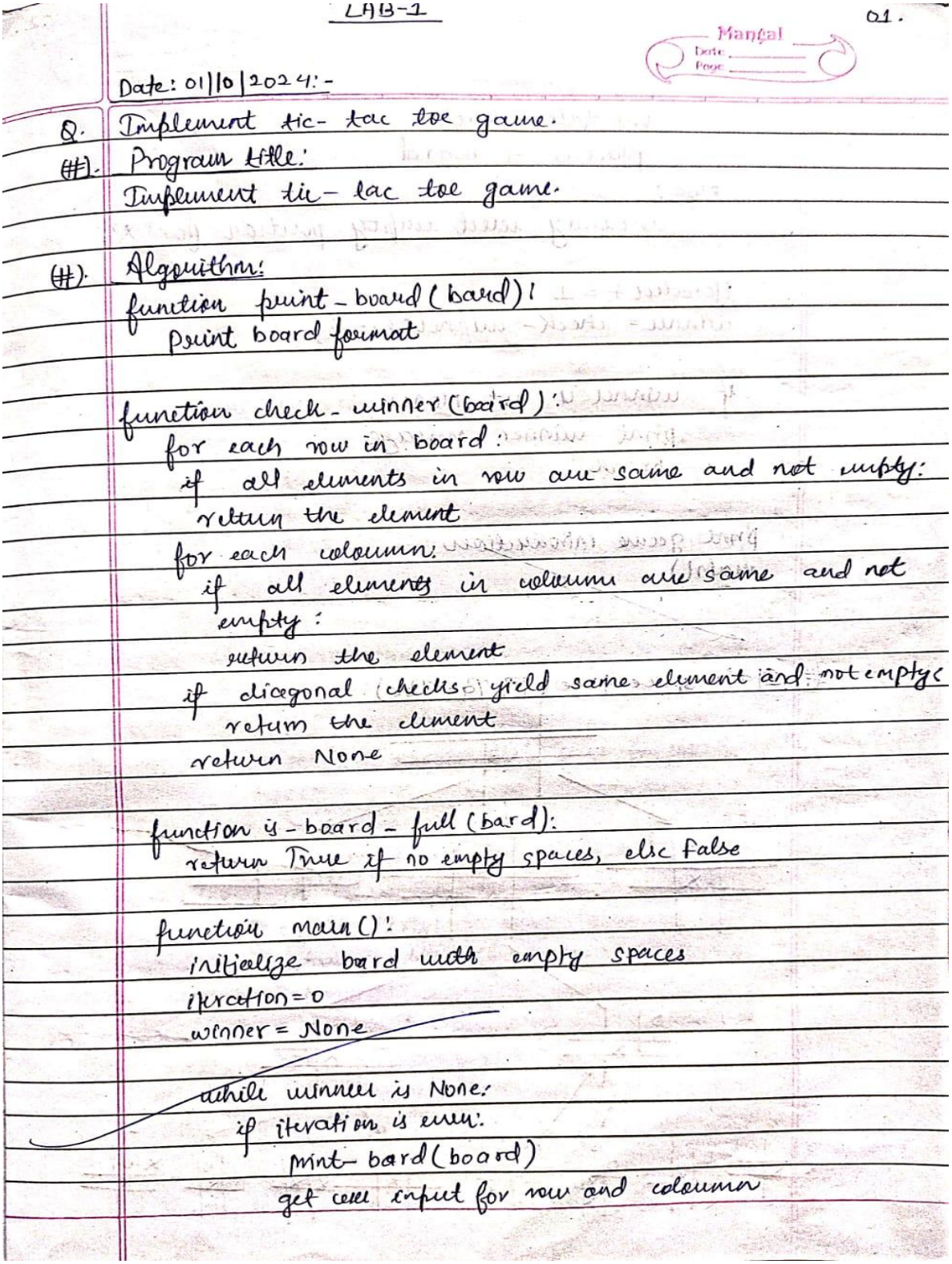
```
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A.")
                cost += 1
                print("COST for moving LEFT: " + str(cost))

                goal_state['A'] = '0'
                cost += 1
                print("COST for SUCK: " + str(cost))

                print("Location A has been Cleaned.")
            else:
                print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A.")
            cost += 1
            print("COST for moving LEFT: " + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("Cost for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location A is already clean.")


    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

OUTPUT:

```
Enter Location of Vacuum: A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room: 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT: 2
COST for SUCK: 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

1/10/2024 **Implement Tic –Tac –Toe Game**

ALGORITHM AND STATE SPACE TREE:



LAB-1

01.

Manjal

Date: 01/10/2024:-

Q. Implement tic- tac toe game.

(#). Program title:
Implement tic- tac toe game.

(#). Algorithm:
function print_board(board):
print board format

function check_winner(board):
for each row in board:
if all elements in row are same and not empty:
return the element
for each column:
if all elements in column are same and not empty:
return the element
if diagonal checks yield same element and not empty:
return the element
return None

function is_board_full(board):
return True if no empty spaces, else False

function main():
initialize board with empty spaces
iteration = 0
winner = None

while winner is None:
if iteration is even:
print_board(board)
get user input for row and column

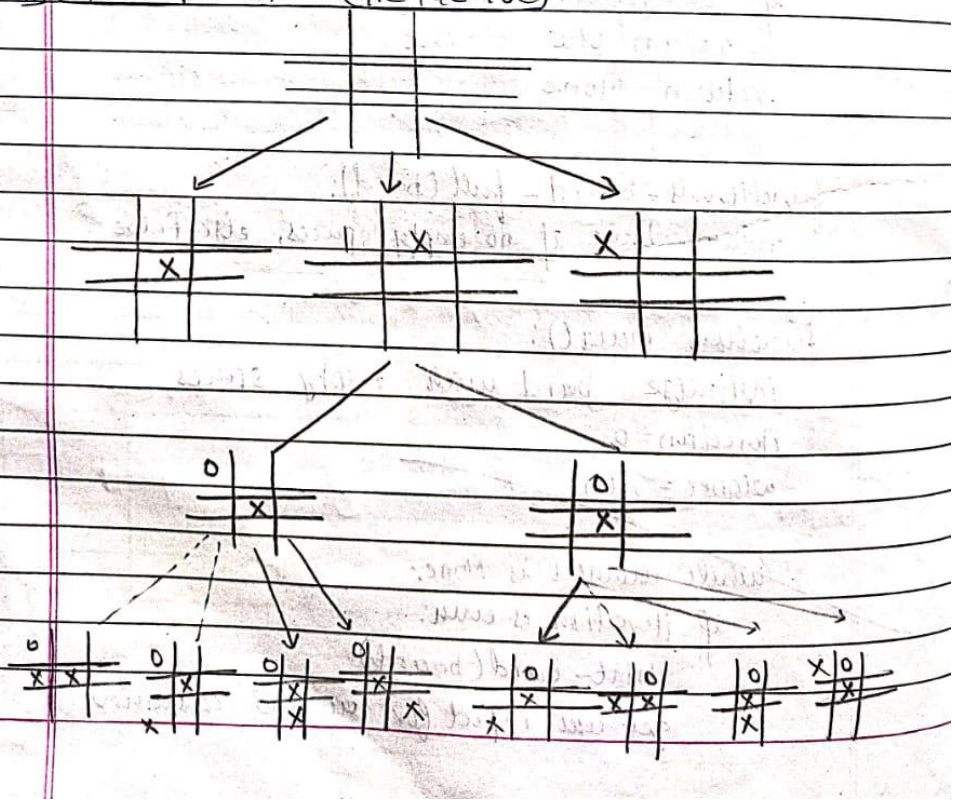
validate input
place 'O' in board
else:
randomly select empty position for 'X'.

iteration += 1
winner = check_winner(board)

if winner is not None:
print winner message
break

print game introduction
main().

State space tree (Tic Tac Toe):



CODE:

```
#Tic-Tac-Toe Game

board = {
   1: ', 2: ', 3: ',
   4: ', 5: ', 6: ',
   7: ', 8: ', 9: '
}

def printBoard(board):
   print(f"{board[1]} | {board[2]} | {board[3]}\n- + - + -\n{board[4]} | {board[5]} | {board[6]}\n-
+ - + -\n{board[7]} | {board[8]} | {board[9]}\n")

def spaceFree(pos):
   return board[pos] == '

def checkWin():
   winning_combinations = [
      (1, 2, 3), (4, 5, 6), (7, 8, 9),  # Rows
      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Columns
      (1, 5, 9), (3, 5, 7)              # Diagonals
   ]
   for combo in winning_combinations:
      if board[combo[0]] == board[combo[1]] == board[combo[2]] and board[combo[0]] != ':
         return True
   return False

def checkMoveForWin(move):
   winning_combinations = [
      (1, 2, 3), (4, 5, 6), (7, 8, 9),  # Rows
      (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Columns
      (1, 5, 9), (3, 5, 7)              # Diagonals
   ]
   for combo in winning_combinations:
      if board[combo[0]] == board[combo[1]] == board[combo[2]] == move:
         return True
   return False

def checkDraw():
   return all(board[key] != ' for key in board.keys())

def insertLetter(letter, pos):
   if spaceFree(pos):
      board[pos] = letter
      printBoard(board)
      if checkDraw():
```

```
            print("Draw")
        elif checkWin():
            if letter == 'X':
                print("Bot Wins!")
            else:
                print("You Win!")
        return
    else:
        print("Position taken, please pick a different position.")
        pos = int(input("Enter new position: "))
        insertLetter(letter, pos)
        return

player = 'O'
bot = 'X'

def playerMove():
    pos = int(input("Enter Position for O: "))
    insertLetter(player, pos)

def compMove():
    bestScore = -1000
    bestMove = None
    for key in board.keys():
        if board[key] == '':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ''
            if score > bestScore:
                bestScore = score
                bestMove = key

    if bestMove is not None:
        insertLetter(bot, bestMove)

def minimax(board, isMax):
    if checkMoveForWin(bot): return 1
    elif checkMoveForWin(player): return -1
    elif checkDraw(): return 0

    if isMax:
        bestScore = -1000
        for key in board.keys():
            if board[key] == '':
                board[key] = bot
                score = minimax(board, False)
```

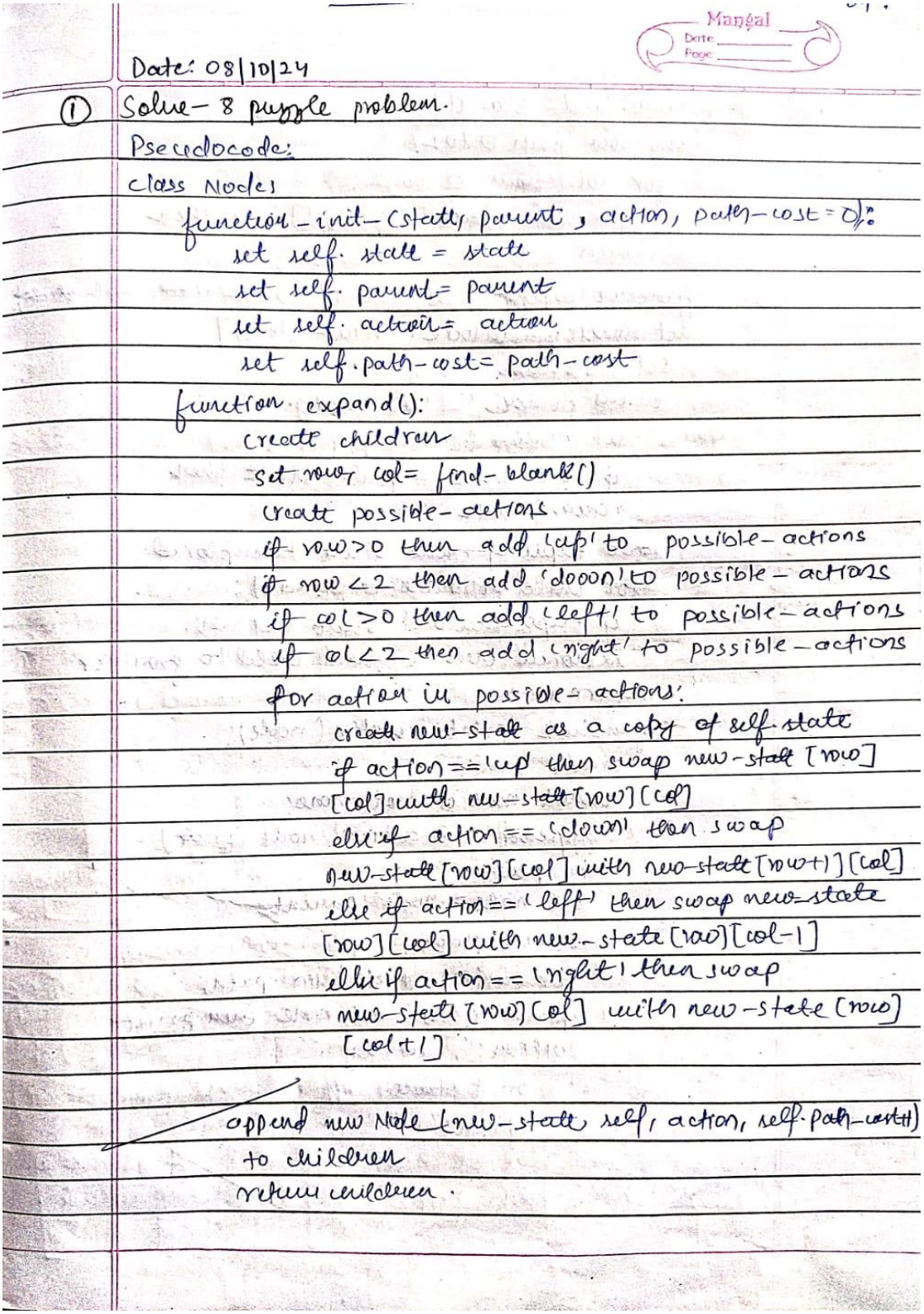
```
            board[key] = ''
            if score > bestScore: bestScore = score
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == '':
                board[key] = player
                score = minimax(board, True)
                board[key] = ''
                if score < bestScore: bestScore = score
        return bestScore

while True:
    playerMove()
    if checkWin() or checkDraw():
        break
    compMove()
    if checkWin() or checkDraw():
        break
```

OUTPUT:

```
o |  |
- + - + -
 |  |
- + - + -
 |  |

o |  |
- + - + -
 | x |
- + - + -
 |  |

o | o |
- + - + -
 | x |
- + - + -
 |  |

o | o | x
- + - + -
 | x |
- + - + -
 |  |

o | o | x
- + - + -
o | x |
- + - + -
 |  |

o | o | x
- + - + -
o | x |
- + - + -
x |  |

Bot Wins!
```

08/10/2024 **8 puzzle problem using DFS**

ALGORITHM AND STATE SPACE TREE:



