VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sirigireddy Pranav Reddy (1BM22CS281)

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 **Sirigireddy Pranav Reddy(1BM22CS281),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

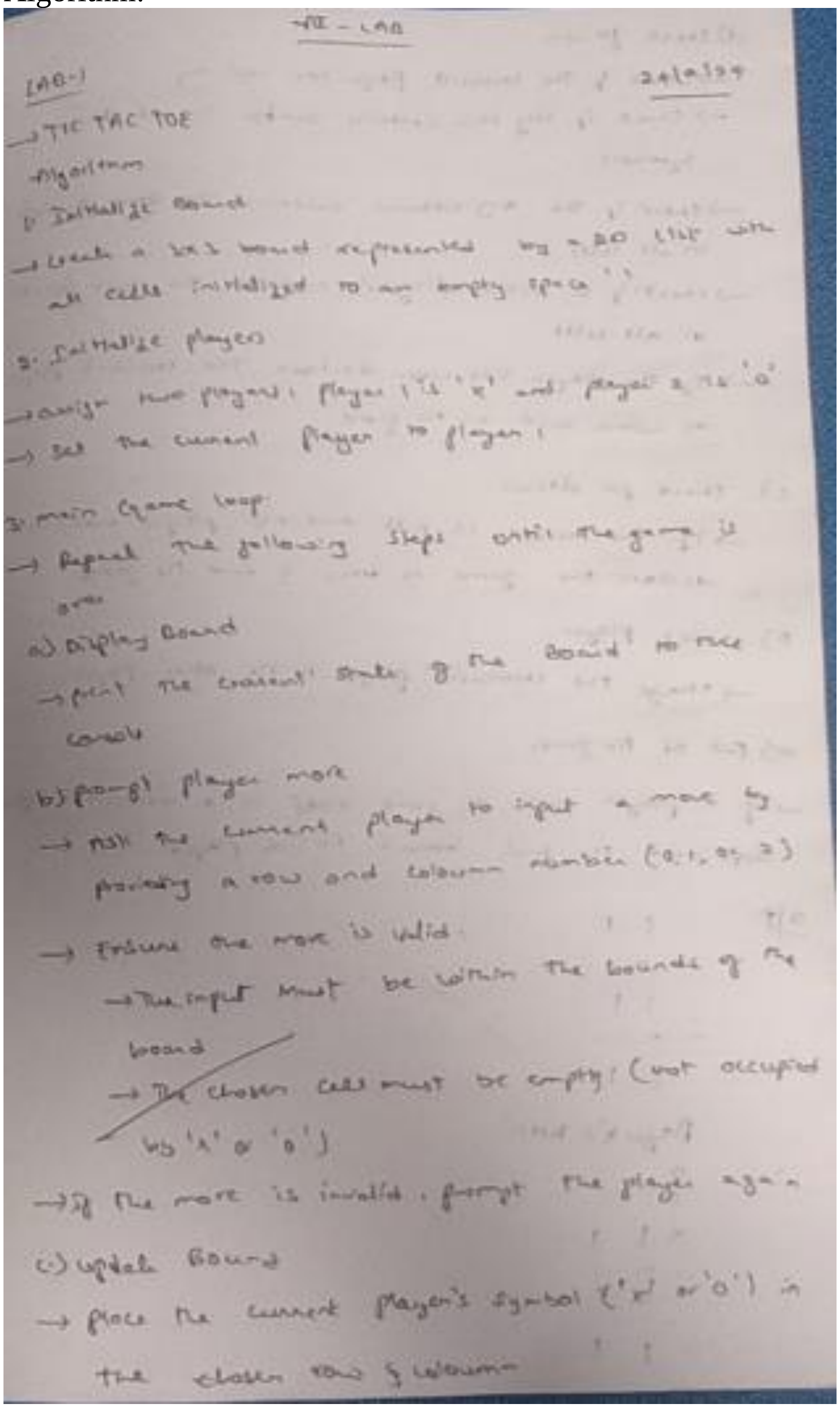
Prof Sunayana Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	1
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	5
3	14-10-2024	Implement A* search algorithm	11
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	17
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	24
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	30
7	2-12-2024	Implement unification in first order logic	38
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	43
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	50
10	16-12-2024	Implement Alpha-Beta Pruning.	57

Github Link:
(You should provide your github link which contains all ten AI lab programs)

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent
Tic-Tac-Toe
Algorithm:


