VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shivsundar sah(1BM22CS258)

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 **Shivsundar sah(1BM22CS258),** 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Shivsundarsah/Shivsundarsah-AI-.git

Lab-> 1

{2024/09/24 Monday}

1) Tic-Tac-Toe

a) Algorithm :- Tic-Tac-Toe()

Step1: Create a 3x3 3 Dimensional array filled with underscores ('_') to indicate empty spaces

Step2:- Set current player to 'X'.

validate the input: if row<0 or row>2 or board[i][j]=='_'

b) check for winner

check all rows, columns and diagonals to see if any player having any three symbols in line.

c) Check for draw

If no winner is found, check if the board is full

d) switch players

If the game is still ongoing switch current player from 'X' to 'O' or from 'O' to 'X'

e) Update the board.

Place the current player's symbol ('X' or 'O') in the selected cell.

f) Display the board.

Print the Board along with its current state whether winner is found or not or it's a draw.

2024/09/01
Tuesday

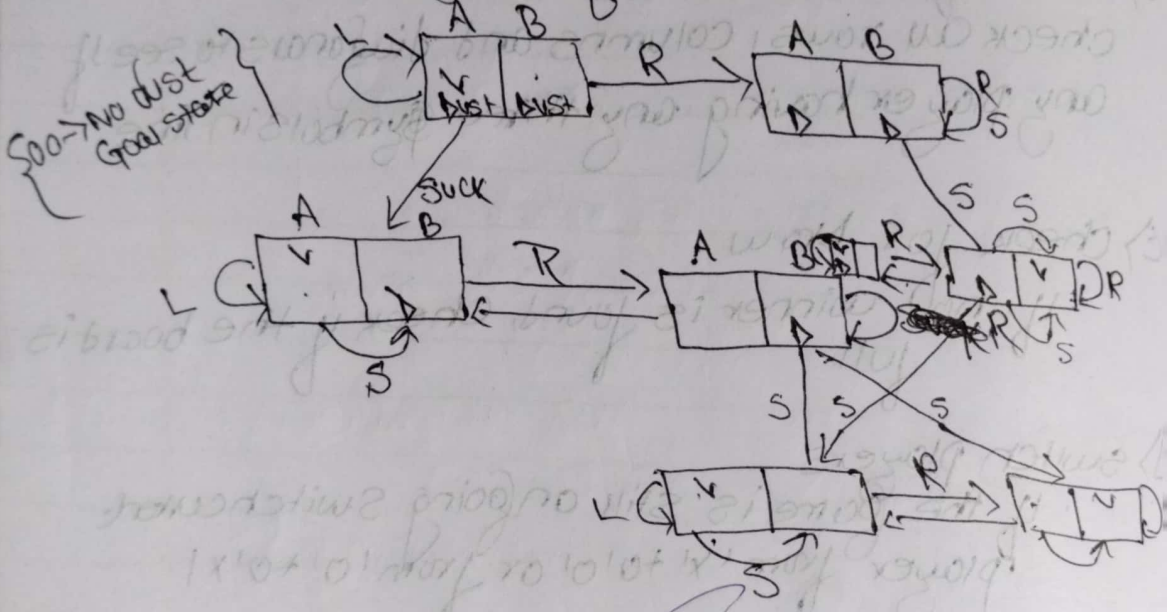
Lab-> 7 Implement vaccum world cleaner.

Pseudo code:

function REFLEX-VACCUM-AGENT([Location, Status]) returns an action

if Status = Dirty then return Suck
else if location = A then return right
else if location = B then return left

State space diagram of vaccum world



problem formulation step:

* State
* Initial state
* Actions
* Transition model
* Goal test
* Path cost

Algorithm: Vaccum world cleaner

1) Start the vaccum cleaner
- Turn on the power switch

2) Create suction
- Activate the rotor to spin the fan
- Generate low pressure inside the vaccum cleaner

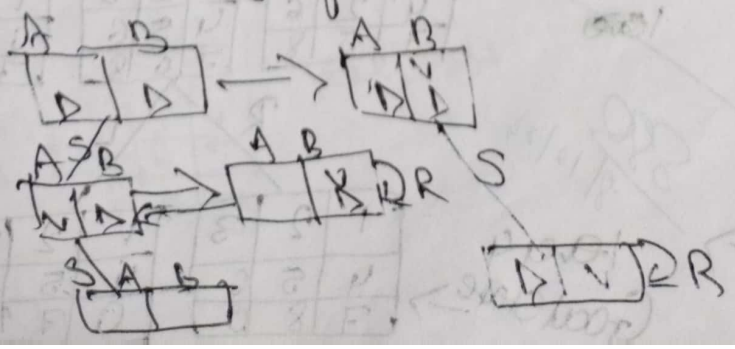
3) Draw Air and Dirt
- Air is pulled in through nozzle
- Dirt and debris are carried into the vaccum by the incomming airflow

4) Collect dirt.
- The incomming air enters the dust container or bag

5) Filter the air
→ The airflow passes through the filters
→ Dust and allergens are trapped, preventing them from escaping.

6) Regular maintaince
→ Clean or replace filter as needed

7) End cleaning session.
→ Turn off vaccum cleaner
→ Store it properly for feature use.



Lab:-1 Implement Tic –Tac –Toe Game

1.Implement Tic Tac Toe game.

```
def print_board(board):
    for row in board:
        print(" | ".join(row))

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '_':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != '_':
            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != '_':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != '_':
        return board[0][2]

    return None

def is_draw(board):
    for row in board:
        if '_' in row:
            return False
    return True

def tic_tac_toe():
    board = [['_' for _ in range(3)] for _ in range(3)]
    current_player = 'X'
```

```
    while True:
        print_board(board)
        try:
            row = int(input(f"Player {current_player}, enter the row (0, 1, 2): "))
            col = int(input(f"Player {current_player}, enter the column (0, 1, 2): "))

            if row < 0 or row > 2 or col < 0 or col > 2 or board[row][col] != '_':
                print("Invalid move. Try again.")
                continue

            board[row][col] = current_player

            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"Player {winner} wins!")
                break

            if is_draw(board):
                print_board(board)
                print("It's a draw!")
                break
            # Switch players
            current_player = 'O' if current_player == 'X' else 'X'

        except ValueError:
            print("Please enter valid integers for row and column.")

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

output:

```
_ | _ | _
_ | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 0
X | _ | _
_ | _ | _
_ | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 0
X | _ | _
O | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 1
X | X | _
O | _ | _
_ | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 1
X | X | _
O | O | _
_ | _ | _
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 2
X | X | X
O | O | _
_ | _ | _
Player X wins!
```

```
_ | _ | _
_ | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 3
Player X, enter the column (0, 1, 2): 1
Invalid move. Try again.
_ | _ | _
_ | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2):
```