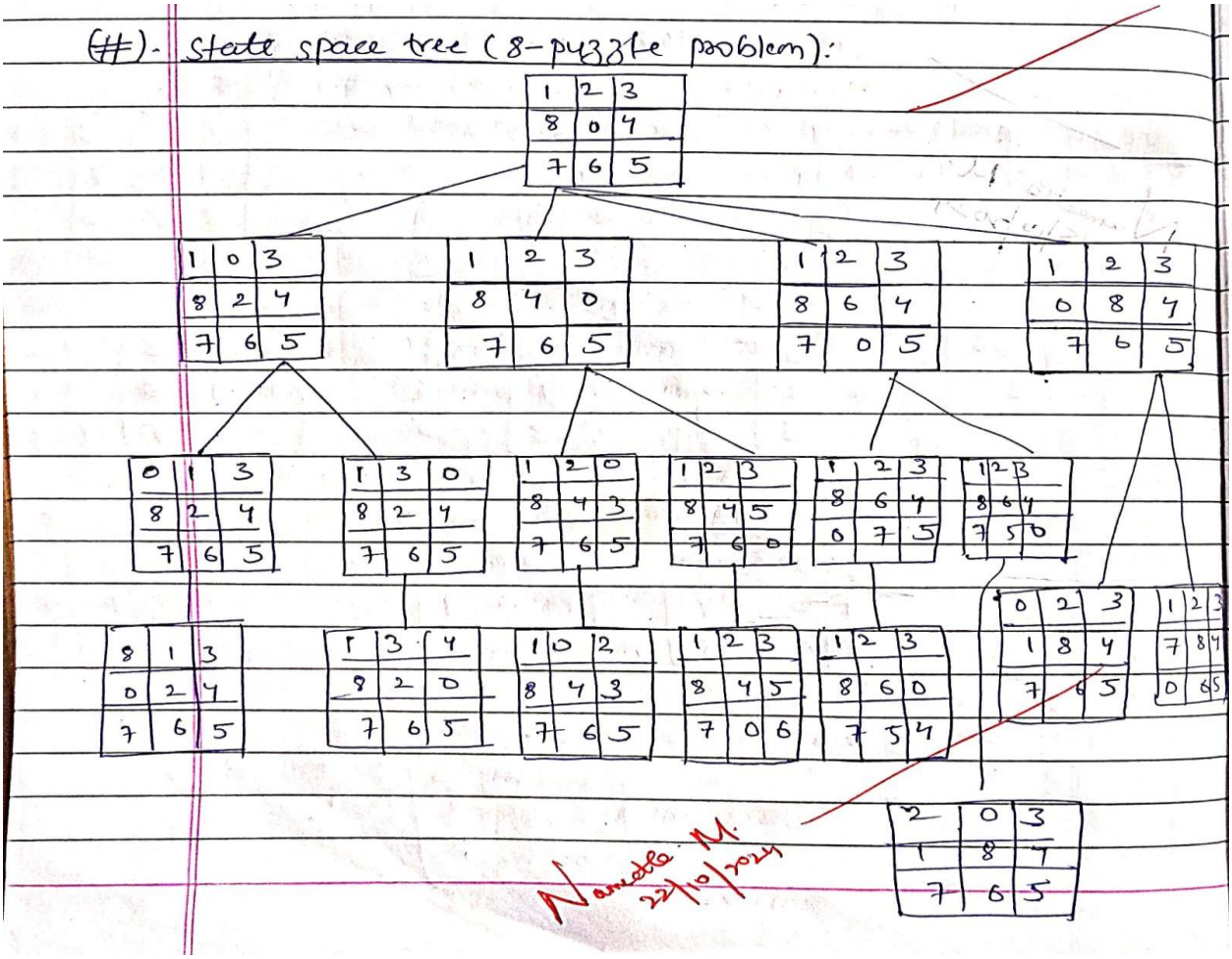
Date: 08/10/24

① Solve- 8 puzzle problem.

Pseudocode:

```
class Node:
    function -init- (state, parent, action, path-cost = 0):
        set self.state = state
        set self.parent = parent
        set self.action = action
        set self.path-cost = path-cost
    function expand():
        create children
        set row, col = find-blank()
        create possible-actions
        if row > 0 then add 'up' to possible-actions
        if row < 2 then add 'dooon' to possible-actions
        if col > 0 then add 'left' to possible-actions
        if col < 2 then add 'right' to possible-actions
        for action in possible-actions:
            create new-state as a copy of self.state
            if action == 'up' then swap new-state [row]
            [col] with new-state [row] [col]
            else if action == 'down' then swap
            new-state [row] [col] with new-state [row+1] [col]
            else if action == 'left' then swap new-state
            [row] [col] with new-state [row] [col-1]
            else if action == 'right' then swap
            new-state [row] [col] with new-state [row]
            [col+1]

            append new Node (new-state, self, action, self.path-cost+1)
            to children
        return children.
```

```
function find_blank():
    from row from 0 to 2:
        for col from 0 to 2:
            if self.state[row][col]==0 then
            return row, col
function depth_first_search(initial_state, goal_state)
set frontier = [Node(initial_state)]
set explored= empty set
while frontier is not empty:
    set node= frontier.pop()
    if node.state == goal_state then
    return node
    add tuple of node state to explored
    for child in node.expand()s
    if tuple of child state not in
    explored then append child to frontier
    return none
function print_solution(node):
    create path
    while node is not none:
        append(node.action, node.state)
        to path
        set node= node.parent
        reverse path()
        for (action, state) in path:
        if action is not none then print
        "action:", action
            print state
            print" "
```



CODE:

```
import numpy as np
from collections import deque

class Node:
    def __init__(self, state, parent, action):
        self.state = state  # The puzzle configuration (as a 2D array)
        self.parent = parent  # Parent node (used to trace back the solution)
        self.action = action  # Action taken to reach this state (e.g., 'up', 'down')

class Puzzle:
    def __init__(self, start, start_index, goal, goal_index):
        self.start = [start, start_index]  # Starting puzzle state and blank tile index
        self.goal = [goal, goal_index]  # Goal puzzle state and blank tile index
        self.solution = None  # To store the final solution path

    def neighbors(self, state):
        mat, (row, col) = state  # Puzzle state and the position of the blank tile (0)
```

```
    results = []

    # Move the blank tile up
    if row > 0:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row - 1][col]
        mat1[row - 1][col] = 0
        results.append(('up', [mat1, (row - 1, col)]))

    # Move the blank tile left
    if col > 0:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col - 1]
        mat1[row][col - 1] = 0
        results.append(('left', [mat1, (row, col - 1)]))

    # Move the blank tile down
    if row < 2:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row + 1][col]
        mat1[row + 1][col] = 0
        results.append(('down', [mat1, (row + 1, col)]))

    # Move the blank tile right
    if col < 2:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col + 1]
        mat1[row][col + 1] = 0
        results.append(('right', [mat1, (row, col + 1)]))

    return results

def solve_dfs(self):
    """Depth-First Search (DFS) implementation for solving the 8-puzzle."""
    # Initialize the frontier with the starting node (stack)
    start = Node(state=self.start, parent=None, action=None)
    frontier = [start]  # Using a list as a stack (LIFO)

    explored = []  # List to keep track of explored states

    while frontier:
        # Pop the node from the stack (LIFO)
        node = frontier.pop()

        # Check if the current state is the goal state
        if (node.state[0] == self.goal[0]).all():
```

```
            actions = []
            cells = []
            while node.parent is not None:
                actions.append(node.action)
                cells.append(node.state)
                node = node.parent
            actions.reverse()
            cells.reverse()
            self.solution = (actions, cells)
            return self.solution

        # Add the current state to the explored set
        explored.append(node.state)

        # Expand the neighbors of the current state
        for action, neighbor in self.neighbors(node.state):
            # If the neighbor state has not been explored and is not in the frontier
            if not any((explored_state[0] == neighbor[0]).all() for explored_state in explored):
                if not any((frontier_node.state[0] == neighbor[0]).all() for frontier_node in
frontier):
                    # Add the neighbor to the frontier
                    child_node = Node(state=neighbor, parent=node, action=action)
                    frontier.append(child_node)

    # No solution found
    raise Exception("No solution found.")

def print_solution(self):
    if self.solution is None:
        print("No solution available.")
        return

    # Print the solution path
    print("Initial State:")
    print(self.start[0], "\n")

    for action, state in zip(self.solution[0], self.solution[1]):
        print(f"Action: {action}")
        print(state[0], "\n")

    print("Goal Reached!")


# Function to take user input and convert it to a 3x3 matrix
def get_puzzle_input():
    print("Enter the 8-puzzle configuration (use 0 for the blank space):")
```

```
    puzzle = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (space-separated numbers): ").split()
        puzzle.append([int(num) for num in row])
    return np.array(puzzle)

# Get user input for start and goal states
start = get_puzzle_input()
goal = get_puzzle_input()

# Find the index of the blank tile (0) in the initial and goal states
start_index = (np.where(start == 0)[0][0], np.where(start == 0)[1][0])
goal_index = (np.where(goal == 0)[0][0], np.where(goal == 0)[1][0])

# Create the puzzle object
puzzle = Puzzle(start, start_index, goal, goal_index)

# Print the initial state before solving
print("Initial State:")
print(start, "\n")

# Solve the puzzle using DFS
puzzle.solve_dfs()

# Print the solution path
puzzle.print_solution()
```

OUTPUT:

```
Enter the 8-puzzle configuration (use 0 for the blank space):
Enter the 8-puzzle configuration (use 0 for the blank space):
Initial State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Initial State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Action: left
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Action: left
[[1 2 3]
 [4 5 6]
 [0 7 8]]

Action: up
[[1 2 3]
 [0 5 6]
 [4 7 8]]

Action: right
[[1 2 3]
 [5 0 6]
 [4 7 8]]

Action: right
[[1 2 3]
 [5 6 0]
 [4 7 8]]
```

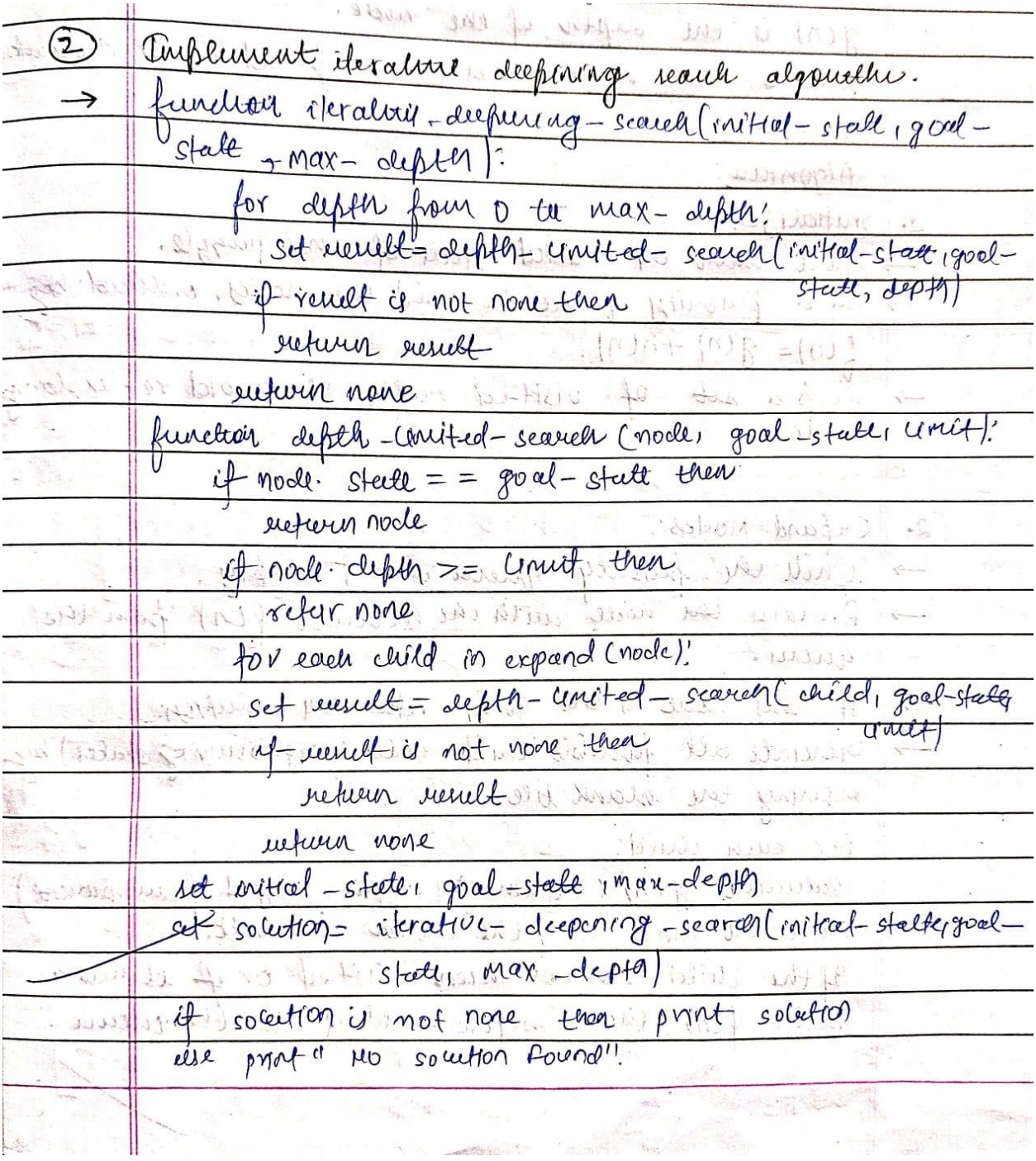
```
Action: down
[[1 2 3]
 [5 6 8]
 [4 7 0]]

Action: left
[[1 2 3]
 [5 6 8]
 [4 0 7]]

Action: left
[[1 2 3]
 [5 6 8]
 [0 4 7]]

Action: up
[[1 2 3]
 [0 6 8]
 [5 4 7]]

Action: right
[[1 2 3]
 [6 0 8]
 [5 4 7]]

Action: right
[[1 2 3]
 [6 8 0]
 [5 4 7]]

Action: down
[[1 2 3]
 [6 8 7]
 [5 4 0]]

Action: left
[[1 2 3]
 [6 8 7]
 [5 0 4]]
```

```
Action: left
[[1 2 3]
 [6 8 7]
 [0 5 4]]

Action: up
[[1 2 3]
 [0 8 7]
 [6 5 4]]

Action: right
[[1 2 3]
 [8 0 7]
 [6 5 4]]

Action: right
[[1 2 3]
 [8 7 0]
 [6 5 4]]

Action: down
[[1 2 3]
 [8 7 4]
 [6 5 0]]

Action: left
[[1 2 3]
 [8 7 4]
 [6 0 5]]

Action: left
[[1 2 3]
 [8 7 4]
 [0 6 5]]

Action: up
[[1 2 3]
 [0 7 4]
 [8 6 5]]
```

```
Action: right
[[1 2 3]
 [7 0 4]
 [8 6 5]]

Action: right
[[1 2 3]
 [7 4 0]
 [8 6 5]]

Action: down
[[1 2 3]
 [7 4 5]
 [8 6 0]]

Action: left
[[1 2 3]
 [7 4 5]
 [8 0 6]]

Action: left
[[1 2 3]
 [7 4 5]
 [0 8 6]]

Action: up
[[1 2 3]
 [0 4 5]
 [7 8 6]]

Goal Reached!
```

08/10/2024 **Iterative Deepening Search Algorithm**

ALGORITHM:



(2) Implement iterative deepening search algorithm.