AI - LAB

LAB-1 24/9/24

→ TIC TAC TOE

Algorithm

1. Initialize Board
→ Create a 3x3 board represented by a 2D list with all cells initialized to an empty space ' '

2. Initialize players
→ Assign two players: player 1 is 'X' and player 2 is 'O'
→ Set the current player to player 1

3. Main Game loop:
→ Repeat the following steps until the game is over

a) Display Board
→ print the current state of the board to the console

b) Prompt player move
→ Ask the current player to input a move by providing a row and column number (0, 1, or 2)
→ Ensure the move is valid
→ The input must be within the bounds of the board
→ The chosen cell must be empty (not occupied by 'X' or 'O')
→ If the move is invalid, prompt the player again

c) Update Board
→ Place the current player's symbol ('X' or 'O') in the chosen row & column

d) Check for win

→ Check if the current player has won by

→ Check if any row contains same symbol in all the symbols

→ Check if the any column contains the same symbol in all cells

→ Check if either diagonal contains the same symbol in all cells

→ If the player has won declare the current player as winner and end the game

e) Check for draw

→ If the board is full and non player has won declare the game as draw & end the game

f) Switch player

→ change the current player to the other player

g) End of the game.

→ If a player wins or game ends in a draw the game stops and the final board is displayed

O/P

```
 | | 
-----
 | | 
-----
 | | 
```

Player X's turn

0

0

```
X| | 
-----
 | | 
-----
 | | 
```

Player O's turn

1
0

X | |
- - - -
O | |
- - - -
 | |

Player X's turn

1
1

X | |
- - - -
O | X |
- - - -
 | |

Player O's turn

2
0

X | |
- - - -
O | X |
- - - -
O | |

Player ~~X's turn~~

2
2

X | |
- - - -
O | X |
- - - -
O | | X

Player X wins

Code:

```
def check_win(board, r, c):
    if board[r - 1][c - 1] == 'X':
        ch = "O"
    else:
        ch = "X"
    if ch not in board[r - 1] and '-' not in board[r - 1]:
        return True
    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1],
board[1][c - 1], board[2][c - 1]):
        return True
    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1],
board[2][2]):
        return True
    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1],
board[2][0]):
        return True
    return False

def displayb(board):
  print(board[0])
  print(board[1])
  print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]
displayb(board)
xo=1
flag=0
while '-' in board[0] or '-' in board[1] or '-' in board[2]:

  if xo==1:
    print("enter position to place X:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='X'
      xo=0
      displayb(board)
    else:
      print("invalid position")
      continue
    if(check_win(board,x,y)):
        print("X wins")
        flag=1
```

```
        break
  else :
    print("enter position to place O:")
    x=int(input())
    y=int(input())
    if(x>3 or y>3):
      print("invalid position")
      continue
    if(board[x-1][y-1]=='-'):
      board[x-1][y-1]='O'
      xo=1
      displayb(board)
    else:
      print("invalid position")
      continue
    if(check_win(board,x,y)):
        print("0 wins")
        flag=1
        break
if flag==0:
  print("Draw")
print("Game Over")
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
1
2
['X', 'O', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
2
1
['X', 'O', '-']
['X', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', 'O', '-']
['X', 'O', '-']
['-', '-', '-']
enter position to place X:
3
1
['X', 'O', '-']
['X', 'O', '-']
['X', '-', '-']
X wins
Game Over
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', '-', '-']
['-', 'O', '-']
['-', '-', '-']
enter position to place X:
3
3
['X', '-', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place O:
1
2
['X', 'O', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place X:
3
2
['X', 'O', '-']
['-', 'O', '-']
['-', 'X', 'X']
enter position to place O:
3
1
['X', 'O', '-']
['-', 'O', '-']
['O', 'X', 'X']
enter position to place X:
2
1
['X', 'O', '-']
['X', 'O', '-']
['O', 'X', 'X']
enter position to place O:
2
3
['X', 'O', '-']
['X', 'O', 'O']
['O', 'X', 'X']
enter position to place X:
1
3
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', 'X']
Draw
Game Over
```

Vacuum Cleaner
Algorithm:

?/10/23

Lab-2

→ Implement function Reflex vacuum agent ([location, status])

function Reflex-vacuum-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

→ Two quadrants

Step1: def display_goal_state (state):

        Print(f"current Goal State: [A: {state[1]}, B: {state[3]}]")

Step2: def display_message (message):

        Print (message)