```
Player O, enter the row (0, 1, 2): 0
Player O, enter the column (0, 1, 2): 2
O | X | O
X | _ | _
_ | _ | _
Player X, enter the row (0, 1, 2): 2
Player X, enter the column (0, 1, 2): 0
O | X | O
X | _ | _
X | _ | _
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 1
O | X | O
X | O | _
X | _ | _
Player X, enter the row (0, 1, 2): 1
Player X, enter the column (0, 1, 2): 2
O | X | O
X | O | X
X | _ | _
Player O, enter the row (0, 1, 2): 2
Player O, enter the column (0, 1, 2): 1
O | X | O
X | O | X
X | O | _
Player X, enter the row (0, 1, 2): 2
Player X, enter the column (0, 1, 2): 2
O | X | O
X | O | X
X | O | X
It's a draw!
```

Lab->2

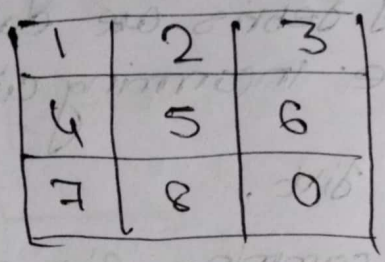
2024 | 10 | 08
Tuesday

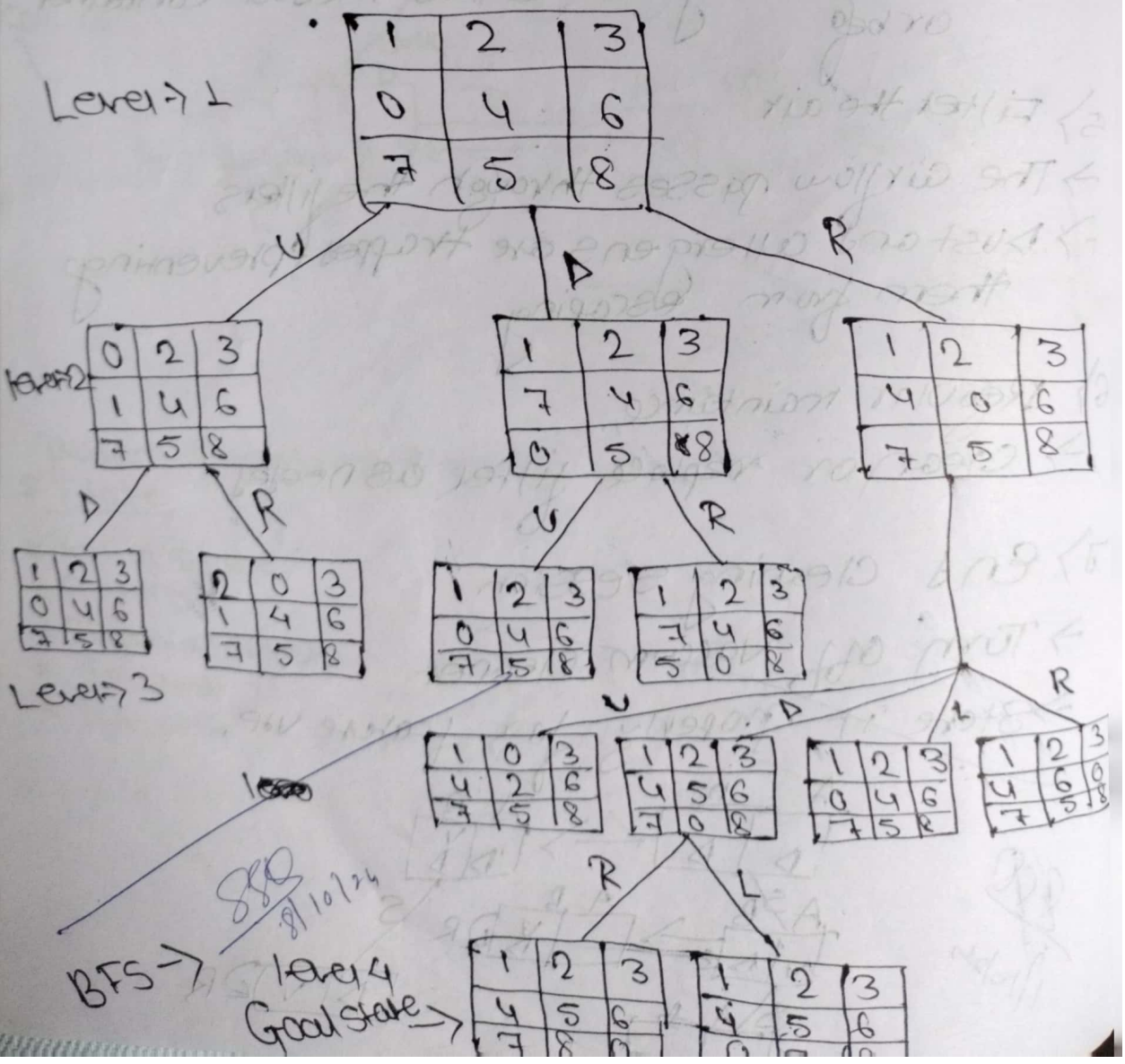
#Solve 8puzzle-problem using DFS and BFS.

Non-Heuristic Algorithm:-(BFS)

Initial State
->Create a 3X3 grid
->Represent each state with a tuple containing number between (0 to 8)

Goal state
->Set a Goal state



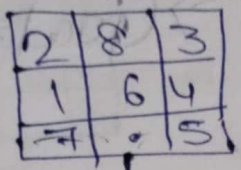


a) Draw the state space diagram for the given graph (Missplaced)

Tuesday

$f = g + h$

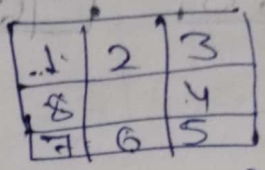
Initial state



$g = 0$

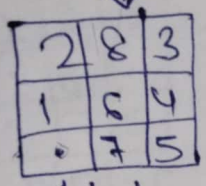
$h = 4$

Goal state



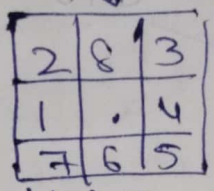
First see node Misplaced no and shift to Left and the minimum

$g = 1$



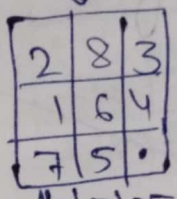
$h(n) = 5$

$f(n) = 6$



$h(n) = 3$

$f(n) = 4$

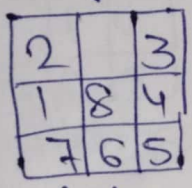


$h(n) = 5$

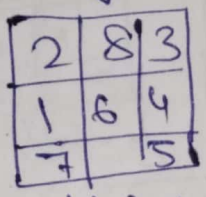
$f(n) = 6$

Level => 1

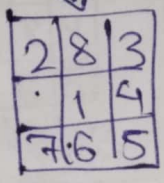
$g = 2$



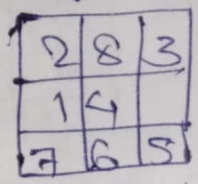
$h(n) = 5$



$h(n) = 4$



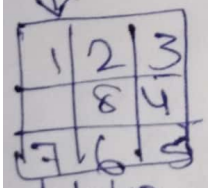
$h(n) = 3$



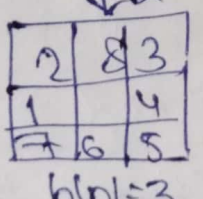
$h(n) = 4$

Level => 2

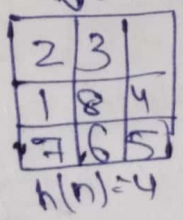
$g = 3$



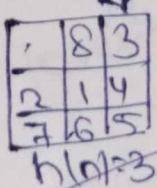
$h(n) = 2$



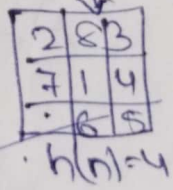
$h(n) = 3$



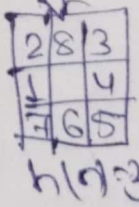
$h(n) = 4$



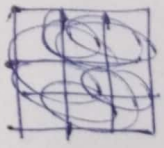
$h(n) = 3$



$h(n) = 4$



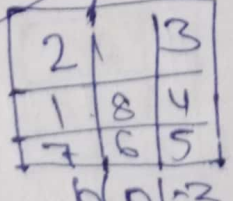
$h(n) = 3$



Level => 3

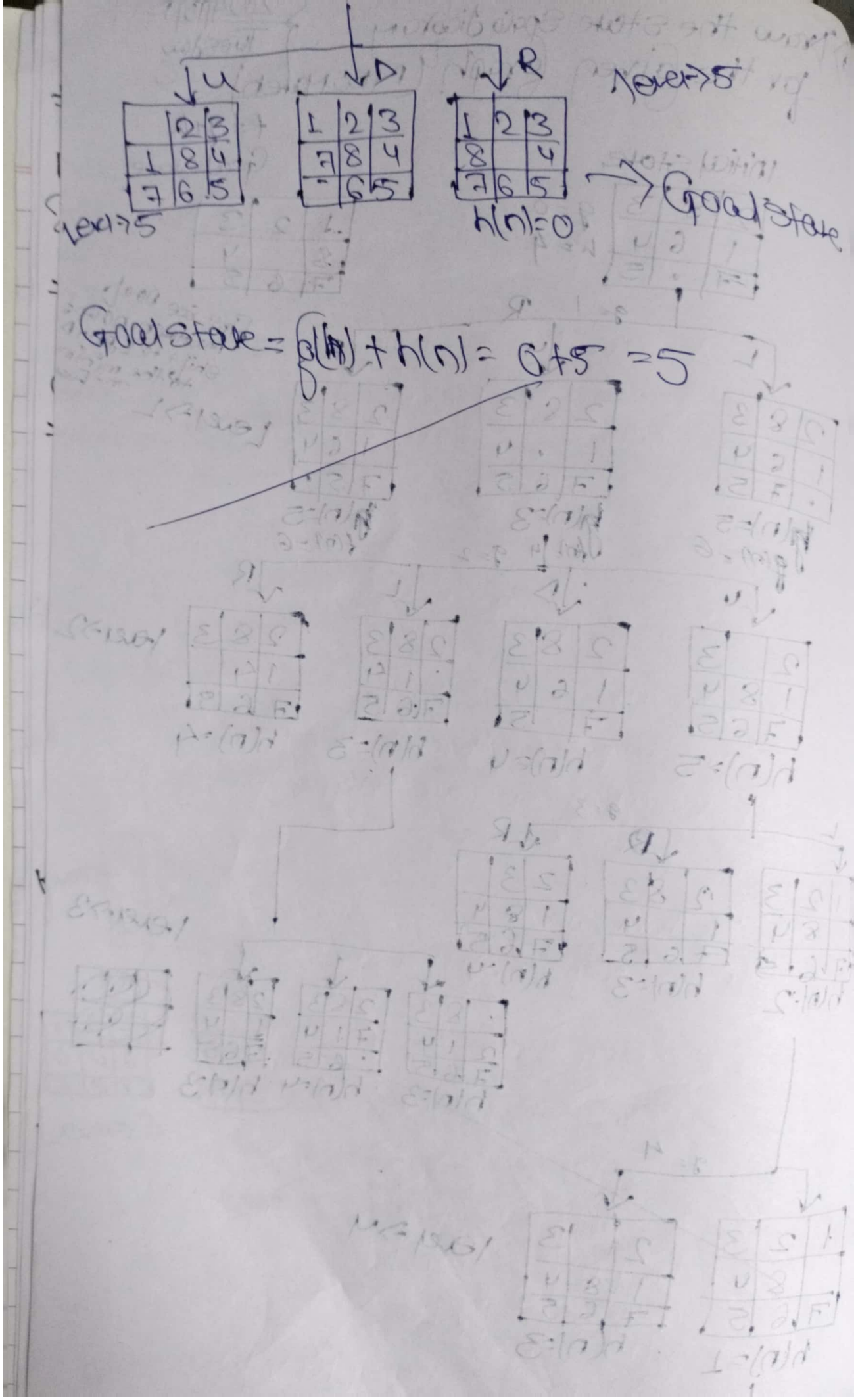
$g = 4$

$h(n) = 1$

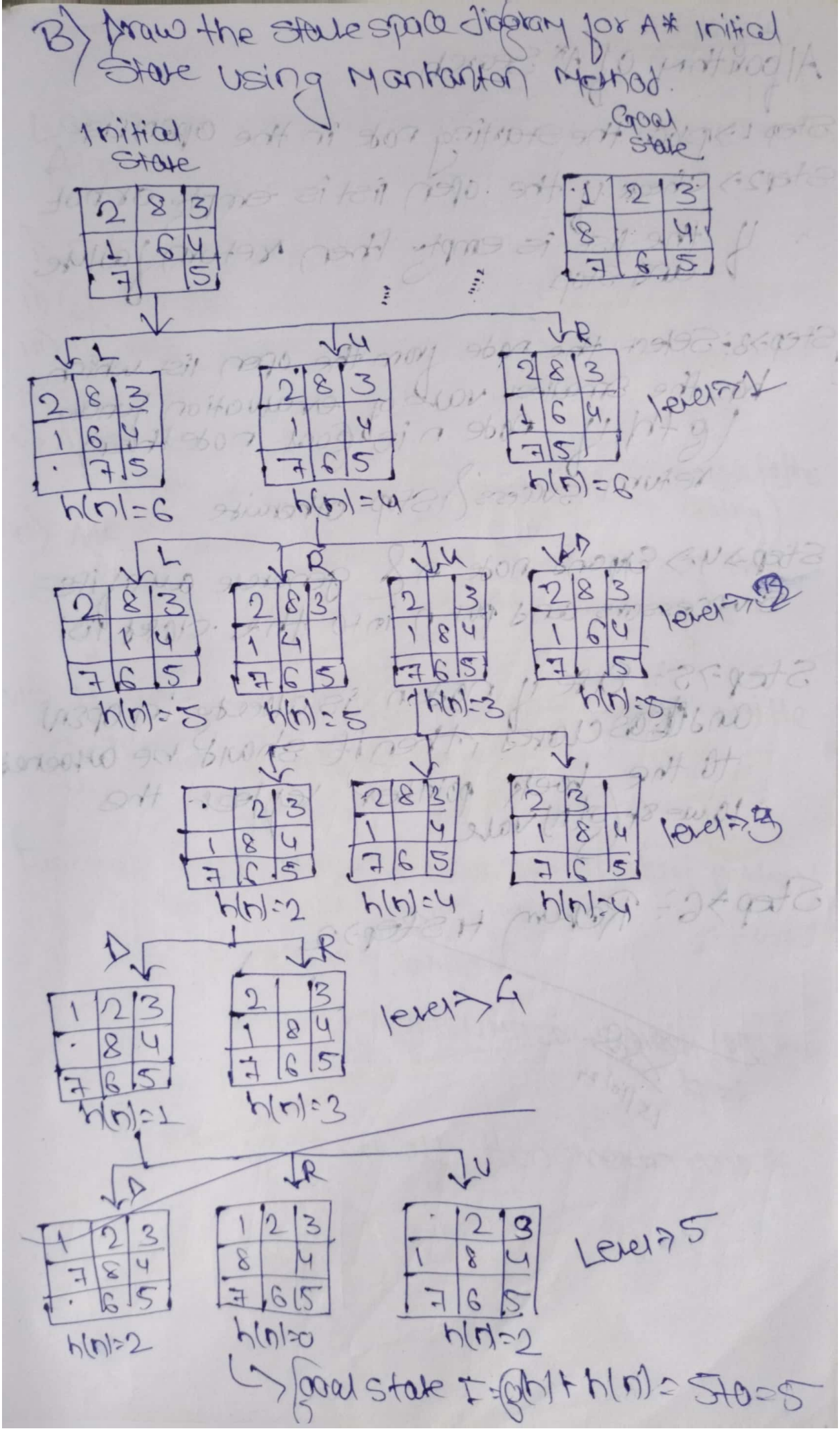


$h(n) = 3$

Level => 4


U
D
R
level=5
level=5
h(n)=0
Goal state
Goal state = g(n) + h(n) = 0 + 5 = 5

B) Draw the state space diagram for A* initial state using Manhattan Method.



Algorithm of A* Search

Step 1 => place the starting node in the open list

Step 2 => Check if the open list is empty or not
If the list is empty then return failure and stop

Step => 3: Select the node from the open list which has the smallest value of evaluation function (g+h), if node n is goal node then return success & stop otherwise

Step => 4 => Expand node n & generate all of its successors and put n into the closed list

Step => 5: Else if node n is already in open and closed, then it should be attached to the back pointer which reflect the lowest g(n) value

Step => 6: Return to Step => 2

15/10/24

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