→ function iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth from 0 to max_depth:
        set result = depth_limited_search(initial_state, goal_state, depth)
        if result is not none then
            return result
    return none

function depth_limited_search(node, goal_state, limit):
    if node.state == goal_state then
        return node
    if node.depth >= limit then
        return none
    for each child in expand(node):
        set result = depth_limited_search(child, goal_state, limit)
        if result is not none then
            return result
    return none

set initial_state, goal_state, max_depth
set solution = iterative_deepening_search(initial_state, goal_state, max_depth)
if solution is not none then print solution
else print "No solution found".

CODE:

```
def depth_limited_search(node, goal, depth):
    if depth == 0 and node == goal:
        return node  # Goal found
    if depth > 0:
        for neighbor in get_neighbors(node):
            result = depth_limited_search(neighbor, goal, depth - 1)
            if result is not None:
                return result
    return None
def iterative_deepening_search(start, goal, max_depth):
    for depth in range(max_depth):
        print(f"Searching at depth: {depth}")
        result = depth_limited_search(start, goal, depth)
        if result is not None:
            return result
    return None


def get_neighbors(node):
    neighbors = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F', 'G'],
        'D': [],
        'E': [],
        'F': [],
        'G': []
    }
    return neighbors.get(node, [])
start_node = 'A'  # Start node
    goal_node = 'G'   # Goal node
    max_depth = 3     # Maximum depth to search

    result = iterative_deepening_search(start_node, goal_node, max_depth)

    if result:
        print(f"Goal {goal_node} found!")
    else:
        print(f"Goal {goal_node} not found within depth limit.")
```

OUTPUT:

```
Searching at depth: 0
Searching at depth: 1
Searching at depth: 2
Goal G found!
```

15/10/2024 **Implement A* Search Algorithm for 8-Puzzle Problem**

ALGORITHM AND STATE SPACE TREE:

Date: 15/10/24 :-

Q. A* algorithm for 8- Puzzle problem:

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

Pseudocode & Algorithm:

To implement the A* algorithm for the 8-puzzle problem, we define two heuristics, $h(n)$:

1. No. of Misplaced Tiles:
2. Manhattan Distance

The A* algorithm will use $f(n) = g(n) + h(n)$, where, $g(n)$ is the depth of the node.

$h(n)$ is the heuristic, either misplaced tiles or Manhattan distance.

Algorithm:

1. Initialize:
   - → Start with the initial state of the puzzle.
   - → Use a priority queue to hold the nodes, ordered by $f(n) = g(n) + h(n)$.
   - → Keep a set of visited nodes to avoid re-exploring nodes.
2. Expand Nodes:
   - → While the priority queue is not empty.
   - → Remove the node with the lowest $f(n)$ from the queue.
   - → If this node is the goal, return the solution.
   - → Generate all possible child nodes (neighbouring states) by moving the blank tile.

   For each child:

   Calculate $g(n)$ (increment depth by 1 from parent)

   calculate $h(n)$ using the chosen heuristic.

   If the child has not been visited or if it has a better $f(n)$ than before, add it to the queue.

3. Termination:

→ If the goal is found, return the path and total cost.

→ If no solution is found, return failure.

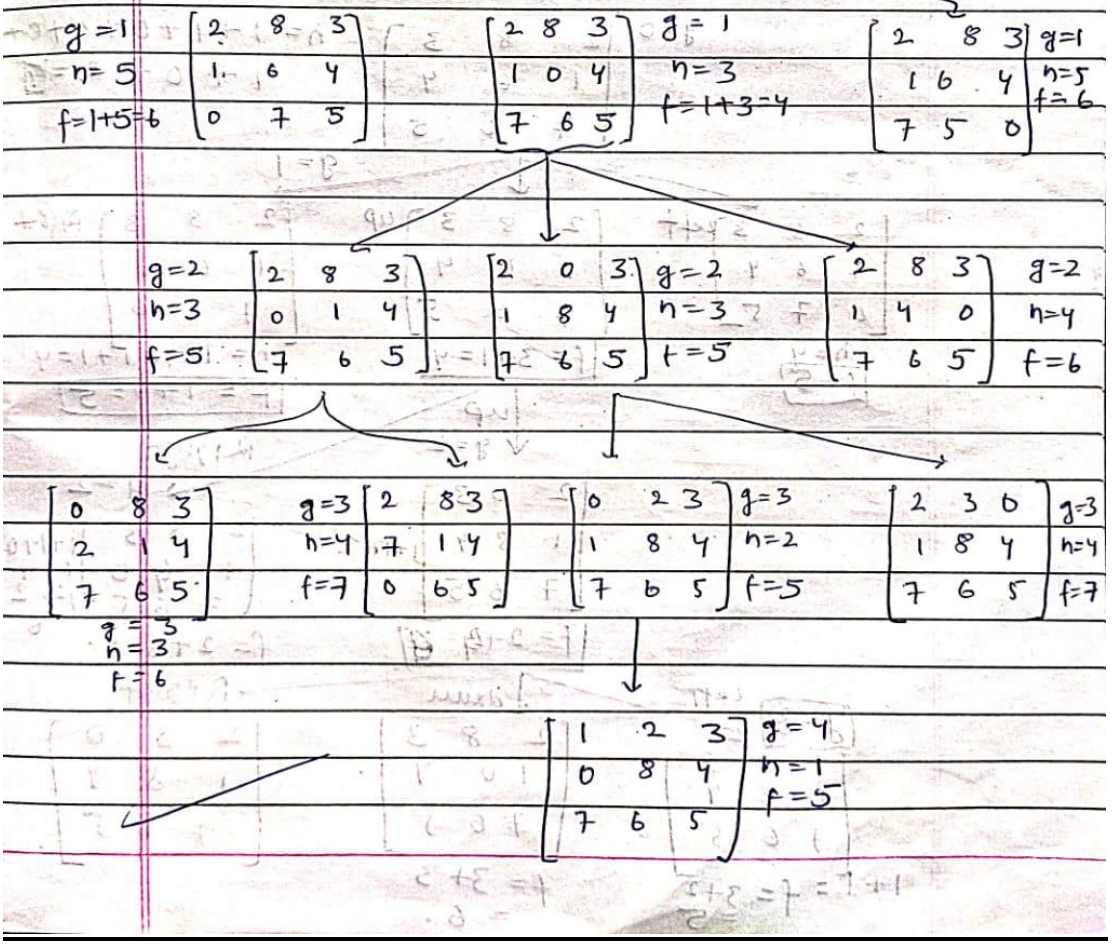
4. State space tree (Misplaced tiles):

Final state:

1	2	3
8	0	4
7	6	5

Initial state:

2	8	3
1	6	4
7	0	5



Mangal Date Page

$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ $g=4$, $h=1$, $f=5$

$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ $g=5$, $h=0$, $f=5$

$\begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 4 \\ 0 & 6 & 5 \end{bmatrix}$ $g=5$, $h=2$, $f=7$

Final state.

4. State space tree (Manhattan distance):

Initial state:

2 8 3
1 6 4
7 0 5

Final state:

1 2 3
8 0 4
7 6 5

$g=0$ $\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix}$ $h = 1+1+0+0+0+1+0+2+1 = 4$

$g=1$

$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 0 & 7 & 5 \end{bmatrix}$ left, $h=4$, $f=5$

$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ up, $f = 3+1 = 4$

$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 5 & 0 \end{bmatrix}$ Right, $h = 1+1+1+1 = 4$, $f = 1+4 = 5$

up, $g=2$

Right, $g=2$

$\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ $1+1+2$, $f = 2+2 = 4$

$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 4 & 0 \\ 7 & 6 & 5 \end{bmatrix}$ $1+1+1+1+2$, $f = 2+6 = 8$.

$g=3$, Left

$\begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ $1+1 = f = 3+2 = 5$

down

$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$ $f = 3+3 = 6$.

Right

$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$.

Mangal
Date
Page

$g = 4$

$$\begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

~~$h = 1+1+1 = 2$~~

~~$f = 4+2 = 6$~~ . . .

$g = 4$ | down

Right

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$h = 1$

$f = 4+1 = 5$

↓ Right

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$h = 0$.

final state.

$$\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$ X.

$f = 4+4 = 8$.

Namrata. M.
15/10/2024

CODE:

Manhattan Distance:

```
import heapq

# Define the goal state for the puzzle
goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

# Helper function to calculate the Manhattan distance heuristic
def manhattan_distance(state, goal_state):
    """Calculate the Manhattan distance heuristic."""
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                # Get current tile number (1-8)
                value = state[i][j]
                # Find its goal position in the goal state
                goal_x, goal_y = divmod(value, 3)
                # Calculate Manhattan distance
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == goal_state

def get_neighbors(state):
    """Generate neighbors for a given state."""
    neighbors = []
    # Find the position of the empty space (0)
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i == 0][0]
    # Possible moves: up, down, left, right
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

    for move_x, move_y in moves:
        if 0 <= move_x < 3 and 0 <= move_y < 3:
            # Create a copy of the current state and swap the empty space
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[move_x][move_y] = new_state[move_x][move_y],
new_state[x][y]
            neighbors.append(new_state)
    return neighbors

# A* algorithm using Manhattan distance heuristic
```

```
def a_star_manhattan(initial_state):
    """A* algorithm implementation for 8-puzzle problem with Manhattan distance heuristic."""
    open_list = []
    closed_set = set()

    # Initial cost
    g = 0
    f = g + manhattan_distance(initial_state, goal_state)

    # Add the initial state to the open list
    heapq.heappush(open_list, (f, g, initial_state, []))

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if is_goal(current_state):
            return path + [current_state]

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

        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) in closed_set:
                continue

            new_g = g + 1
            new_f = new_g + manhattan_distance(neighbor, goal_state)
            heapq.heappush(open_list, (new_f, new_g, neighbor, path + [current_state]))

    return None

# Helper function to print a state
def print_state(state):
    for row in state:
        print(row)
    print()

# Initial state
initial_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]

# Run the A* algorithm with Manhattan distance heuristic
print("Solution using Manhattan distance heuristic:")
solution_manhattan = a_star_manhattan(initial_state)
if solution_manhattan:
    for step in solution_manhattan:
        print_state(step)
else:
```

print("No solution found using Manhattan distance heuristic.")

OUTPUT:

```
Solution using Manhattan distance heuristic:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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


=== Code Execution Successful ===
```

Misplaced Tiles:

```
import heapq

# Define the goal state for the puzzle
goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

# Helper function to calculate the misplaced tiles heuristic
def misplaced_tiles(state, goal_state):
    """Calculate the number of misplaced tiles."""
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                count += 1
    return count

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == goal_state

def get_neighbors(state):
    """Generate neighbors for a given state."""
    neighbors = []
    # Find the position of the empty space (0)
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i == 0][0]
    # Possible moves: up, down, left, right
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

    for move_x, move_y in moves:
        if 0 <= move_x < 3 and 0 <= move_y < 3:
            # Create a copy of the current state and swap the empty space
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[move_x][move_y] = new_state[move_x][move_y],
new_state[x][y]
            neighbors.append(new_state)
    return neighbors

# A* algorithm using misplaced tiles heuristic
def a_star_misplaced(initial_state):
    """A* algorithm implementation for 8-puzzle problem with misplaced tiles heuristic."""
    open_list = []
    closed_set = set()

    # Initial cost
    g = 0
```

```
    f = g + misplaced_tiles(initial_state, goal_state)

    # Add the initial state to the open list
    heapq.heappush(open_list, (f, g, initial_state, []))

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if is_goal(current_state):
            return path + [current_state]

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

        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) in closed_set:
                continue

            new_g = g + 1
            new_f = new_g + misplaced_tiles(neighbor, goal_state)
            heapq.heappush(open_list, (new_f, new_g, neighbor, path + [current_state]))

    return None

# Helper function to print a state
def print_state(state):
    for row in state:
        print(row)
    print()

# Initial state
initial_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]

# Run the A* algorithm with misplaced tiles heuristic
print("Solution using misplaced tiles heuristic:")
solution_misplaced = a_star_misplaced(initial_state)
if solution_misplaced:
    for step in solution_misplaced:
        print_state(step)
else:
    print("No solution found using misplaced tiles heuristic.")
```

OUTPUT:

```
Solution using misplaced tiles heuristic:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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


=== Code Execution Successful ===
```

22/10/2024 **Implement Hill Climbing search algorithm to solve N-Queens problem**