```python
initial_states = []
cost = 0
goal_state = ["A", 0, "B", 0]

def vacuum_world(location):
    global cost
    current_state = ["A", initial_states[0], "B", initial_states[1]]
    print("Current state:",current_state)
    while current_state[1] != 0 or current_state[3] != 0:
        if location == "A":
            if current_state[1] == 1:
                print("Location A is dirty.")
                print("Cleaning Location A...")
                cost += 1
                current_state[1] = 0
                print("Cost for suck:", cost)
            print("Moving to B...")
            location = "B"
        elif location == "B":
            if current_state[3] == 1:
                print("Location B is dirty.")
                print("Cleaning Location B...")
                cost += 1
                current_state[3] = 0
                print("Cost for suck:", cost)
            print("Moving to A...")
            location = "A"
        print("Current state:",current_state)
    print("Final state:", current_state)
    print("Total cost:", cost)

for room in ['A', 'B']:
    state = input(f"Is Room {room} dirty? (1 for dirty, 0 for clean): ").strip()
    while state not in ['0', '1']:
        print("Invalid input. Please enter 1 for dirty or 0 for clean.")
        state = input(f"Is Room {room} dirty? (1 for dirty, 0 for clean): ").strip()
    initial_states.append(int(state))

location = input("Enter the starting location of the vacuum cleaner (A or B): ").strip().upper()
while location not in ['A', 'B']:
    print("Invalid location. Please enter 'A' or 'B'.")
    location = input("Enter the starting location of the vacuum cleaner (A or B): ").strip().upper()

vacuum_world(location)
```

```
Is Room A dirty? (1 for dirty, 0 for clean): 1
Is Room B dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A or B): A
Current state: ['A', 1, 'B', 1]
Location A is dirty.
Cleaning Location A...
Cost for suck: 1
Moving to B...
Current state: ['A', 0, 'B', 1]
Location B is dirty.
Cleaning Location B...
Cost for suck: 2
Moving to A...
Current state: ['A', 0, 'B', 0]
Final state: ['A', 0, 'B', 0]
Total cost: 2
```

```
Is Room A dirty? (1 for dirty, 0 for clean): 0
Is Room B dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A or B): A
Current state: ['A', 0, 'B', 1]
Moving to B...
Current state: ['A', 0, 'B', 1]
Location B is dirty.
Cleaning Location B...
Cost for suck: 1
Moving to A...
Current state: ['A', 0, 'B', 0]
Final state: ['A', 0, 'B', 0]
Total cost: 1
```

```
Is Room A dirty? (1 for dirty, 0 for clean): 1
Is Room B dirty? (1 for dirty, 0 for clean): 0
Enter the starting location of the vacuum cleaner (A or B): A
Current state: ['A', 1, 'B', 0]
Location A is dirty.
Cleaning Location A...
Cost for suck: 1
Moving to B...
Current state: ['A', 0, 'B', 0]
Final state: ['A', 0, 'B', 0]
Total cost: 1
```

vaccum world

```python
initial_states = []
cost = 0
goal_state = [0, 0, 0, 0]

def vacuum_world(location):
    global cost
    current_state = [initial_states[0], initial_states[1], initial_states[2], initial_states[3]]
    print("Current state:", current_state)

    while current_state != goal_state:
        if location == "A":
            if current_state[0] == 1:
                print("Location A is dirty.")
                print("Cleaning Location A...")
                cost += 1
                current_state[0] = 0
                print("Cost for cleaning:", cost)
            print("Moving to B...")
            location = "B"
        elif location == "B":
            if current_state[1] == 1:
                print("Location B is dirty.")
                print("Cleaning Location B...")
                cost += 1
                current_state[1] = 0
                print("Cost for cleaning:", cost)
            print("Moving to C...")
            location = "C"
        elif location == "C":
            if current_state[2] == 1:
                print("Location C is dirty.")
                print("Cleaning Location C...")
                cost += 1
                current_state[2] = 0
                print("Cost for cleaning:", cost)
            print("Moving to D...")
            location = "D"
        elif location == "D":
            if current_state[3] == 1:
                print("Location D is dirty.")
                print("Cleaning Location D...")
                cost += 1
                current_state[3] = 0
                print("Cost for cleaning:", cost)
            print("Moving to A...")
            location = "A"
        print("Current state:", current_state)
    print("Final state:", current_state)
    print("Total cost:", cost)

for quadrant in ['A', 'B', 'C', 'D']:
    state = input(f"Is Quadrant {quadrant} dirty? (1 for dirty, 0 for clean): ").strip()
    while state not in ['0', '1']:
        print("Invalid input. Please enter 1 for dirty or 0 for clean.")
        state = input(f"Is Quadrant {quadrant} dirty? (1 for dirty, 0 for clean): ").strip()
    initial_states.append(int(state))

location = input("Enter the starting location of the vacuum cleaner (A, B, C, or D): ").strip().upper()
while location not in ['A', 'B', 'C', 'D']:
    print("Invalid location. Please enter 'A', 'B', 'C', or 'D'.")
    location = input("Enter the starting location of the vacuum cleaner (A, B, C, or D): ").strip().upper()

vacuum_world(location)
```

```
Is Quadrant A dirty? (1 for dirty, 0 for clean): 1
Is Quadrant B dirty? (1 for dirty, 0 for clean): 1
Is Quadrant C dirty? (1 for dirty, 0 for clean): 1
Is Quadrant D dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A, B, C, or D): A
Current state: [1, 1, 1, 1]
Location A is dirty.
Cleaning Location A...
Cost for cleaning: 1
Moving to B...
Current state: [0, 1, 1, 1]
Location B is dirty.
Cleaning Location B...
Cost for cleaning: 2
Moving to C...
Current state: [0, 0, 1, 1]
Location C is dirty.
Cleaning Location C...
Cost for cleaning: 3
Moving to D...
Current state: [0, 0, 0, 1]
Location D is dirty.
Cleaning Location D...
Cost for cleaning: 4
Moving to A...
Current state: [0, 0, 0, 0]
Final state: [0, 0, 0, 0]
Total cost: 4
```

```
Is Quadrant A dirty? (1 for dirty, 0 for clean): 0
Is Quadrant B dirty? (1 for dirty, 0 for clean): 1
Is Quadrant C dirty? (1 for dirty, 0 for clean): 0
Is Quadrant D dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A, B, C, or D): A
Current state: [0, 1, 0, 1]
Moving to B...
Current state: [0, 1, 0, 1]
Location B is dirty.
Cleaning Location B...
Cost for cleaning: 1
Moving to C...
Current state: [0, 0, 0, 1]
Moving to D...
Current state: [0, 0, 0, 1]
Location D is dirty.
Cleaning Location D...
Cost for cleaning: 2
Moving to A...
Current state: [0, 0, 0, 0]
Final state: [0, 0, 0, 0]
Total cost: 2
```

```
Is Quadrant A dirty? (1 for dirty, 0 for clean): 0
Is Quadrant B dirty? (1 for dirty, 0 for clean): 0
Is Quadrant C dirty? (1 for dirty, 0 for clean): 0
Is Quadrant D dirty? (1 for dirty, 0 for clean): 1
Enter the starting location of the vacuum cleaner (A, B, C, or D): A
Current state: [0, 0, 0, 1]
Moving to B...
Current state: [0, 0, 0, 1]
Moving to C...
Current state: [0, 0, 0, 1]
Moving to D...
Current state: [0, 0, 0, 1]
Location D is dirty.
Cleaning Location D...
Cost for cleaning: 1
Moving to A...
Current state: [0, 0, 0, 0]
Final state: [0, 0, 0, 0]
Total cost: 1
```

Lab → 3 10/24

Algorithm of A* Search

Step 1 → place the starting node in the open list

Step 2 → Check if the open list is empty or not
If the list is empty then return failure and stop

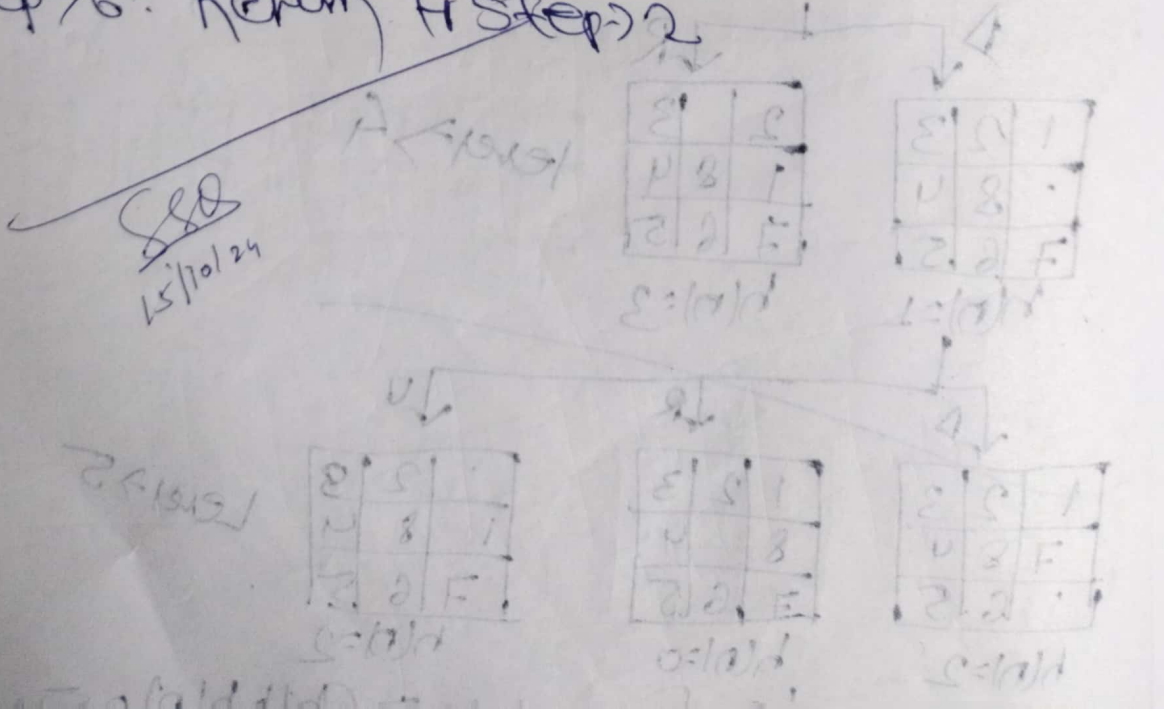
Step → 3 : Select the node from the open list which has the smallest value of evaluation function (g+h), if node n is goal node then return success & stop otherwise

Step → 4 → Expand node n & generate all of its successors and put n into the closed list

Step → 5 : Else if node n is already in open and closed, then it should be attached to the back pointer keeper the lowest g(n) value

Step → 6 : Return to Step → 2

15/10/24



lab:-3 Implement A* search algorithm

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state          # The current state of the puzzle
        self.parent = parent        # The parent node
        self.move = move            # The move taken to reach this state
        self.cost = cost            # The cost to reach this node

    def heuristic(self):
        """Count the number of misplaced tiles."""
        goal_state = [[1,2,3], [8,0,4], [7,6,5]]
        count = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] != 0 and self.state[i][j] != goal_state[i][j]:
                    count += 1
        return count

def get_blank_position(state):
    """Find the position of the blank (0) in the state."""
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    """Get possible moves from the blank position."""
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))  # Up
    if x < 2: moves.append((x + 1, y, 'Up'))    # Down
    if y > 0: moves.append((x, y - 1, 'Right'))  # Left
```

```
    if y < 2: moves.append((x, y + 1, 'Left'))   # Right
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    """Generate a new state by moving the blank tile."""
    new_state = [row[:] for row in state]  # Deep copy
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]]
        new_state[new_blank_pos[0]][new_blank_pos[1]],
new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    """Perform A* search."""
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:
        # Sort the open list by total estimated cost (cost + heuristic)
        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)

        # Print the current state, move, and heuristic value
        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (misplaced tiles): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")
```

```
        if current_node.heuristic() == 0:  # Goal state reached
            # Construct the path
            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]  # Get the direction of the move
            new_node = Node(state=new_state, parent=current_node, move=move_direction,
cost=cost)

            if new_node not in open_list:  # Avoid duplicates in the open list
                open_list.append(new_node)

    return None  # No solution found

# Example usage:
initial_state = [[2,8,3], [1,6,4], [7,0,5]]  # An example initial state
solution_path = a_star_search(initial_state)
```

```
if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

Current state:

[2, 8, 3]

[1, 6, 4]

[7, 0, 5]

Move: Start

Heuristic value (misplaced tiles): 4

Cost to reach this node: 0

Current state:

[2, 8, 3]

[1, 0, 4]

[7, 6, 5]

Move: Down

Heuristic value (misplaced tiles): 3

Cost to reach this node: 1

Current state:

[2, 0, 3]

[1, 8, 4]

[7, 6, 5]

Move: Down