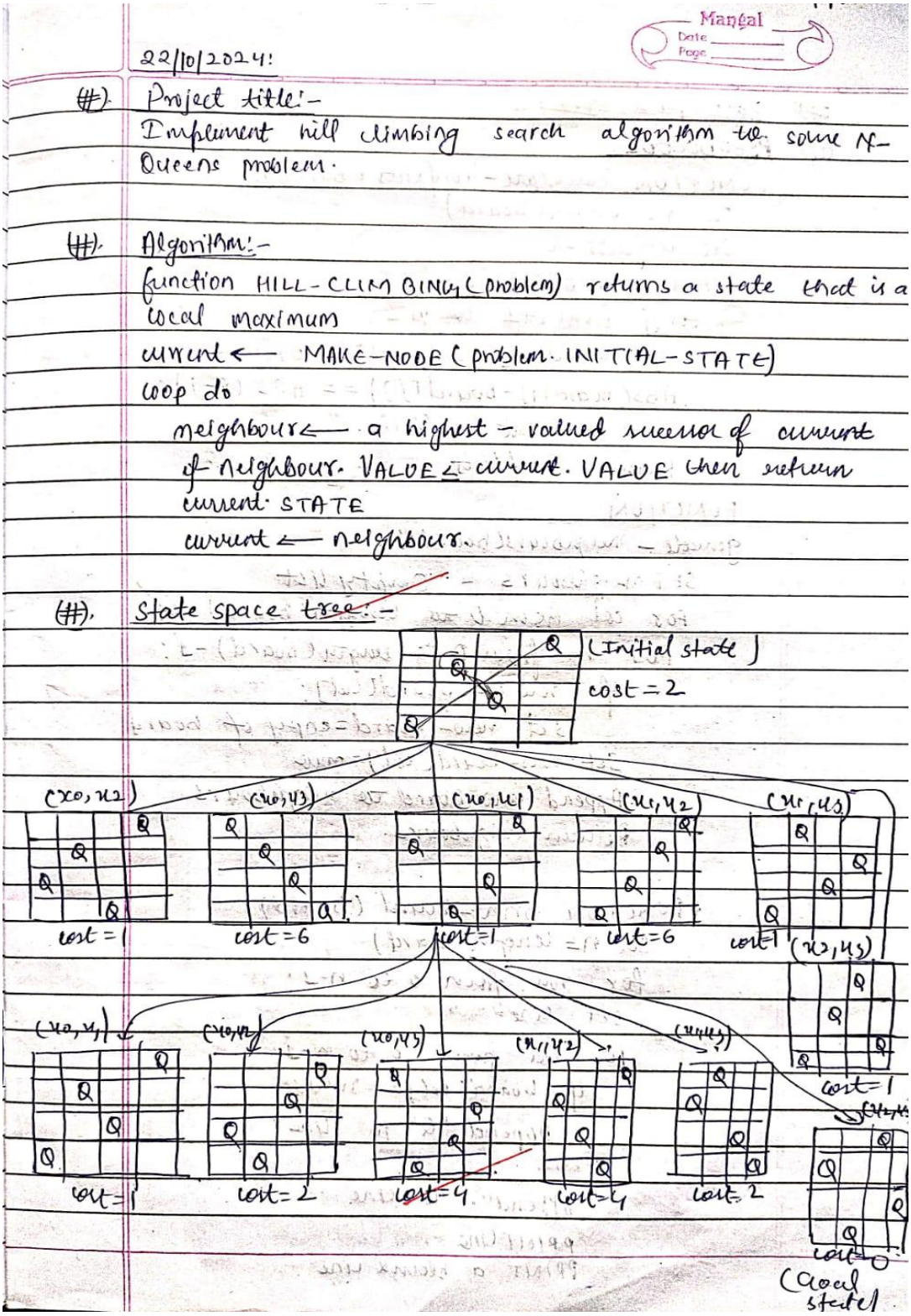
ALGORITHM AND STATE SPACE TREE:

22/10/2024:

(#). Project title:-
Implement hill climbing search algorithm to solve N-Queens problem.

(##). Algorithm:-
function HILL-CLIMBING(problem) returns a state that is a local maximum
current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
neighbour ← a highest-valued successor of current
if neighbour.VALUE ≤ current.VALUE then return current.STATE
current ← neighbour.

(###). State space tree:-



Mangal
Date
Page

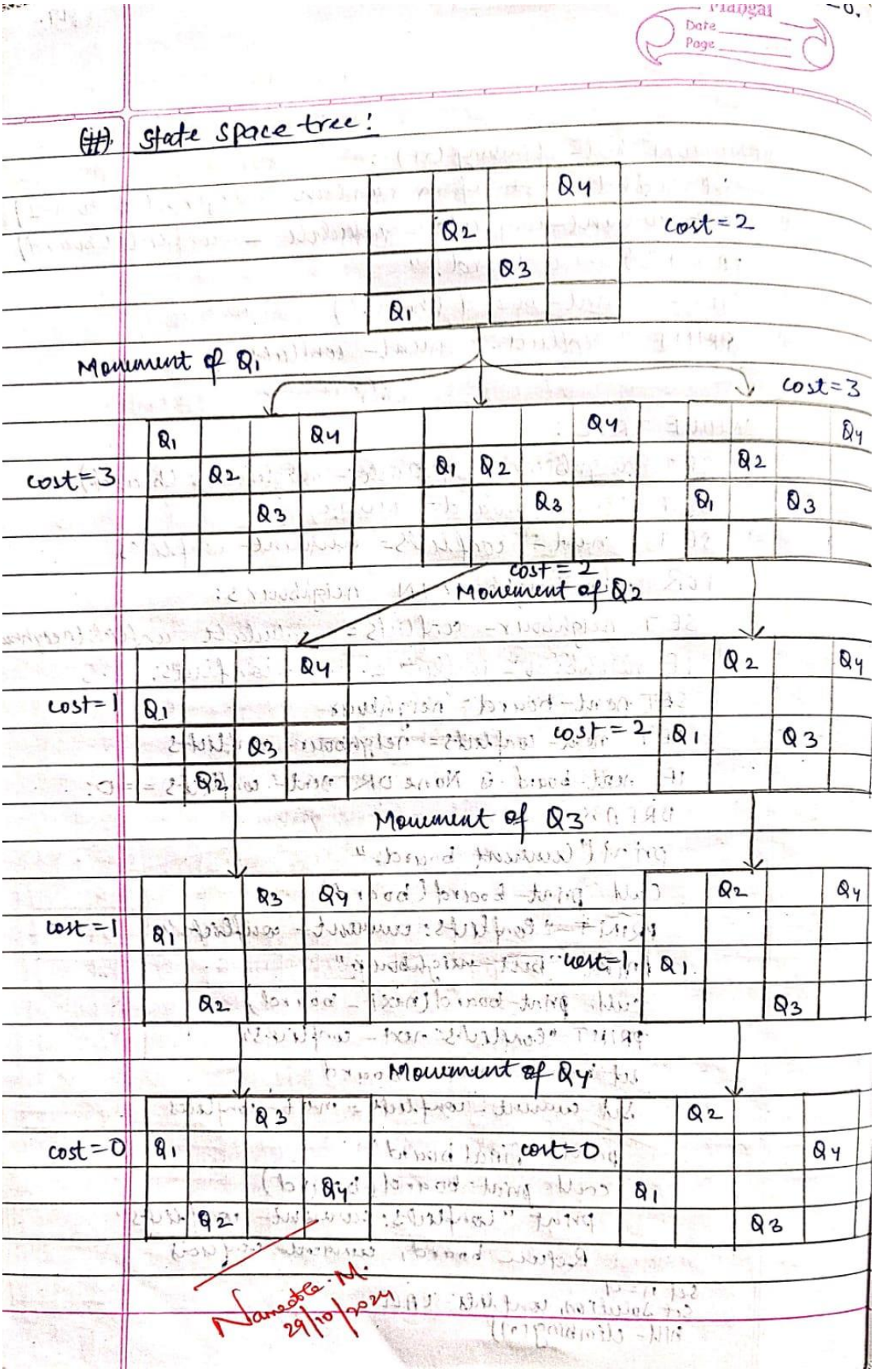
~~State space tree:~~

PSEUDOCODE:

```
FUNCTION calculate - conflicts (board):
  Set n = length(board)
  Set conflicts = 0
  For i From 0 to n-1:
    For j from i+1 to n-1:
      if board[i] == board[j] OR
      ABS(board[i] - board[j]) == ABS (i-j):
        increment conflicts.
    Return conflicts
  FUNCTION
  generate - neighbours(board):
    SET neighbours = Empty list
    For col from 0 to length(board) - 1:
    For row from 0 to length(board) - 1:
      if row != board[col]:
          Set new- board = copy of board
        Set new- board[col] = row
        Append new-board to neighbours
        Return neighbours.

    FUNCTION print-board (board):
      Set n = length(board)
      for row from 0 to n-1:
        set line = " "
        For col from 0 to n-1:
          if board[col] == row:
            Append "Q" to line
          else
            Append "." to line
          PRINT line
          PRINT a blank line
```

```
FUNCTION hill-climbing(n):
    Set board = list of n random integers (0 to n-1)
    set current-conflicts = calculate - conflicts(board)
    PRINT "Initial board:"
    CALL print-board(board)
    PRINT "conflicts: current-conflicts".

WHILE TRUE:
    SET neighbours = generate-neighbours(board)
    SET next-board = None
    SET next-conflicts = current-conflicts
    FOR each neighbour IN neighbours:
        SET neighbour-conflicts = calculate-conflicts(neighbour
        IF neighbour-conflicts < next-conflicts:
            SET next-board = neighbour
            SET next-conflicts = neighbour-conflicts
    IF next-board is None OR next-conflicts == 0:
        BREAK
    print("Current board:"
    Call print-board(board)
    PRINT "conflicts: current-conflicts"
    Print "Best neighbour:"
    call print-board(next-board)
    PRINT "conflicts: next-conflicts"
    set board = next-board
    set current-conflicts = next-conflicts
    print "final board"
    call print-board(board)
    print "conflicts: current-conflicts"
    Return board, current-conflicts

set n=4
set solution, conflicts = CALL
hill-climbing(n)
```

(iii) State space tree:



Namrata. M.
29/10/2024

CODE:

```
import random

def heuristic(state):
    """Calculate the number of conflicts between queens."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def generate_neighbors(state):
    """Generate all neighboring states by swapping two queens."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            # Swap queens at positions i and j
            new_state = state.copy()
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(new_state)
    return neighbors

def print_board(state):
    """Print the board configuration."""
    n = len(state)
    board = [["." for _ in range(n)] for _ in range(n)]
    for row in range(n):
        board[row][state[row]] = "Q"
    for line in board:
        print(" ".join(line))
    print()

def get_user_input(n):
    """Get initial state from user input."""
    while True:
        try:
            input_state = input(f"Enter the initial positions for {n} queens (0 to {n-1}, space-
separated): ")
            positions = list(map(int, input_state.split()))
            if len(positions) != n or any(p < 0 or p >= n for p in positions):
                raise ValueError
            return positions
```

```
        except ValueError:
            print("Invalid input. Please enter exactly {} numbers between 0 and {}.".format(n, n - 1))

def hill_climbing(n):
    """Perform the Hill-Climbing algorithm using the swapping technique."""
    # Get initial state from user
    current_state = get_user_input(n)
    current_cost = heuristic(current_state)

    print("Initial State:")
    print_board(current_state)
    print(f"Initial Heuristic (Conflicts): {current_cost}\n")

    while current_cost > 0:
        neighbors = generate_neighbors(current_state)
        next_state = None
        next_cost = current_cost  # Initialize with the current cost

        for neighbor in neighbors:
            cost = heuristic(neighbor)
            print(f"Evaluating Neighbor:")
            print_board(neighbor)
            print(f"Heuristic (Conflicts): {cost}")

            # Update the next state if a better (lower cost) neighbor is found
            if cost < next_cost:
                next_cost = cost
                next_state = neighbor

        if next_state is None:  # Local maximum reached
            print("Local maximum reached. No better neighbors found.")
            break  # Exit the loop

        # Move to the best neighbor found
        print("Moving to Next State:")
        current_state = next_state
        current_cost = next_cost
        print_board(current_state)
        print(f"Heuristic (Conflicts): {current_cost}\n")

    return current_state, current_cost  # Return the best found state and its cost

# Run the algorithm for the 4-queens problem
n = 4  # Change this value for different sizes of the board
solution, solution_cost = hill_climbing(n)
print("Best Solution Found:")
```

```
print_board(solution)
print(f"Final Heuristic (Conflicts): {solution_cost}")
```

OUTPUT:

```
Enter the initial positions for 4 queens (0 to 3, space-separated): 3 1 2 0
Initial State:
. . . Q
. Q . .
. . Q .
Q . . .

Initial Heuristic (Conflicts): 2

Evaluating Neighbor:
. Q . .
. . . Q
. . Q .
Q . . .

Heuristic (Conflicts): 1
Evaluating Neighbor:
. . Q .
. Q . .
. . . Q
Q . . .

Heuristic (Conflicts): 1
Evaluating Neighbor:
Q . . .
. Q . .
. . Q .
. . . Q

Heuristic (Conflicts): 6
```

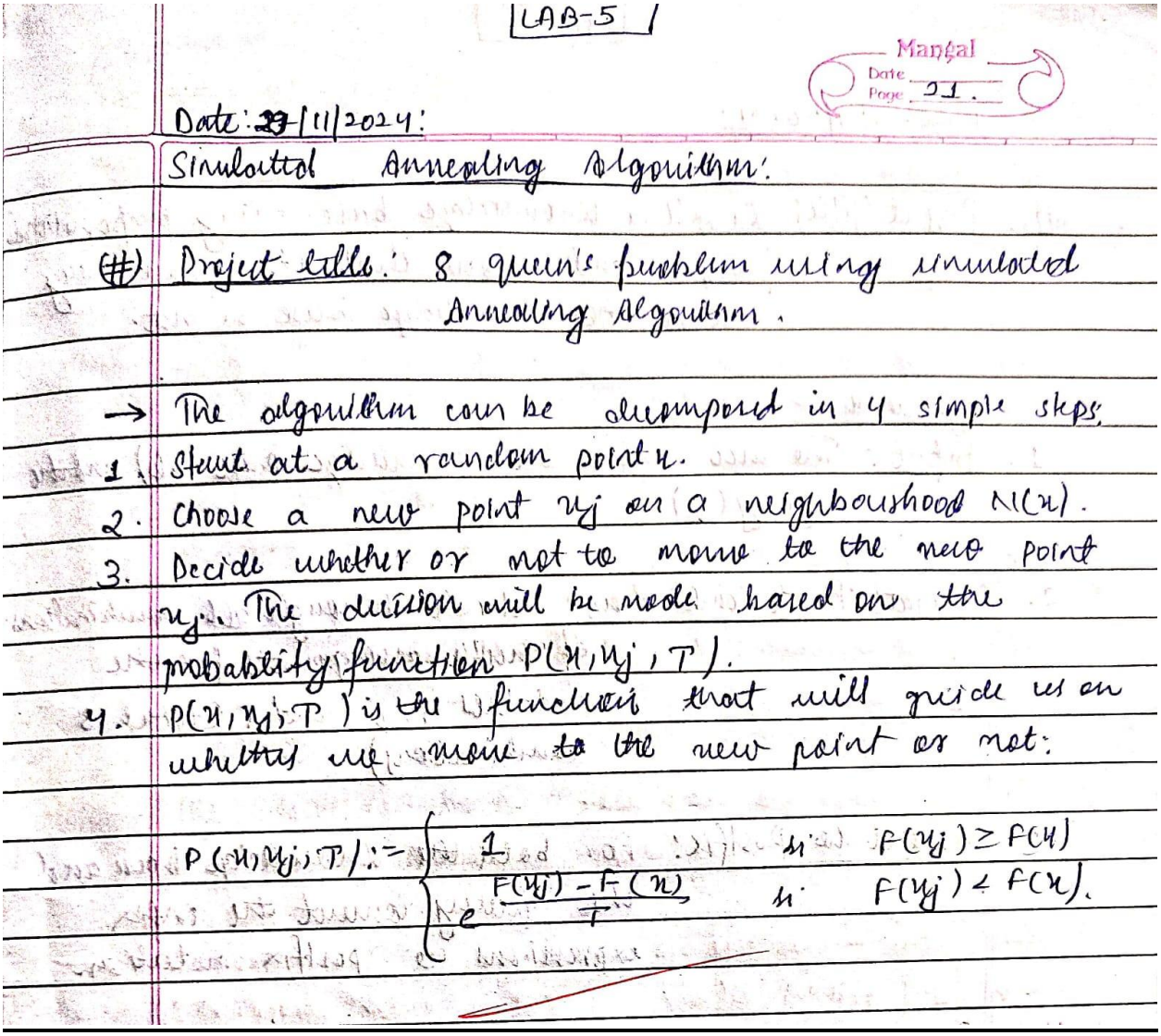
```
 Evaluating Neighbor:
 . . . Q
 . . Q .
 . Q . .
 Q . . .

 Heuristic (Conflicts): 6
 Evaluating Neighbor:
 . . . Q
 Q . . .
 . . Q .
 . Q . .

 Heuristic (Conflicts): 1
 Evaluating Neighbor:
 . . . Q
 . Q . .
 Q . . .
 . . Q .

 Heuristic (Conflicts): 1
 Moving to Next State:
 . Q . .
 . . . Q
 . . Q .
 Q . . .

 Heuristic (Conflicts): 1
```

```
Evaluating Neighbor:
. . . Q
. Q . .
. . Q .
Q . . .

Heuristic (Conflicts): 2
Evaluating Neighbor:
. . Q .
. . . Q
. Q . .
Q . . .

Heuristic (Conflicts): 2
Evaluating Neighbor:
Q . . .
. . . Q
. . Q .
. Q . .

Heuristic (Conflicts): 4
Evaluating Neighbor:
. Q . .
. . Q .
. . . Q
Q . . .

Heuristic (Conflicts): 4
```

```
Evaluating Neighbor:
. Q . .
Q . . .
. . Q .
. . . Q

Heuristic (Conflicts): 2
Evaluating Neighbor:
. Q . .
. . . Q
Q . . .
. . Q .

Heuristic (Conflicts): 0
Moving to Next State:
. Q . .
. . . Q
Q . . .
. . Q .

Heuristic (Conflicts): 0

Best Solution Found:
. Q . .
. . . Q
Q . . .
. . Q .

Final Heuristic (Conflicts): 0

=== Code Execution Successful ===
```