Heuristic value (misplaced tiles): 3

Cost to reach this node: 2

Current state:

[2, 8, 3]

[0, 1, 4]

[7, 6, 5]

Move: Right

Heuristic value (misplaced tiles): 3

Cost to reach this node: 2

Current state:

[0, 2, 3]

[1, 8, 4]

[7, 6, 5]

Move: Right

Heuristic value (misplaced tiles): 2

Cost to reach this node: 3

Current state:

[1, 2, 3]

[0, 8, 4]

[7, 6, 5]

Move: Up

Heuristic value (misplaced tiles): 1

Cost to reach this node: 4

Current state:

[1, 2, 3]

[8, 0, 4]

[7, 6, 5]

Move: Left

Heuristic value (misplaced tiles): 0

Cost to reach this node: 5

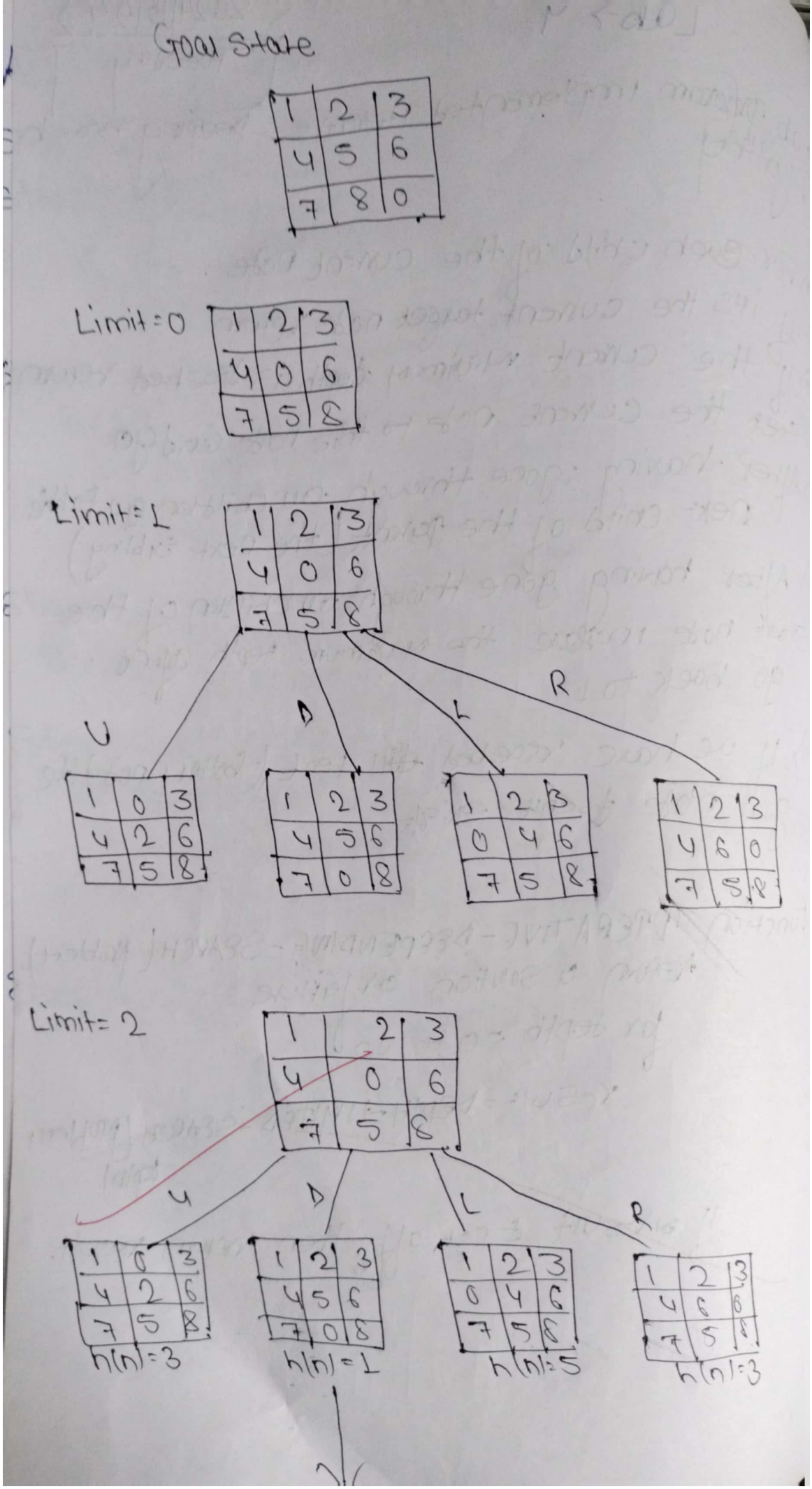
Lab -> 4

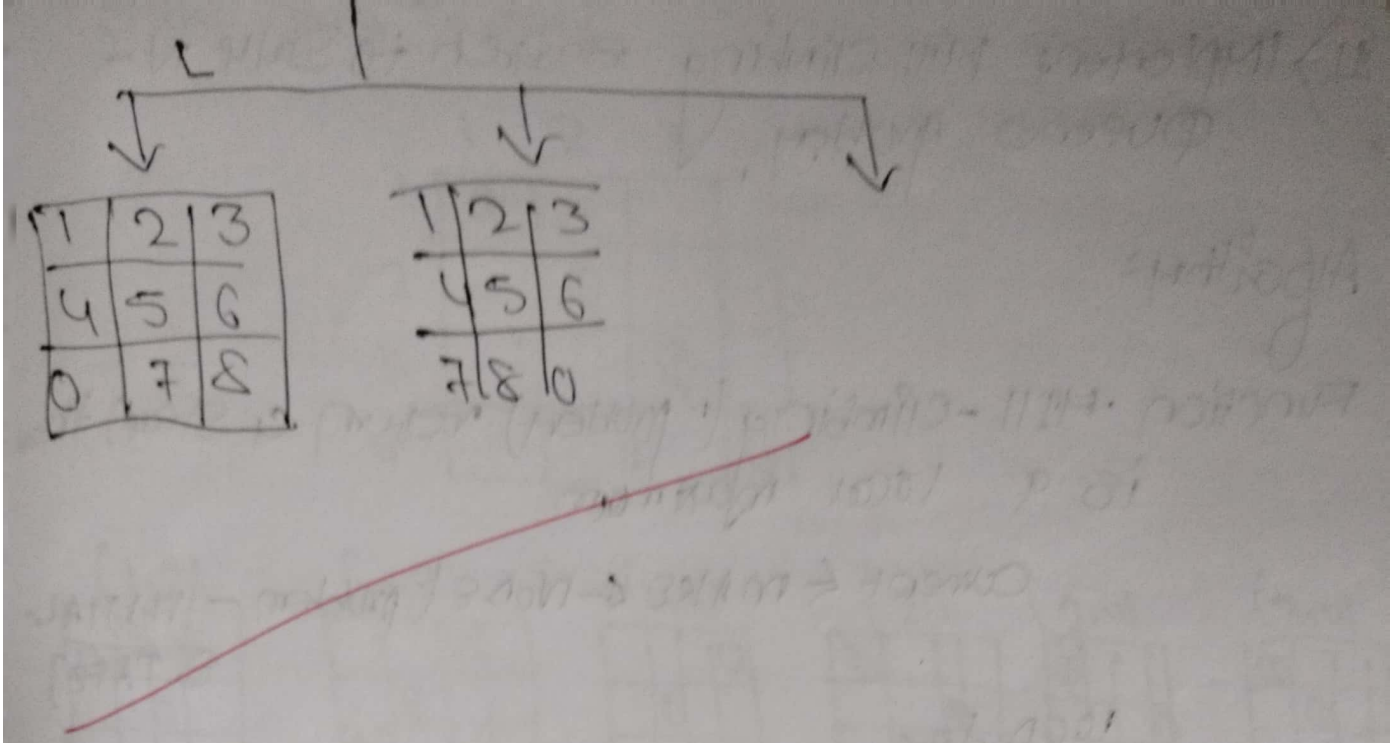
2024/10/24
Tuesday

Lab program implemented Iterative Deepening Search Algorithm

i) For each child of the current node
ii) If its the current target node return
iii) If the current maximum depth is reached return
iv) Set the current node to this node and go
v) After having gone through all children go to the next child of the parent (the next sibling)
vi) After having gone through all children of the start node increase the maximum depth agent go back to 1
vii) If we have reached the leaf (bottom node) the goal node doesn't exist

Function ITERATIVE-DEEPENDING-SEARCH(problem)
returns a solution or failure
for depth = 0 to ∞ do
result = DEPTH-LIMITED-SEARCH(problem, depth)
if result ≠ cut off then return result.


Goal State
1 2 3
4 5 6
7 8 0
Limit=0
1 2 3
4 0 6
7 5 8
Limit=1
1 2 3
4 0 6
7 5 8
U
D
L
R
1 0 3
4 2 6
7 5 8
1 2 3
4 5 6
7 0 8
1 2 3
0 4 6
7 5 8
1 2 3
4 6 0
7 5 8
Limit= 2
1 2 3
4 0 6
7 5 8
U
D
L
R
1 0 3
4 2 6
7 5 8
h(n)=3
1 2 3
4 5 6
7 0 8
h(n)=1
1 2 3
0 4 6
7 5 8
h(n)=5
1 2 3
4 6 0
7 5 8
h(n)=3


1 2 3
4 5 6
0 7 8
1 2 3
4 5 6
7 8 0

11) Implement Hill climbing search to solve N-Queens problem.

Algorithm:

Function HILL-CLIMBING (problem) return a state that is a local maximum

current ← MAKE-NODE (problem - INITIAL STATE)

loop do

neighbor ← a highest-valued successor of current

if neighbor value ≤ current value then return current STATE

current ← neighbor.

State → 4 queens on the board one queen per column
- variable x_0, x_1, x_2, x_3 where x_i is the row position of the queen per column.

→ Domain for each variable $x_i \leftarrow [0,1,2,3]$, $\forall i$

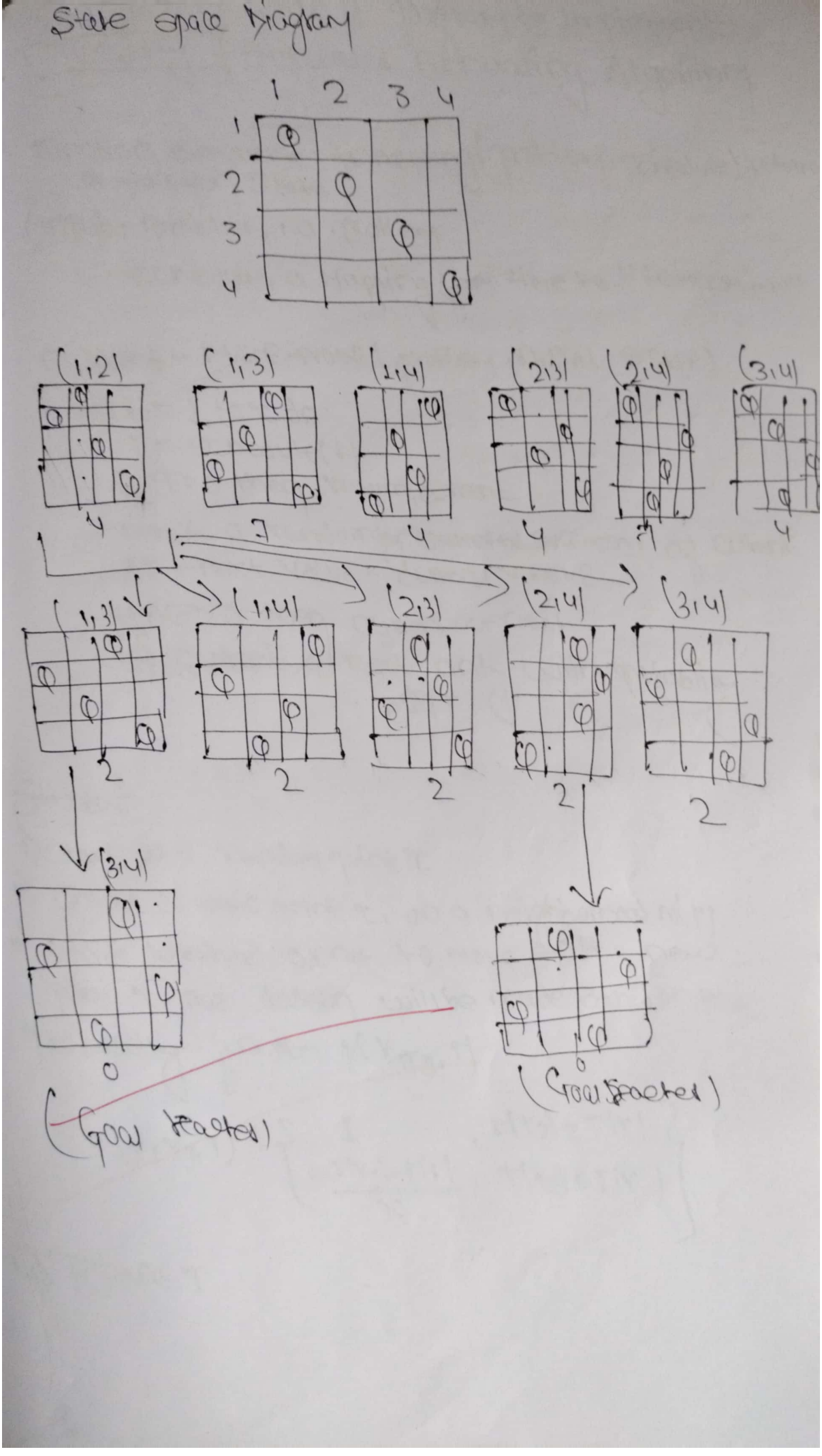
Initial state → a random state

- Goal state → 4 queens on the board. No pairs of queen are attacking each other

- Neighbour relation → swap to row position of 2 queens

- Cost function: The no. of pairs of queens attacking each other directly or indirectly

State space Diagram



Lab :-4 Implement Hill Climbing search algorithm to solve N-Queens problem

1) 8 PUZZLE USING ITERATIVE DEEPENING DEPTH FIRST SEARCH ALGORITHM

Code:

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.depth = depth
        self.path = path  # Keep track of the path taken to reach this state

    def is_goal(self, goal):
        return self.board == goal

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')] # up, down, left, right
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]  # Update the path with the new move
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1,
new_path))
        return moves
```

```
    def display(self):
        # Display the board in a matrix form
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")  # Display the moves taken to reach this state
        print()  # Newline for better readability


def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None


def dls(state, goal, depth):
    if state.is_goal(goal):
        return state
    if depth <= 0:
        return None
```

```
    for move in state.generate_moves():
        print("Current state:")
        move.display()  # Display the current state
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None


def main():
    # User input for initial state, goal state, and maximum depth
    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): ")
    max_depth = int(input("Enter maximum depth: "))
    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3  # Calculate the position of the empty tile
    initial_state = PuzzleState(initial_board, empty_tile_pos)
    solution = iddfs(initial_state, goal_board, max_depth)


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

OUTPUT :

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 0 8
Enter maximum depth: 10
Searching at depth: 0
Searching at depth: 1
Current state:
Moves: ['Up']
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

Current state:
Moves: ['Left']
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Moves: ['Left']
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Goal found at depth: 1
```

2) N QUEENS PROBLEM USING HILL CLIMBING METHOD

Code:

```
import random


def calculate_cost(board):
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:  # Same column
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal
                attacks += 1
    return attacks


def get_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Only change the row of the queen
```

```
            new_board = board[:]
            new_board[col] = row
            neighbors.append(new_board)
    return neighbors

def hill_climb(board, max_restarts=100):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)
    iteration = 0
    restarts = 0

    while restarts < max_restarts:  # Add limit to the number of restarts
        while current_cost != 0:  # Continue until cost is zero
            neighbors = get_neighbors(board)
            best_neighbor = None
            best_cost = current_cost

            for neighbor in neighbors:
                cost = calculate_cost(neighbor)
                if cost < best_cost:  # Looking for a lower cost
                    best_cost = cost
                    best_neighbor = neighbor

            if best_neighbor is None:  # No better neighbor found
                break  # Break the loop if we are stuck at a local minimum
```

```
            board = best_neighbor
            current_cost = best_cost
            iteration += 1
            print(f"Iteration {iteration}:")
            print_board(board, current_cost)

        if current_cost == 0:
            break  # We found the solution, no need for further restarts
        else:
            # Restart with a new random configuration
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]
            current_cost = calculate_cost(board)
            restarts += 1
            print(f"Restart {restarts}:")
            print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]  # Create an empty board
    for col in range(n):
        display_board[board[col]][col] = 'Q'  # Place queens on the board
    for row in range(n):
        print(' '.join(display_board[row]))  # Print the board
```

```
    print(f"Cost: {cost}\n")


if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-separated): ").split()))
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        if cost == 0:
            print(f"Solution found with no conflicts:")
        else:
            print(f"No solution found within the restart limit:")
        print_board(solution, cost)
```

Output:

```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 0 0 0
Initial board configuration:
Q Q Q Q
. . . .
. . . .
. . . .
Cost: 6

Iteration 1:
Q . Q Q
. . . .
. . . .
. Q . .
Cost: 3

Iteration 2:
. . Q Q
Q . . .
. . . .
. Q . .
Cost: 1
```

```
Iteration 3:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0

Solution found with no conflicts:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0
```

2024/10/29 Tuesday

Write a program to Implement Simulated Annealing Algorithm

Function simulated-Annealing(problem, schedule) returns a solution State

Input: problem, a problem

schedule, a mapping from time to "temperature"

```
current ← MAKE-NODE(problem, INITIAL-STATE)
For t = 1 to ∞ do
    T = schedule(t)
    If T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE - current.VALUE
    If ΔE > 0 then current ← next
    else current ← next only with probability e^(ΔE/T)
```

Steps:

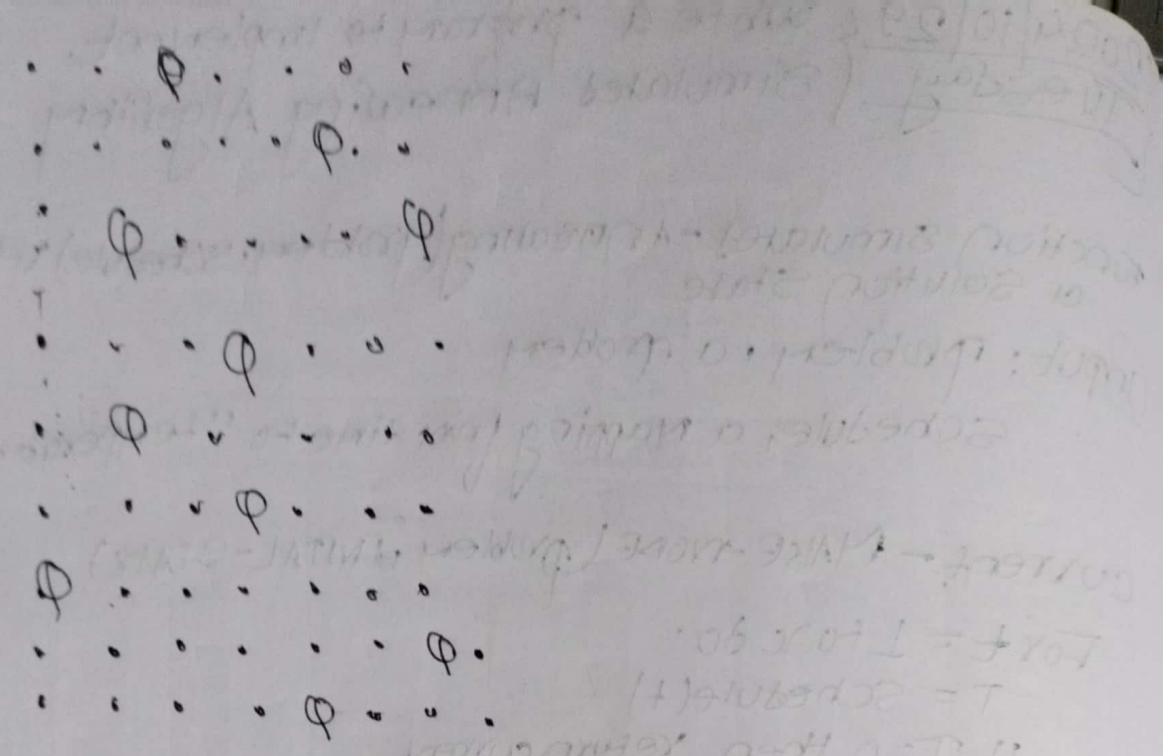
i) Start at a random point x

ii) Choose a new point x_1 on a neighborhood $N(x)$

iii) Decide whether or not to move to the new point x_1, the decision will be made based on the probability function $p(x_{rg}, T)$

$$p(x, x_g, T) = \begin{cases} 1, & F(x_g) \geq F(x) \\ e^{\frac{F(x_g) - F(x)}{x}}, & F(x_g) < F(x) \end{cases}$$

iv) Reduce T



2) Tower of Hanoi problem

Best state (final configuration): [2 2 2]

Number of correct disks on destination peg: 3

peg 0: [ ]

peg 1: [ ]

peg 2: [0, 1, 2]

29/10/24

lab:-5 Simulated Annealing Algorithm

1.Write a program to implement Simulated Annealing Algorithm

Code:

```
import mlrose_hiive as mlrose
import numpy as np


def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and
(position[j] != position[i] - (j - i)):
                no_attack_on_j += 1
        if (no_attack_on_j == len(position) - 1 - i):
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking


objective = mlrose.CustomFitness(queens_max)


problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True,
max_val=8)
T = mlrose.ExpDecay()


initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])
```

```
best_position, best_objective,fitness_curve=
mlrose.simulated_annealing(problem=problem, schedule=T,
max_attempts=500,init_state=initial_position)


print('The best position found is:', best_position)

print('The number of queens that are not attacking each other is:',
best_objective)
```

Output:

```
The best position found is: [4 1 7 0 3 6 2 5]
The number of queens that are not attacking each other is: 8.0
```

2.Tower Of Hanoi

Code:

```
import mlrose_hiive as mlrose
import numpy as np

def hanoi_fitness(state):
    correct_disks = 0
    destination_peg = 2
    for i in range(len(state)):
        if state[i] == destination_peg:
            correct_disks += 1
        else:
            break
    return correct_disks

fitness_fn = mlrose.CustomFitness(hanoi_fitness)
problem = mlrose.DiscreteOpt(length=3, fitness_fn=fitness_fn, maximize=True,
max_val=3)
schedule = mlrose.ExpDecay()

initial_state = np.array([0, 0, 0])
best_state, best_fitness, fitness_curve = mlrose.simulated_annealing(problem,
schedule=schedule, max_attempts=1000, init_state=initial_state)

print("Best state (final configuration):", best_state)
print("Number of correct disks on destination peg:", best_fitness)

def print_hanoi_solution(state):
```

```
    print("\nTower of Hanoi Configuration:")
    pegs = {0: [], 1: [], 2: []}
    for disk, peg in enumerate(state):
        pegs[peg].append(disk)
    for peg in pegs:
        print(f"Peg {peg}: {pegs[peg]}")

print_hanoi_solution(best_state)
```

Output:

```
Best state (final configuration): [2 2 2]
Number of correct disks on destination peg: 3.0

Tower of Hanoi Configuration:
Peg 0: []
Peg 1: []
Peg 2: [0, 1, 2]
```

2024/11/12 Tuesday } Lab -> 6

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

Function TT-ENTAILS?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic.
α, the query, a sentence in propositional logic.