20/10/2024 **Simulated Annealing Algorithm**

ALGORITHM AND STATE SPACE TREE:



LAB-5

Mantal
Date
Page 31.

Date: 29/11/2024:

Simulated Annealing Algorithm:

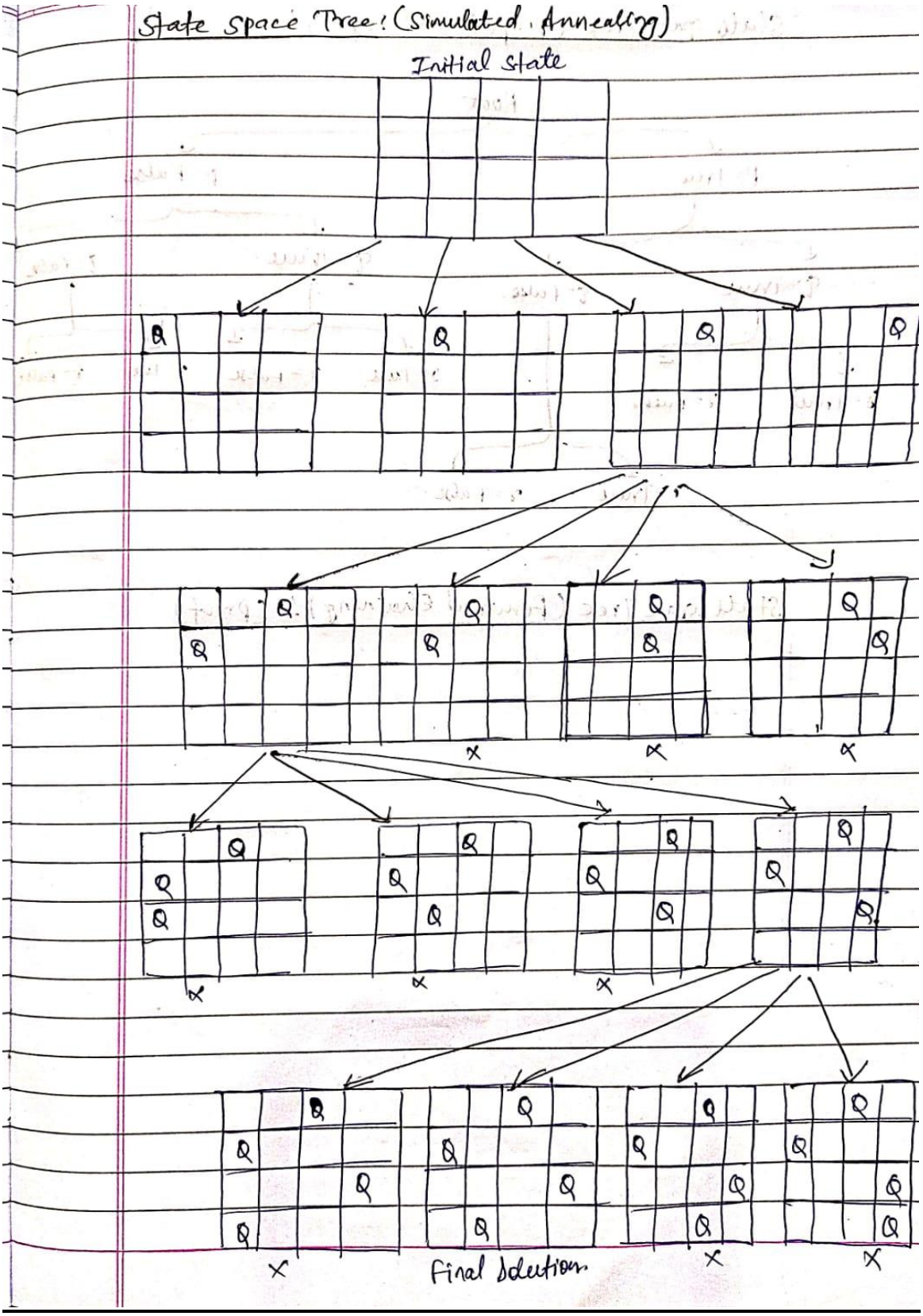
(#) Project title: 8 queen's problem using simulated Annealing Algorithm.

→ The algorithm can be decomposed in 4 simple steps:

1. Start at a random point x.
2. Choose a new point x_j on a neighbourhood $N(x)$.
3. Decide whether or not to move to the new point x_j. The decision will be made based on the probability function $P(x, x_j, T)$.
4. $P(x, x_j, T)$ is the function that will guide us on whether we move to the new point or not:

$$P(x, x_j, T) := \begin{cases} 1 & \text{if } F(x_j) \geq F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & \text{if } F(x_j) < F(x). \end{cases}$$

State space Tree: (Simulated Annealing)



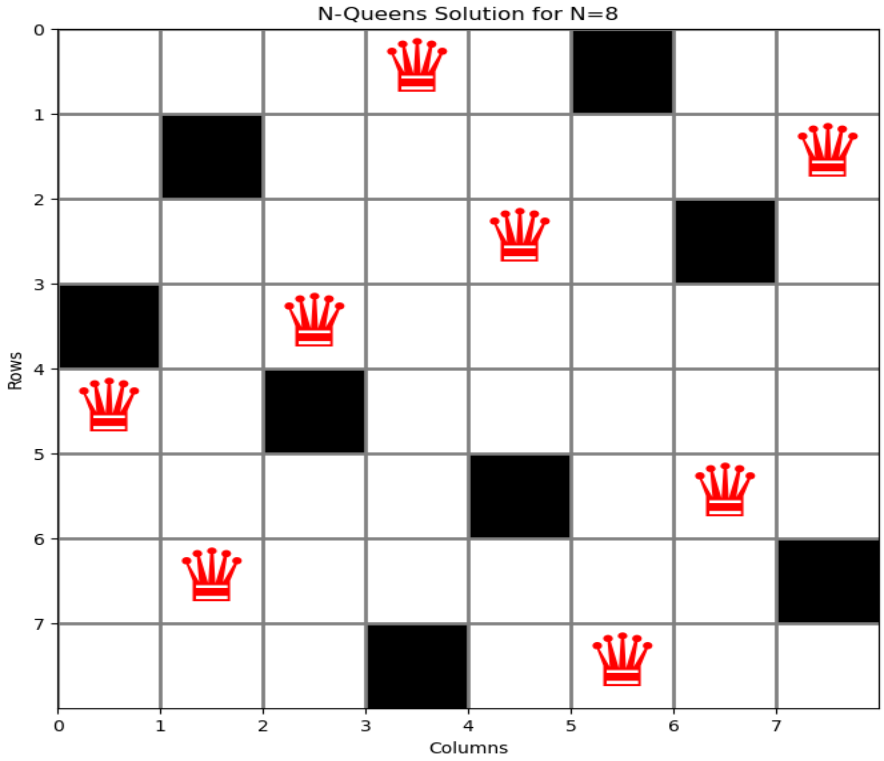
CODE:

```
import mlrose_hiive as mlrose
import numpy as np
import matplotlib.pyplot as plt

# Define the objective function
def queens_max(position):
    n = len(position)
    attacking_pairs = 0
    for i in range(n):
        for j in range(i + 1, n):
            if (position[i] == position[j] or
                abs(position[i] - position[j]) == abs(i - j)):
                attacking_pairs += 1
    # Total pairs - attacking pairs gives non-attacking pairs
    return (n * (n - 1)) // 2 - attacking_pairs

# Assign the objective function to "CustomFitness" method
objective = mlrose.CustomFitness(queens_max)

# Description of the problem
problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)

# Define decay schedule
T = mlrose.ExpDecay()

# Define initial state
initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

# Solve problem using simulated annealing
best_result = mlrose.simulated_annealing(
    problem=problem,
    schedule=T,
    max_attempts=500,
    max_iters=5000,
    init_state=initial_position
)

# Extract best state and best fitness from the result
best_state = best_result[0]  # The best state (positions of queens)
best_fitness = best_result[1]  # The number of non-attacking queens

# Print results
print('The best position found is: ', best_state)
print('The number non attacking pair of queens: ', best_fitness)
```

```
# Function to visualize the N-Queens solution
def plot_n_queens(positions):
    n = len(positions)
    board = np.zeros((n, n))

    # Place queens on the board
    for col, row in enumerate(positions):
        board[row, col] = 1  # 1 represents a queen

    plt.figure(figsize=(8, 8))
    plt.imshow(board, cmap='binary', extent=[0, n, 0, n])

    # Add grid lines
    plt.xticks(np.arange(0, n, 1))
    plt.yticks(np.arange(0, n, 1))
    plt.grid(color='gray', linestyle='-', linewidth=2)

    # Add queens
    for col, row in enumerate(positions):
        plt.text(col + 0.5, row + 0.5, '♛', fontsize=48, ha='center', va='center', color='red')

    plt.title(f'N-Queens Solution for N={n}')
    plt.xlabel('Columns')
    plt.ylabel('Rows')
    plt.gca().invert_yaxis()  # Invert y-axis to match chessboard orientation
    plt.show()

# Visualize the solution
plot_n_queens(best_state)
```

OUTPUT:

```
The best position found is:  [4 6 3 0 2 7 5 1]
The number non attacking pair of queens:  28.0
```



12/11/2024 **Knowledge Base Using Prepositional Logic**

ALGORITHM AND STATE SPACE TREE:

Date: 12/11/2024:

(##). Project Title: Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

Algorithm:

1. Input: The user enters the knowledge base (KB) and the query (q).
2. Generate Truth combinations: Iterate through all combinations of truth assignments for the variables P, q and r (total 8 combinations).
3. Convert to Postfix: For both the knowledge base and the query, convert the infix expressions to postfix notation
4. Evaluate postfix Expressions: For each combination of truth values, evaluate both the knowledge base and the query.
5. Check Entailment: If there is any combination where knowledge base evaluates to True and the query evaluates to False, return False (indicating that the knowledge base does not entail the query). If no such combination is found, return True.
6. Output: Print whether the knowledge base entails the query.

Mangal
Date
Page 23.

Output:

enter rule: p^ q^ r

enter the query: p

Truth table Reference.

KB	alpha
True	True
False	True
False	True
False	True
False	False
False	False
False	False
False	False

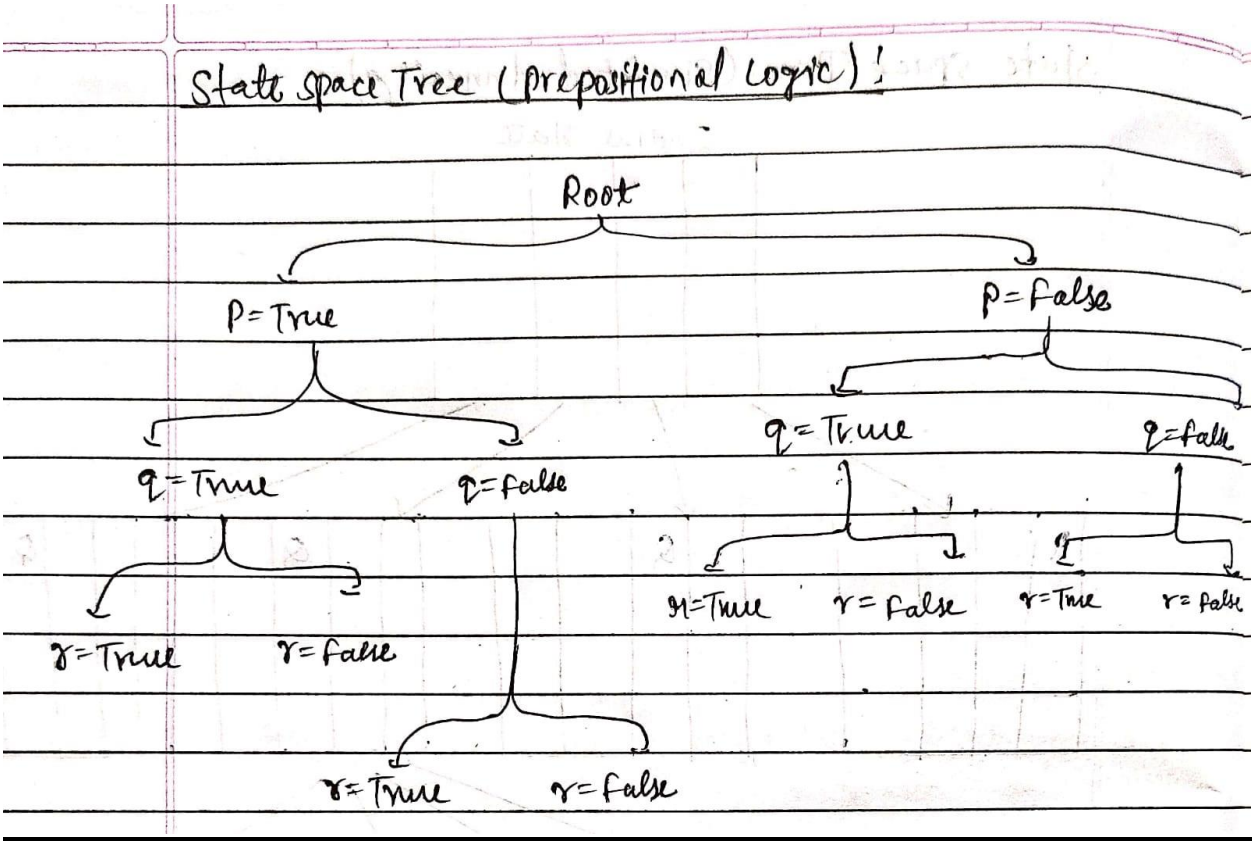
#Result: The knowledge base entails query.

Truth space Table

combination(P,q,r)	KB evaluation	Query evaluation	entailment check
(T,T,T)	True	True	True
(T,T,F)	False	True	True
(T,F,T)	False	True	True
(T,F,F)	False	True	True
(F,T,T)	False	False	True
(F,T,F)	False	False	True
(F,F,T)	False	False	True
(F,F,F)	False	False	True.

Namrata M
19/11/2024

(#) Result: The knowledge base does not entail query state not found KB! True and Query: False in state 2.



CODE:

```
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)
]
variable = {'p': 0, 'q': 1, 'r': 2}
kb = ''
q = ''
priority = {'~': 3, 'v': 1, '^': 2}

def input_rules():
    global kb, q
    kb = input("Enter rule: ")
    q = input("Enter the Query: ")
```

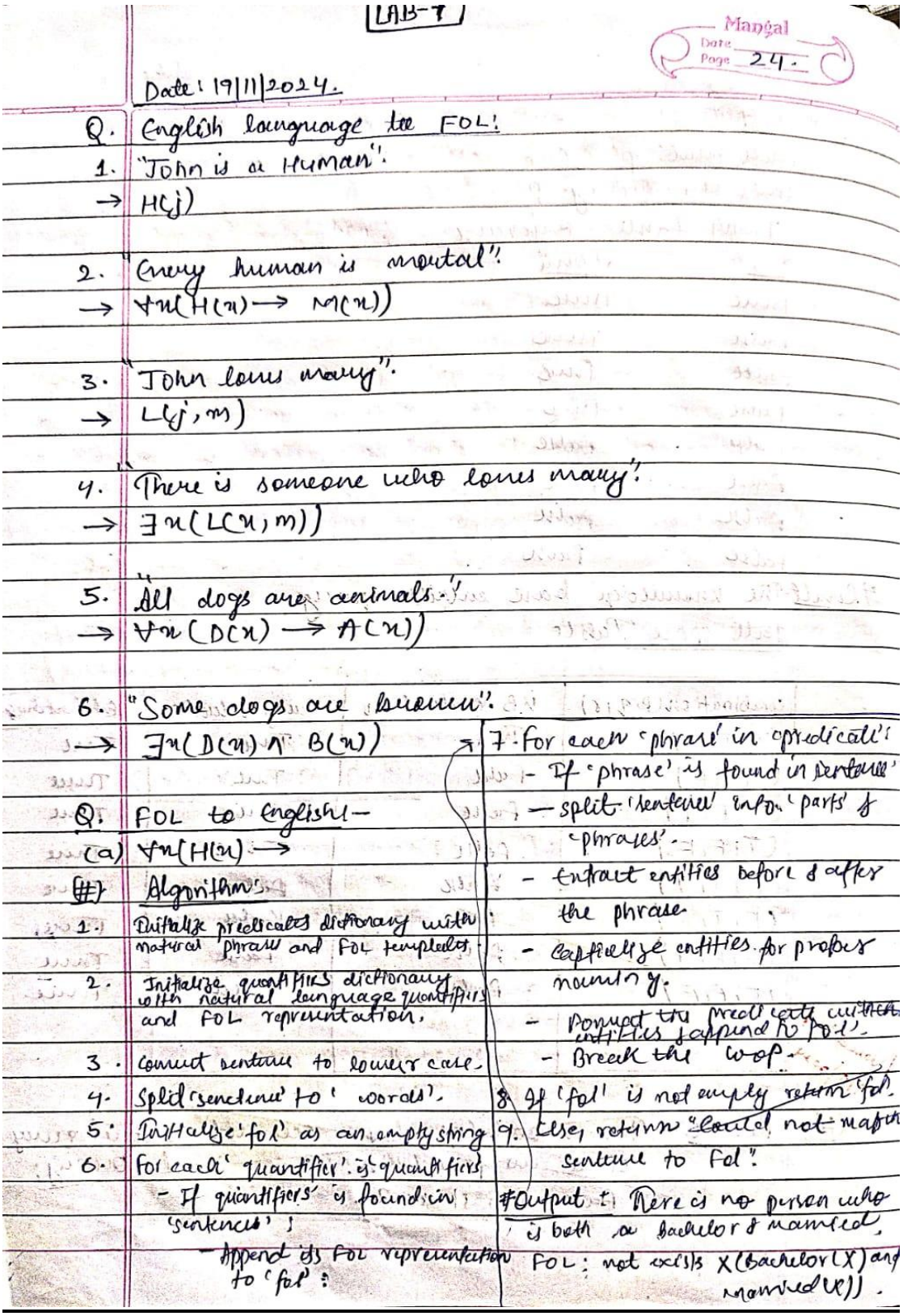
```
def entailment():
    global kb, q
    print('*' * 10 + " Truth Table Reference " + '*' * 10)
    print('kb', 'alpha')
    print('*' * 10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-' * 10)
        if s and not f:
            return False
    return True

def isOperand(c):
    return c.isalpha() and c != 'v'

def isLeftParenthesis(c):
    return c == '('

def isRightParenthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0

def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False

def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParenthesis(c):
                stack.append(c)
            elif isRightParenthesis(c):
```

```
            operator = stack.pop()
            while not isLeftParenthesis(operator):
                postfix += operator
                operator = stack.pop()
        else:
            while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                postfix += stack.pop()
            stack.append(c)
    while not isEmpty(stack):
        postfix += stack.pop()
    return postfix

def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i, val2, val1))
    return stack.pop()

def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1

# Main execution
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails the query.")
else:
    print("The Knowledge Base does not entail the query.")
```

OUTPUT:

```
********** Truth Table Reference **********
kb alpha
**********
True True
----------
True True
----------
True False
----------
The Knowledge Base does not entail the query.
```

19/11/2024 **FOL Conversions**



LAB-7

Date: 19/11/2024.

Q. English language to FOL:

1. "John is a Human".
→ H(j)

2. "Every human is mortal".
→ ∀x(H(x) → M(x))

3. "John loves mary".
→ L(j, m)

4. "There is someone who loves mary".
→ ∃x(L(x, m))

5. "All dogs are animals".
→ ∀x(D(x) → A(x))

6. "Some dogs are brown".
→ ∃x(D(x) ∧ B(x))

Q. FOL to English:-

(a) ∀x(H(x) →

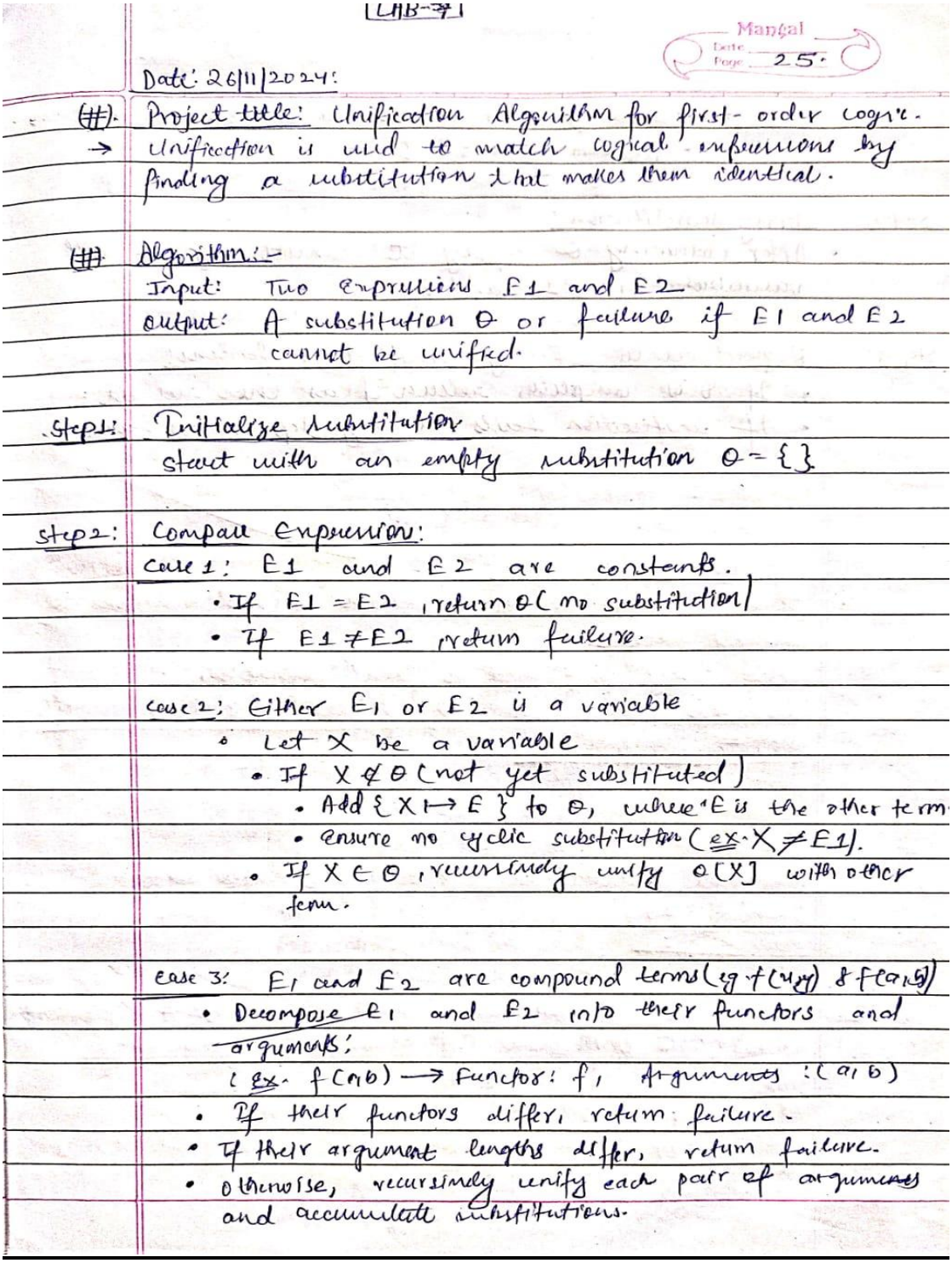
(#) Algorithm:

1. Initialize predicates dictionary with natural phrase and FOL templates.
2. Initialize quantifiers dictionary with natural language quantifiers and FOL representation.
3. Convert sentence to lower case.
4. Split 'sentence' to 'words'.
5. Initialize 'fol' as an empty string
6. For each 'quantifier' in quantifiers
   - If quantifiers' is found in 'sentence':
     - Append its FOL representation to 'fol':
7. For each 'phrase' in predicate':
   - If 'phrase' is found in sentence'
   - split 'sentence' into 'parts' of 'phrases'.
   - Extract entities before & after the phrase.
   - Capitalize entities for proper nouns.
   - Replace the predicate with entities mapped to 'FOL'.
   - Break the loop.
8. If 'fol' is not empty return 'fol'.
9. Else, return "Could not map sentence to fol".

#Output: There is no person who is both a bachelor & married.
FOL: not exists X(bachelor(X) and married(X)).

26/11/2024 **Unification In First Order Logic**

ALGORITHM:



[LAB-7]

Mangal
Date
Page 25

Date: 26/11/2024:

(##) Project title: Unification Algorithm for first-order logic.

→ Unification is used to match logical expressions by finding a substitution that makes them identical.

(##) Algorithm:-

Input: Two expressions E1 and E2

Output: A substitution θ or failure if E1 and E2 cannot be unified.

Step1: Initialize substitution

start with an empty substitution θ = {}

Step2: Compare expressions:

Case 1: E1 and E2 are constants.

- If E1 = E2, return θ (no substitution)
- If E1 ≠ E2, return failure.

Case 2: Either E1 or E2 is a variable

- Let X be a variable
  - If X ∉ θ (not yet substituted)
    - Add {X ↦ E} to θ, where E is the other term
    - Ensure no cyclic substitution (ex. X ≠ E1).
  - If X ∈ θ, recursively unify θ[X] with other term.

Case 3: E1 and E2 are compound terms (eg f(x,y) & f(a,b))

- Decompose E1 and E2 into their functors and arguments:
  (ex. f(a,b) → Functor: f, Arguments: (a, b))
- If their functors differ, return failure.
- If their argument lengths differ, return failure.
- Otherwise, recursively unify each pair of arguments and accumulate substitutions.

Mangal
Date
Page 26

Case 4: E1 or E2 is neither a variable nor a term.
- Return failure.

Step3: Apply Substitutions
- After obtaining θ, apply it consistently to all variables in E1 and E2.

Step4: Repeat until Fully unified or failure
- If θ is complete, return it as the result.
- If unification fails at any step, return failure.