Symbols -> a list of the proposition symbols in KB & α
return TT-CHECK-ALL(KB, α, Symbols, α))

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
if Empty?(Symbols) then
if PL-True?(KB, model) then return PL-True?(α, model)
else return true // when KB is false, always return false
else do
P ← FIRST(Symbols)
rest ← REST(Symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true}))
and
TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))

Propositional interface.

Ex: $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

Checking that $KB \models \alpha$

A	B	C	A∨C	B∨¬C	KB	α
F	F	F	F	T	False	False
F	F	T	T	F	False	False
F	T	F	F	T	False	True
F	T	T	T	T	True	True
T	F	F	T	T	True	True
T	F	T	T	F	False	True
T	T	F	T	T	True	True
T	T	T	T	T	True	True

KB entails α: True

SSB
12/11/24

Lab:-6 Propostional logic

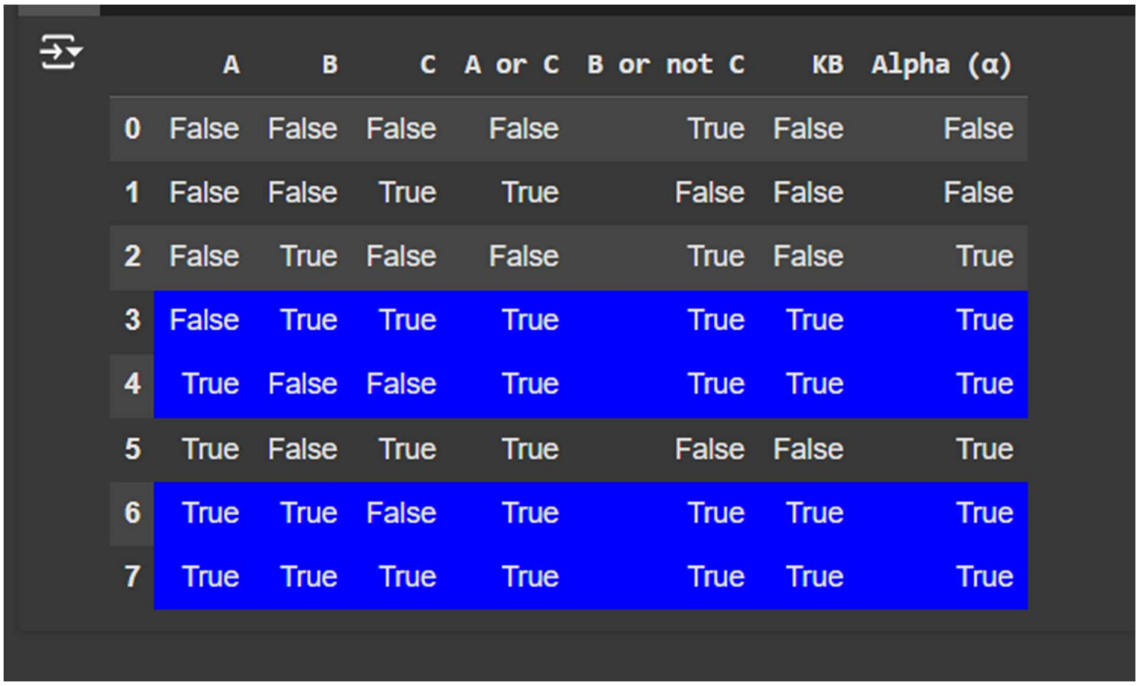
Q).Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Code:

```
import pandas as pd


# Define the truth table for all combinations of A, B, C
truth_values = [(False, False, False),
                (False, False, True),
                (False, True, False),
                (False, True, True),
                (True, False, False),
                (True, False, True),
                (True, True, False),
                (True, True, True)]


# Columns: A, B, C
table = pd.DataFrame(truth_values, columns=["A", "B", "C"])


# Calculate intermediate columns
table["A or C"] = table["A"] | table["C"]        # A ∨ C
table["B or not C"] = table["B"] | ~table["C"]    # B ∨ ¬C


# Knowledge Base (KB): (A ∨ C) ∧ (B ∨ ¬C)
table["KB"] = table["A or C"] & table["B or not C"]


# Alpha (α): A ∨ B
table["Alpha (α)"] = table["A"] | table["B"]


# Define a highlighting function
def highlight_rows(row):
    if row["KB"] and row["Alpha (α)"]:
```

```
        return ['background-color: blue'] * len(row)
    else:
        return [''] * len(row)

# Apply the highlighting function
styled_table = table.style.apply(highlight_rows, axis=1)

# Display the styled table
styled_table
```

output:



	A	B	C	A or C	B or not C	KB	Alpha (α)
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

Lab -> 7 (FOL)

2024/11/19
Tuesday

Implement unification in FOL (First order logic)

Algorithm

unify(ψ_1, ψ_2):

Step 1: if ψ_1 or ψ_2 is a variable or constant, then:

a) if ψ_1 or ψ_2 are identical, then return NIL

b) else if ψ_1 is a variable

a) then if ψ_1 occurs in ψ_2, then return FAILURE

b) else return {(ψ_2/ψ_1)}

c) else if ψ_2 is a variable

a) if ψ_2 occurs in ψ_1 then return FAILURE,

b) else return {(ψ_1/ψ_2)}

d) else return FAILURE

Step 2: if the initial predicate symbol in ψ_1 & ψ_2 are not same, then return FAILURE

Step 3: if ψ_1 & ψ_2 have a different number of arguments, then return FAILURE

Step 4: Set substitution set(SUBST) to NIL

Step 5: For i = 1 to the no. of elements in ψ_1.

a) call unify function with the ith element of ψ_1 and ith element of ψ_2, and put the result into S.

b) if S = failure then return failure

c) if S ≠ NIL then do,

a) Apply S to the remainder of both

b) SUBST = APPEND (S, SUBST)

Step6 :- Return subst.

Example :-

1) $P(x, F(y)) \rightarrow (1)$

$P(a, F(g(x))) \rightarrow (2)$

(1) & (2) are identical if x is replaced with a in eqn (1)

$P(a, F(y)) \rightarrow (1)$

replace y with g(x)

$P(a, F(g(x))) \rightarrow (1)$

=> Unification is done.

ssl

1, 5, 11, 24

Output :- [{A: x}, {B: x}, {x: x}
{A: John}
{B: John}
]

Query = {C: John}

output: {C: John} is TRUE with substitution ({C: John})

lab :-7 FOL

LAB 7.

Q).Implement Unification in first order logic.

Code:

```
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}

    # Step 1: Check if Y1 or Y2 is a variable or constant
    if Y1 == Y2:  # Identical constants or variables
        print(f"Unification Success: {Y1} and {Y2} are identical.")
        return subst
    elif is_variable(Y1):  # Y1 is a variable
        return unify_variable(Y1, Y2, subst)
    elif is_variable(Y2):  # Y2 is a variable
        return unify_variable(Y2, Y1, subst)

    # Step 2: Predicate symbols not the same
    if predicate_symbol(Y1) != predicate_symbol(Y2):
        print(f"Unification Failure: Predicate symbols {predicate_symbol(Y1)} and
{predicate_symbol(Y2)} don't match.")
        return None  # FAILURE

    # Step 3: Different number of arguments
    args1, args2 = arguments(Y1), arguments(Y2)
    if len(args1) != len(args2):
        print(f"Unification Failure: Different number of arguments in {Y1} and {Y2}.")
        return None  # FAILURE

    # Step 5: Recursively unify each element in the lists
    for a1, a2 in zip(args1, args2):
        subst = unify(a1, a2, subst)
```

```
        if subst is None:
            return None  # FAILURE

    # Step 6: Return SUBST (final substitution set)
    print(f"Unification Success: {Y1} and {Y2} unified with substitution {subst}.")
    return subst

def unify_variable(var, x, subst):
    if var in subst:
        print(f"Unification Success: Variable {var} is already in the substitution.")
        return unify(subst[var], x, subst)
    elif occurs_in(var, x):
        print(f"Unification Failure: Variable {var} occurs in {x} (circular reference).")
        return None  # FAILURE due to circular reference
    else:
        print(f"Unification Success: Substituting {var} with {x}.")
        subst[var] = x
        return subst

def predicate_symbol(expr):
    return expr[0] if isinstance(expr, list) else expr

def arguments(expr):
    return expr[1:] if isinstance(expr, list) else []

def is_variable(x):
    return isinstance(x, str) and x.islower()

def occurs_in(var, x):
    if var == x:
        return True
```

```
    elif isinstance(x, list):
        return any(occurs_in(var, xi) for xi in x)
    return False

# Example usage: Replace Y1 and Y2 with p(x, f(y)) and p(a, f(g(x)))
Y1 = ['p', 'x', ['f', 'y']]      # p(x, f(y))
Y2 = ['p', 'A', ['f', ['g', 'x']]]  # p(a, f(g(x)))
subst = unify(Y1, Y2)

if subst:
    print("Final Substitution:", subst,"Unification Successful")
else:
    print("Unification failed.")
```

Output:

```
Unification Success: Substituting x with A.
Unification Success: Substituting y with ['g', 'x'].
Unification Success: ['f', 'y'] and ['f', ['g', 'x']] unified with substitution {'x': 'A', 'y': ['g', 'x']}.
Unification Success: ['p', 'x', ['f', 'y']] and ['p', 'A', ['f', ['g', 'x']]] unified with substitution {'x': 'A', 'y': ['g', 'x']}.
Final Substitution: {'x': 'A', 'y': ['g', 'x']} Unification Successful
```

Lab-8

Forward Reasoning Algorithm

11/26 Tuesday

Function FOL-FC-ASE(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 variable: new, the new sentences inferred on each iteration

repeat until new is empty
  new ← { }
  for each rule in KB do
    (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(rule)
    for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
      for some p'1, ..., 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
        φ ← UNIFY(q', α)
        if φ is not fail then return φ
  add new to KB
return false

Ex-> parent(John, Mary)
parent(Mary, Anna)

2nd ⇒ parent(x, y) ∧ parent(y, z) ⇒ grandparent(x, z)

Ex-> Human(Socrates)
∀x(Human(x) ⇒ Mortal(x))

lab :-8 Forward reasoning algorithm

Q). Create a knowledge base consisting of first order logic statements and

prove the given query using forward reasoning.