CODE:

```
//Supported Lines
//"John is a human."
//"Every human is mortal."
//"John loves Mary."
//"All dogs are animals."
//"Some dogs are brown."
//"There is someone who loves Mary."
//"Mary is the mother of John."
//"If it is raining, then the ground is wet."
//"John and Mary are both students."
//"No one is both a teacher and a student."
//"Nobody is taller than themselves."
```

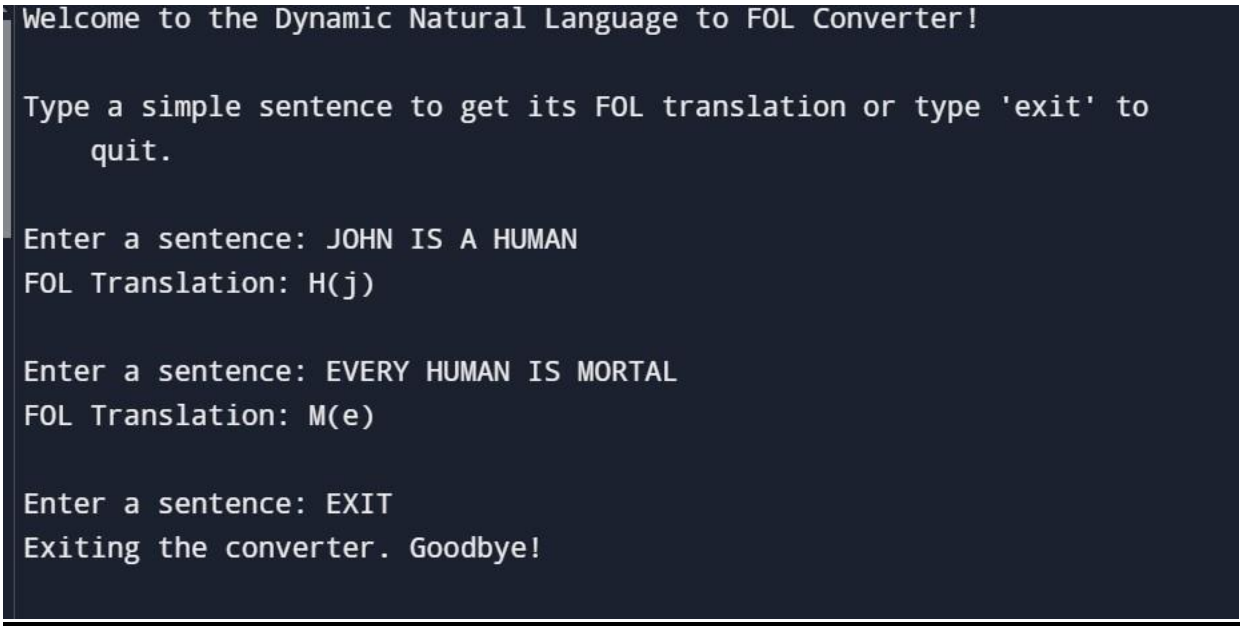
CODE

```
# Function to dynamically transform sentences into FOL
def sentence_to_fol(sentence):
    sentence = sentence.lower().strip()

    if "is a human" in sentence:
        subject = sentence.split(" is a human")[0]
        return f"H({subject[0]})"  # Use the first letter of the subject as a variable

    elif "is mortal" in sentence:
        subject = sentence.split(" is mortal")[0]
        return f"M({subject[0]})"

    elif "loves" in sentence:
        parts = sentence.split(" loves ")
        subject = parts[0]
        object_ = parts[1].replace(".", "")
        return f"L({subject[0]}, {object_[0]})"

    elif "all" in sentence and "are" in sentence:
        parts = sentence.replace("all ", "").split(" are ")
        subject_class = parts[0]
        object_class = parts[1].replace(".", "")
        return f"∀x({subject_class[0].upper()}(x) → {object_class[0].upper()}(x))"

    elif "some" in sentence and "are" in sentence:
        parts = sentence.replace("some ", "").split(" are ")
        subject_class = parts[0]
        object_class = parts[1].replace(".", "")
```

```
        return f"∃x({subject_class[0].upper()}(x) ∧ {object_class[0].upper()}(x))"

    elif "there is someone who" in sentence:
        action = sentence.replace("there is someone who ", "").strip().replace(".", "")
        if "loves" in action:
            parts = action.split(" loves ")
            object_ = parts[1]
            return f"∃x(L(x, {object_[0]}))"

    elif "is the mother of" in sentence:
        parts = sentence.split(" is the mother of ")
        subject = parts[0]
        object_ = parts[1].replace(".", "")
        return f"M({subject[0]}, {object_[0]})"

    elif "if" in sentence and "then" in sentence:
        parts = sentence.split(", then ")
        condition = parts[0].replace("if ", "").strip()
        conclusion = parts[1].strip().replace(".", "")
        return f"({condition[0].upper()} → {conclusion[0].upper()})"

    elif "and" in sentence:
        parts = sentence.split(" and ")
        subjects = [p.strip().replace(".", "") for p in parts]
        return f"({' ∧ '.join([s[0].upper() for s in subjects])})"

    elif "no one is both" in sentence:
        qualities = sentence.replace("no one is both ", "").strip().replace(".", "").split(" and ")
        return f"¬∃x({qualities[0][0].upper()}(x) ∧ {qualities[1][0].upper()}(x))"

    elif "nobody is taller than themselves" in sentence:
        return "¬∃x(T(x, x))"

    else:
        return "Translation rule not found for this sentence."

# Main program
def main():
    print("Welcome to the Dynamic Natural Language to FOL Converter!")
    print("\nType a simple sentence to get its FOL translation or type 'exit' to quit.")

    while True:
        user_input = input("\nEnter a sentence: ").strip()
        if user_input.lower() == 'exit':
            print("Exiting the converter. Goodbye!")
            break
```

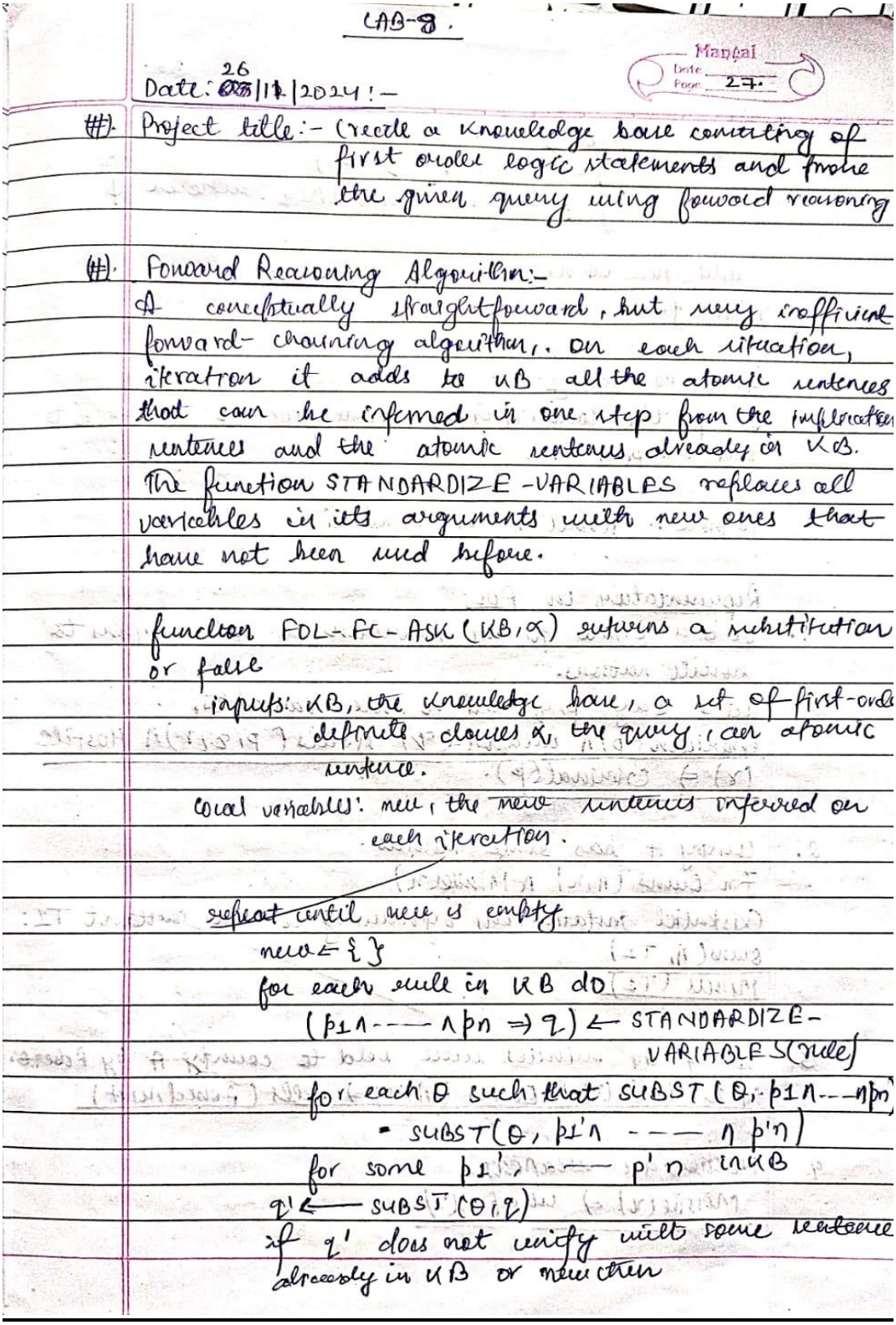
```
        translation = sentence_to_fol(user_input)
        print(f"FOL Translation: {translation}")

# Run the program
if __name__ == "__main__":
    main()
```

OUTPUT:



26/11/2024 **FORWARD REASONING**

ALGORITHM AND STATE SPACE TREE:



LAB-8.

Date: 26/11/2024 :-

Manjal Date Page 27.

##. Project title :- Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

##. Forward Reasoning Algorithm :-

A conceptually straightforward, but very efficient forward-chaining algorithm. On each iteration, iteration it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ---- ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, p1 ∧ --- ∧ pn) = SUBST(θ, p1' ∧ --- ∧ p'n)
for some p1', ---- p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then

add q' to new
$\phi \leftarrow$ UNIFY (q', α)
if ϕ is not fail then return ϕ

add new to KB
return false

Given Case study:
As per the lower, it is a crime for an American to sell weapons - - -.

To prove: "Robert is a criminal".

Representation in FOL:

1. It is a crime for an American to sell weapons to hostile nations.
Let's say p, q and r are variables.
American(p) ∧ weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p).

2. Country A has some missiles.
∃x Owns(A, x) ∧ Missile(x)
Existential Instantiation, introducing a new constant T1:
Owns(A, T1)
Missile(T1)

3. All of the missiles were sold to country A by Robert.
∀x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

4. Missiles are weapons.
Missile(x) ⇒ Weapon(x)

Mangal
Date
Page 29

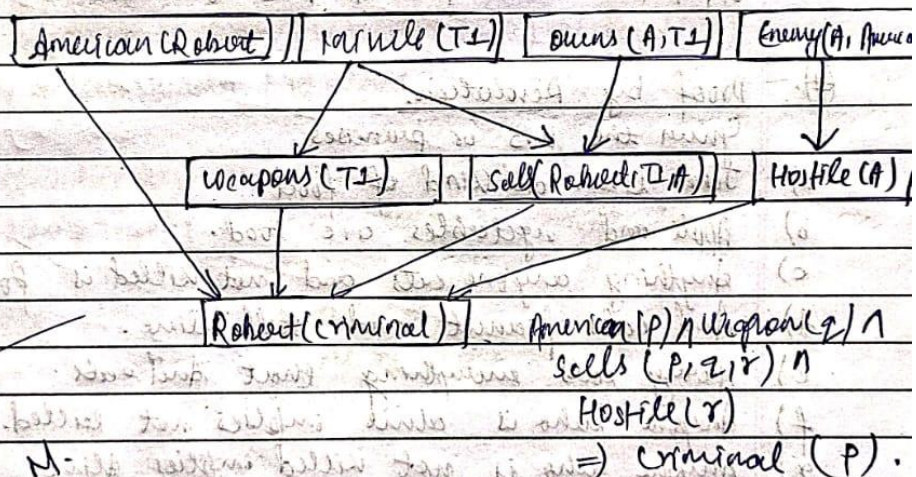
5. Enemy of America is known as hostile $\forall x$
Enemy $(x, \text{America}) \Rightarrow \text{Hostile}(x)$.

6. Robert is an American
American (Robert)

7. The country A, an enemy of America.
Enemy (A, America)

(##) Output:
Is 'Criminal (Robert)' provable? True.

(##). Forward chaining proof:



American (P) $\wedge$ Weapon (Q) $\wedge$
Sells (P, Q, R) $\wedge$
Hostile (R)
$\Rightarrow$ Criminal (P).

Nandee. M.
3/12/2024

CODE:

```
# Define the Fact class
class Fact:
    def __init__(self, name, *args):
        self.name = name
        self.args = args

    def __str__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

    def __eq__(self, other):
        return self.name == other.name and self.args == other.args

    def __hash__(self):
        return hash((self.name, self.args))

# Representing constants
A = "A"  # Country A
Robert = "Robert"  # Robert

# Initial facts
facts = set([
    Fact("American", Robert),
    Fact("Enemy", A, "America"),
    Fact("Missile", "T1"),  # We instantiate the missile T1
    Fact("Owns", A, "T1"),
    Fact("Weapon", "T1"),  # Missiles are weapons
    Fact("Sells", Robert, "T1", A),  # Robert sells missile to A
    Fact("Hostile", A)  # A is hostile because it's an enemy of America
])

# Inference rules
def apply_rules(facts):
    new_facts = set()

    # Rule: Missiles are weapons
    for fact in facts:
        if fact.name == "Missile":
            new_facts.add(Fact("Weapon", *fact.args))

    # Rule: If American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)
    for fact1 in facts:
        if fact1.name == "American":
            p = fact1.args[0]  # p is the person
            for fact2 in facts:
```

```
            if fact2.name == "Weapon":
                q = fact2.args[0]  # q is the weapon
                for fact3 in facts:
                    if fact3.name == "Sells" and fact3.args[0] == p and fact3.args[1] == q:
                        r = fact3.args[2]  # r is the country
                        for fact4 in facts:
                            if fact4.name == "Hostile" and fact4.args[0] == r:
                                new_facts.add(Fact("Criminal", p))

    return new_facts

# Forward chaining
def forward_chaining(facts):
    inferred_facts = set(facts)  # Start with the initial facts
    while True:
        new_facts = apply_rules(inferred_facts) - inferred_facts
        if not new_facts:  # No new facts inferred, stop the loop
            break
        inferred_facts.update(new_facts)
    return inferred_facts

# Run forward chaining
final_facts = forward_chaining(facts)

# Check if "Criminal(Robert)" is in the final facts
if Fact("Criminal", Robert) in final_facts:
    print("Criminal(Robert)")
else:
    print("Not Criminal")
```

OUTPUT:

```
Criminal(Robert)

In [ ]:
```

03/12/2024 **RESOLUTION**

ALGORITHM AND STATE SPACE TREE:

Mangal
Date
Page 30

Date: 03/12/2024:

(#). Project title:- Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.

(#). Algorithm:-

1) Convert all sentences to CNF.
2) Negate conclusion S & convert result to CNF.
3) Add negated conclusion S to the premise clauses.
4) Repeat until contradiction or no progress is made.
(a) Select 2 clauses (call them parent clauses)
(b) Resolve them together, performing all required unification.
(c) If resolvent is the empty clause, a contradiction has been found.
(d) If not, add resolvent to the premises. If we reach step 4, we have proved the conclusion.

(#). Proof by Resolution:

Given the KB or premises

a) John likes all kind of food.
b) Apple and vegetables are food.
c) Anything anyone eats and not killed is food.
d) Anil eats peanut and still alive.
e) Harry eats everything that Anil eats.
f) Anyone who is alive implies not killed.
g) Anyone who is not killed implies alive.
h) John likes peanuts.