Code:

```
facts = {
    "American(Robert)": True,
    "Missile(T1)": True,
    "Enemy(A, America)": True,
    "Owns(A, T1)": True,
    "Hostile(A)": False,
    "Weapon(T1)": False,
    "Sells(Robert, T1, A)": False,
    "Criminal(Robert)": False,
}

rules = [
    ("American(Robert) and Weapon(T1) and Sells(Robert, T1, A) and Hostile(A)", "Criminal(Robert)"),
    ("Owns(A, T1) and Missile(T1)", "Weapon(T1)"),
    ("Missile(T1) and Owns(A, T1)", "Sells(Robert, T1, A)"),
    ("Enemy(A, America)", "Hostile(A)"),
]

def check_fact(fact):
    return facts.get(fact, False)

def parse_condition(condition):
    return condition.split(" and ")

def forward_reasoning():
    new_inferences = True
    while new_inferences:
        new_inferences = False
```

```
        for condition, conclusion in rules:
            condition_facts = parse_condition(condition)
            if all(check_fact(fact) for fact in condition_facts):
                if not check_fact(conclusion):
                    facts[conclusion] = True
                    new_inferences = True
                    print(f"Inferred: {conclusion}")

def print_inferred_facts():
    forward_reasoning()
    print("\nFinal Inferred Facts:")
    for fact, value in facts.items():
        print(f"{fact} is {'TRUE' if value else 'FALSE'}")

print_inferred_facts()
```

Output:

Inferred: Weapon(T1)

Inferred: Sells(Robert, T1, A)

Inferred: Hostile(A)

Inferred: Criminal(Robert)

Final Inferred Facts:

American(Robert) is TRUE

Missile(T1) is TRUE

Enemy(A, America) is TRUE

Owns(A, T1) is TRUE

Hostile(A) is TRUE

Weapon(T1) is TRUE

Sells(Robert, T1, A) is TRUE

Criminal(Robert) is TRUE

2024/11/28
Tuesday

Lab 9

Convert FOL to Resolution

Algorithm

1) convert all sentence to CNF
2) Negate conclusion S and convert result to CNF
3) Add negated conclusion S to the premise clauses.
4) Repeat until contradiction or no progress is made
   a) Select two clauses (call them parent clauses)
   b) Resolve them together, performing all required unification
   c) If resolution is the empty clause, a contradiction has been found (i.e. S follows from the premises)
   d) If not, add resolvent to the premises

If we succeed in Step 4, we have proved the conclusion.

Proof by Resolution

a) 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 peanuts and still alive

e. Harry eats everything that Anil eats.

In FOL.

a -> $\forall x$: Food(x) -> likes(John, x)

b -> food(apple) $\wedge$ Food(vegetable)

c -> $\forall x \forall y$: eats(x, y) $\wedge$ ~killed(x) -> food(y)

d -> eats(Anil, Peanuts) $\wedge$ alive(Anil)

e -> $\forall x$: eats(Anil, x) -> eats(Harry, x)

Eliminate Implicate

a) $\forall x: Food(x) \rightarrow likes(John, x)$

$\Rightarrow \forall x - food(x) \vee likes(John, x)$

b) $food(Apple) \wedge food(vegetable)$

$\Rightarrow food(Apple) \wedge food(vegetable)$

c) $\forall x \forall y\ eats(x, y) \wedge \neg killed(x) \Rightarrow food(y)$

$\rightarrow \forall x \forall y - (eats(x, y) - killed(x)) \vee food(y)$

d) $eats(Anil, peanuts) \wedge alive(Anil)$

$\Rightarrow eats(Anil, peanuts) \wedge alive(Anil)$

e) $\forall x: eats(Anil, x) \Rightarrow eats(Harry, x)$

$\rightarrow \forall x \rightarrow eats(Anil, x) \vee eats(Harry, x)$

8/8

3 | 12 | 24

Lab-09 Resolution in First-

Order Logic

CODE:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf


# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple Vegetables Peanuts x y')


# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')


# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),


    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),


    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),
```

```
    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
```

```
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()

    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents: # Empty clause found
                    return True # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses): # No new information
            return False # No contradiction; proof failed

        clauses.update(new_clauses)
```

```
# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be proven.")
```

OUTPUT:

```
Does John like peanuts?  Yes, proven by resolution.
```

Alpha Beta Pruning Algorithm

Alpha(α) - Beta(β) proposes to the find 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

⇒ Alpha(α) - Beta(β) gives optimal solution & it takes less time to get the value from the root node.

Proof by Resolution

a) Given the KBs 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 peanuts and still alive

e. Harry eats everything that Anil eats

In FOL.

a → ∀x: Food(x) → likes(John, x)

b → food(apple) ∧ Food(vegetable)

c → ∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)

d → eats(Anil, Peanuts) ∧ alive(Anil)

e → ∀x: eats(Anil, x) → eats(Harry, x)

Eliminate / Implicate

a) $\forall x$: Food(x) → likes(John, x)
→ $\forall x$ - food(x) ∨ likes(John, x)

b) food(Apple) ∧ food(vegetable)
→ food(Apple) ∧ food(vegetable)

c) $\forall x \forall y$ eats(x, y) ∧ - killed(x) → food(y)
→ $\forall x \forall y$ - (eats(x, y) - killed(x)) ∨ food(y)

d) eats(Anil, peanuts) ∧ alive(Anil)
→ eats(Anil, peanuts) ∧ alive(Anil)

e) $\forall x$: eats(Anil, x) → eats(Harry, x)
→ $\forall x$ → eats(Anil, x) ∨ eats(Harry, x)

8 8 8
3 | 12 | 24

LAB-10 Alpha Beta Pruning

CODE:

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning with detailed step output

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for the current player
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    # Terminating condition: leaf node is reached
    if depth == 3:
        print(f"Leaf node reached: Depth={depth}, NodeIndex={nodeIndex},
Value={values[nodeIndex]}")
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        print(f"Maximizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")

        # Recur for left and right children
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
```

```
            best = max(best, val)
            alpha = max(alpha, best)

            print(f"Maximizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best}, Alpha={alpha}, Beta={beta}")

            # Alpha Beta Pruning
            if beta <= alpha:
                print(f"Maximizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")
                break
        return best
    else:
        best = MAX
        print(f"Minimizer: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")

        # Recur for left and right children
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            print(f"Minimizer updated: Depth={depth}, NodeIndex={nodeIndex}, Best={best}, Alpha={alpha}, Beta={beta}")

            # Alpha Beta Pruning
            if beta <= alpha:
                print(f"Minimizer Pruned: Depth={depth}, NodeIndex={nodeIndex}, Alpha={alpha}, Beta={beta}")
                break
        return best
```

```
# Driver Code
if _name_ == "_main_":
    values = [3, 5, 6, 9, 1, 2, 0, -1] # Leaf node values
    print("Starting Alpha-Beta Pruning...")
    optimal_value = minimax(0, 0, True, values, MIN, MAX)
    print(f"\nThe optimal value is: {optimal_value}")
```

OUTPUT:

```
Starting Alpha-Beta Pruning...
Maximizer: Depth=0, NodeIndex=0, Alpha=-1000, Beta=1000
Minimizer: Depth=1, NodeIndex=0, Alpha=-1000, Beta=1000
Maximizer: Depth=2, NodeIndex=0, Alpha=-1000, Beta=1000
Leaf node reached: Depth=3, NodeIndex=0, Value=3
Maximizer updated: Depth=2, NodeIndex=0, Best=3, Alpha=3, Beta=1000
Leaf node reached: Depth=3, NodeIndex=1, Value=5
Maximizer updated: Depth=2, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer: Depth=2, NodeIndex=1, Alpha=-1000, Beta=5
Leaf node reached: Depth=3, NodeIndex=2, Value=6
Maximizer updated: Depth=2, NodeIndex=1, Best=6, Alpha=6, Beta=5
Maximizer Pruned: Depth=2, NodeIndex=1, Alpha=6, Beta=5
Minimizer updated: Depth=1, NodeIndex=0, Best=5, Alpha=-1000, Beta=5
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000
Minimizer: Depth=1, NodeIndex=1, Alpha=5, Beta=1000
Maximizer: Depth=2, NodeIndex=2, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=4, Value=1
Maximizer updated: Depth=2, NodeIndex=2, Best=1, Alpha=5, Beta=1000
Leaf node reached: Depth=3, NodeIndex=5, Value=2
Maximizer updated: Depth=2, NodeIndex=2, Best=2, Alpha=5, Beta=1000
Minimizer updated: Depth=1, NodeIndex=1, Best=2, Alpha=5, Beta=2
Minimizer Pruned: Depth=1, NodeIndex=1, Alpha=5, Beta=2
Maximizer updated: Depth=0, NodeIndex=0, Best=5, Alpha=5, Beta=1000

The optimal value is: 5
```