⇒ Representation in FOL:

a) $\forall x: food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$

e) $\forall x$: eats (Anil, x) → eats (Harry, x)
f) $\forall y$: ¬ killed(x) → alive(x)
g) $\forall x$: alive(x) → ¬ killed(x)
h) likes (John, peanuts)

→ Eliminate implications:

a) $\forall x \neg$ food (x) V likes (John, x)
c) $\forall x \forall y \neg$ [eats (x, y) ∧ ¬ killed (x)] V food (y)
e) $\forall x \neg$ eats (Anil, x) V eats (Harry, x)
f) $\forall x \neg$ [¬ killed (x)] V alive (x)
g) $\forall x \neg$ alive (x) V ¬ killed (x)

→ Move negation (¬) inwards:

c) $\forall x y \neg$ eats (x, y) V killed (x) V food (y)
f) $\forall x$ killed (x)-] V alive (x)

→ Standardize variables:

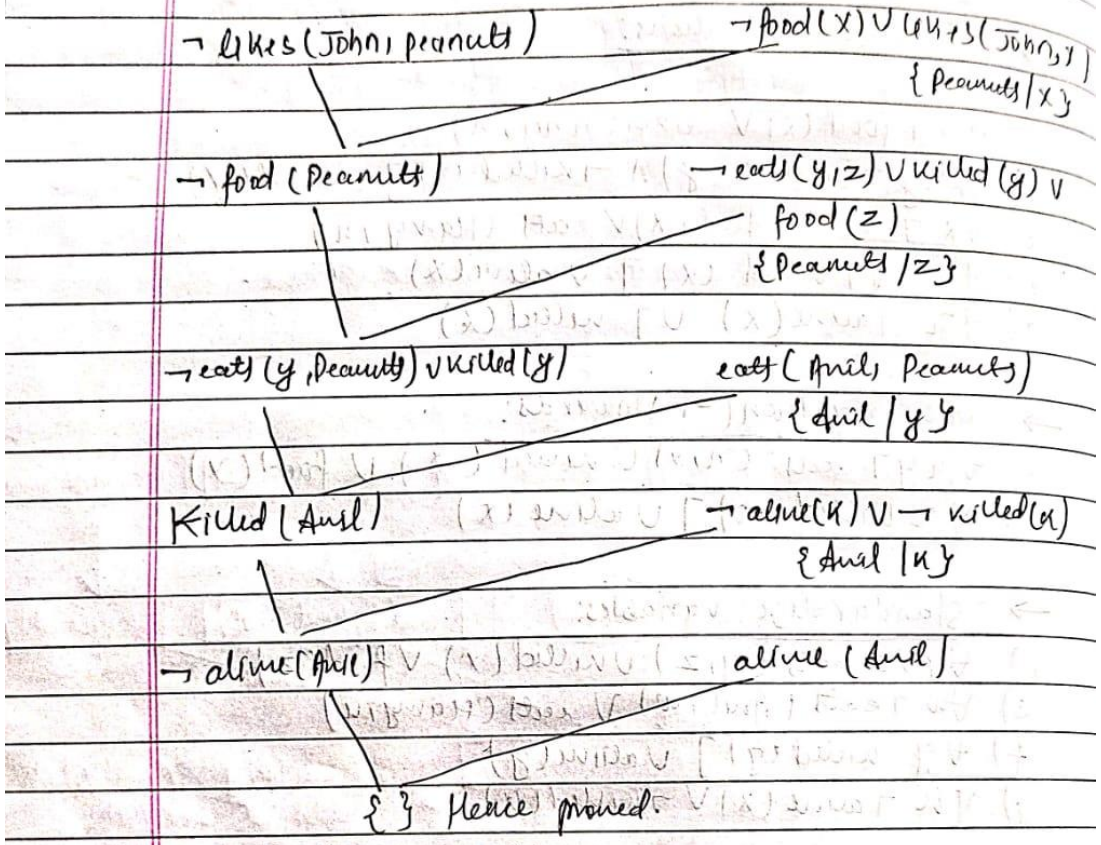
c) $\forall y \forall z$ eats (y, z) V killed (x) V food (z)
e) $\forall w \neg$ eats (Anil, w) V eats (Harry, w)
f) $\forall g$ killed (g] V alive (g)
g) $\forall k \neg$ alive (k) V ¬ killed (k)

→ Drop universal:

a) ¬ food (x) V likes (John, x)
b) food (Apple)
c) food (vegetables)
d) ¬ eats (y, z) V killed (y) V food (z)
e) eats (Anil, Peanuts)
f) alive (Anil)
g) ¬ eats (Anil, w) V eats (Harry, w)

h) killed(g) V alive(g)
i) ¬ alive(k) V ¬ killed(k)

(#). Proof:



CODE:

```
def is_variable(term):
    return term.islower() and "(" not in term and ")" not in term

def unify_literal(lit1, lit2):
    """Unify two literals if possible, returning a substitution map or None."""
    if lit1 == lit2:
        return {}

    if "(" in lit1 and "(" in lit2:
        pred1, args1 = lit1.split("(")
        pred2, args2 = lit2.split("(")
        if pred1 != pred2:
            return None
        args1 = args1[:-1].split(",")
        args2 = args2[:-1].split(",")
        subst = {}
        for a1, a2 in zip(args1, args2):
            if a1 != a2:
                if is_variable(a1):
                    subst[a1] = a2
                elif is_variable(a2):
                    subst[a2] = a1
                else:
                    return None
        return subst

    if is_variable(lit1):
        return {lit1: lit2}
    if is_variable(lit2):
        return {lit2: lit1}
    return None


def apply_substitution(clause, substitution):
    """Apply a substitution to a clause."""
    new_literals = []
    for lit in clause.literals:
        for var, const in substitution.items():
            lit = lit.replace(var, const)
        new_literals.append(lit)
    return Clause(new_literals)


class Clause:
```

```
    def __init__(self, literals):
        self.literals = set(literals)

    def __repr__(self):
        return " ∨ ".join(sorted(self.literals))

    def resolve(self, other):
        resolvents = []
        for literal in self.literals:
            for other_literal in other.literals:
                unifier = unify_literal(literal, f"¬{other_literal}" if not other_literal.startswith("¬")
else other_literal[1:])
                if unifier is not None:
                    new_clause = Clause((self.literals - {literal}) | (other.literals - {other_literal}))
                    resolvents.append(apply_substitution(new_clause, unifier))
        return resolvents


def resolution(clauses, query):
    negated_query = Clause([f"¬{query}"])
    clauses.append(negated_query)

    new = set()
    seen_pairs = set()

    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
        for ci, cj in pairs:
            if (ci, cj) in seen_pairs or (cj, ci) in seen_pairs:
                continue
            seen_pairs.add((ci, cj))

            resolvents = ci.resolve(cj)
            for resolvent in resolvents:
                if not resolvent.literals:
                    return True
                new.add(frozenset(resolvent.literals))

        if new.issubset(set(map(frozenset, (c.literals for c in clauses)))):
            return False
        clauses.extend(Clause(list(literals)) for literals in new - set(map(frozenset, (c.literals for c
in clauses))))
        new.clear()


# Knowledge Base (KB)
```

```
KB = [
    Clause(["¬food(x)", "likes(john, x)"]),
    Clause(["food(apple)"]),
    Clause(["food(vegetables)"]),
    Clause(["¬eats(y, z)", "killed(y)", "food(z)"]),
    Clause(["eats(anil, peanuts)"]),
    Clause(["alive(anil)"]),
    Clause(["¬eats(anil, w)", "eats(harry, w)"]),
    Clause(["killed(g)", "alive(g)"]),
    Clause(["¬alive(k)", "¬killed(k)"]),
]

# Query: Prove John likes peanuts
query = "likes(john, peanuts)"

if resolution(KB, query):
    print(f"The conclusion '{query}' is proven by resolution.")
else:
    print(f"The conclusion '{query}' cannot be proven.")
```

OUTPUT:

```
The conclusion 'likes(john, peanuts)' is proven by resolution.
```

```
In [ ]:
```

17/12/2024 **Alpha-Beta Pruning**

ALGORITHM AND STATE SPACE TREE:

(LAB-10)

Mangal
Date
Page 33

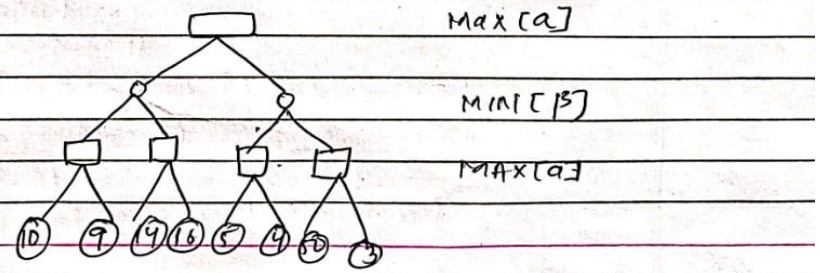
Date: 17/12/2024

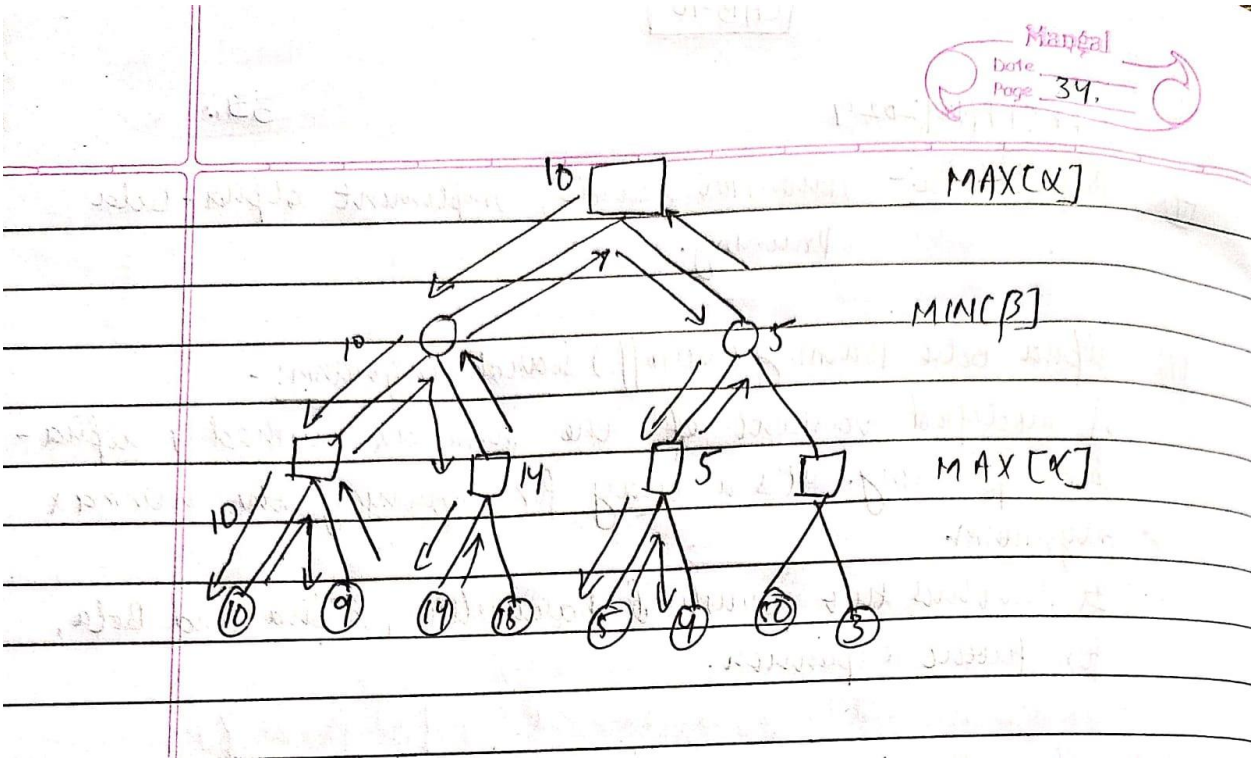
(#). Project title:- Adversial search, Implement Alpha-Beta Pruning.

(#). Alpha Beta Pruning (Cutoff) search Algorithm:-
A modified variant of the minmax method is alpha-beta pruning. It's a way for improving the minmax algorithm.
It involves two threshold parameters, Alpha and Beta, for future expansion.

(#). Algorithm:
- Alpha(α) - Beta (β) purposes to find the optimal path without looking at every node in the game tree.
- Max contains Alpha(α) and MIN contains Beta(β) bound during the calculation.
- In both MIN and MAX node, we return when α ≥ β which compares with its parent node only.
- Both minmax and Alpha(α) - Beta(β) cut-off give same path.
- Alpha(α) - Beta(β) gives optimal solution as it takes less time to get the value for the root node.

(#). Problem:





CODE:

```
import math

# Alpha-Beta Pruning Functions
def alpha_beta_search(state):
    """ Alpha-Beta Search to get the optimal action """
    value = max_value(state, -math.inf, math.inf)
    print("Optimal Value:", value)
    return value

def max_value(state, alpha, beta):
    """ Function to calculate the MAX value node """
    if terminal_test(state):  # If leaf node, return utility value
        return utility(state)
    v = -math.inf
    for child in state["children"]:  # Iterate through child nodes
        v = max(v, min_value(child, alpha, beta))
        if v >= beta:
            return v  # Beta cutoff
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta):
    """ Function to calculate the MIN value node """
```

```
    if terminal_test(state):  # If leaf node, return utility value
        return utility(state)
    v = math.inf
    for child in state["children"]:  # Iterate through child nodes
        v = min(v, max_value(child, alpha, beta))
        if v <= alpha:
            return v  # Alpha cutoff
        beta = min(beta, v)
    return v

# Utility Functions
def terminal_test(state):
    """ Check if the node is a leaf node """
    return "value" in state  # Leaf node if it contains 'value'

def utility(state):
    """ Return the utility value of a leaf node """
    return state["value"]

# Build the Binary Tree Based on Leaf Nodes
def build_tree(values):
    """ Recursively build a binary tree from a list of leaf node values """
    if len(values) == 1:  # Single value -> Leaf node
        return {"value": values[0]}
    mid = len(values) // 2
    left_subtree = build_tree(values[:mid])
    right_subtree = build_tree(values[mid:])
    return {"children": [left_subtree, right_subtree]}

# Main Program
if __name__ == "__main__":
    leaf_nodes = [10, 9, 14, 18, 5, 4, 50, 3]
    tree = build_tree(leaf_nodes)  # Build the binary tree
    print("Alpha-Beta Pruning Search:")
    alpha_beta_search(tree)
```

OUTPUT:

```
Alpha-Beta Pruning Search:
Optimal Value: 10
```

```
In [ ]:
```