Step3: def vaccum_world_two_quadrant ():

        goal_state = ['A', 0, 'B', 0]

        current_state = ['A', 0, 'B', 0]

        current_state [1] = int (input ("Enter status for location A (0: No Dust, 1: Dust): "))

        current_state[3] = int (input ("Enter status for location B (0: No Dust, 1: Dust): "))

        display_goal_state (current_state)

        cost = 0

        location = 'A'

```
while current_state != goal_state:
    if location == 'A':
        display_message("vacuum is placed in location A")
        if current_state[0] == 1:
            display_message("location A is Dirty")
            display_message("cleaning location A...")
            current_state[0] = 0
            display_message("location A has been cleaned")
            cost += 1
            display_message(f"cost for suck at location A
                                          {cost}")
Step 4.  else:
            display_message("location A is already clean")
        display_message("moving right to location B...")
        location = 'B'
    elif location == 'B':
        display_message("vacuum is placed in location B")
        if current_state[1] == 1:
            display_message("location B is Dirty")
            display_message("cleaning location B...")
            current_state[1] = 0
            display_message("location B has been cleaned")
            cost += 1
            display_message(f"cost for suck at location B:
                                          {cost}")
        else:
            display_message("location B is already clean")
```

Enter status for location A (0: no Dust, 1: Dust): 0
Enter status for location B (0: no Dust, 1: Dust): 0
current goal state: [A: 0, B: 0]
Goal state reached: Both rooms are clean
current goal state: [A: 0, C: 0]

→ four quadrant:

Step: 1 Initialize the environment

(i) take input for the dust status of rooms A, B, C

Step-2: Set the goal state

(i) Goal state {'A': 0, 'B': 0, 'C': 0, 'D': 0}

(ii) Start from room A

Step-3: loop until goal state is achieved

(i) while current state ≠ goal-state

* if vaccum is at room A:

* clean if dirty, then move to B

* If vaccum is at room B:

* clean if dirty, then move to C

* if vaccum is at room C:

* clean if dirty then move to D.

* if vaccum is at room D:

* clean if dirty, then move back to A

Step-4: Check if goal state is reached

Step-5: When all rooms are clean the process ends

O/P

Enter Status for location A (0: No Dust, 1: Dust) 1

Enter Status for location B (0: No Dust, 1: Dust) 0

Enter Status for location C (0: No Dust, 1: Dust) 0

Enter Status for location D (0: No Dust, 1: Dust) 0

Current Goal State : [A:1, B:0, C:0, D:1]

vaccum is placed in location A

location A is Dirty

cleaning location A:1

moving right to location B.

Current Goal State : [A:0, B:0, C:0, D:0]

Goal State reached : All rooms are clean

Currently Goal State : [A:0, B:0, C:0, D:0]

10/2024

Enter status for location A (0: No Dust, 1: Dust) 0

Enter Status for location B (0: No Dust, 1: Dust) 0

Enter Status for location C (0: No Dust, 1: Dust) 0

Enter Status for location D (0: No Dust 1: Dust) 0

Current Goal State : [A:0, B:0, C:0, D:0]

Goal State reached : All rooms are clean

Current State reached

Current Goal State : [A:0, B:0, C:0, D:0]

Code:

```
count = 0
def rec(state, loc):
    global count
    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        return

    if state[loc] == 1:
        state[loc] = 0
        count += 1
        print(f"Cleaned {loc}.")
        next_loc = 'B' if loc == 'A' else 'A'
        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
        if(state[next_loc]!=1):
          state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
    if(state[loc]==1):
       rec(state,loc)
    else:
      next_loc = 'B' if loc == 'A' else 'A'
      dire="left" if loc=="B" else "right"
      print(loc,"is clean")
      print(f"Moving vacuum {dire}")
      if state[next_loc] == 1:
          rec(state, next_loc)

state = {}
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
loc = input("Enter location (A or B): ")
rec(state, loc)
print("Cost:",count)
print(state)
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
Is B dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```

Program 2

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

8 puzzle using DFS

Algorithm:

LAB-3

→ 8 puzzle problem using BFS & DFS algorithm

BFS

1) Initialize
   → create a queue and add the initial state
   → create a set to track visited states
2) while the queue is not empty
   → Dequeue the first state
   → print the state and the move direction
   → if it's the goal state, return the solution
   → otherwise generate all valid neighboring states
   → For each neighbor : if not visited, add to the queue and mark as visited
3) Returns "NO solution" if the queue is empty and goal not found

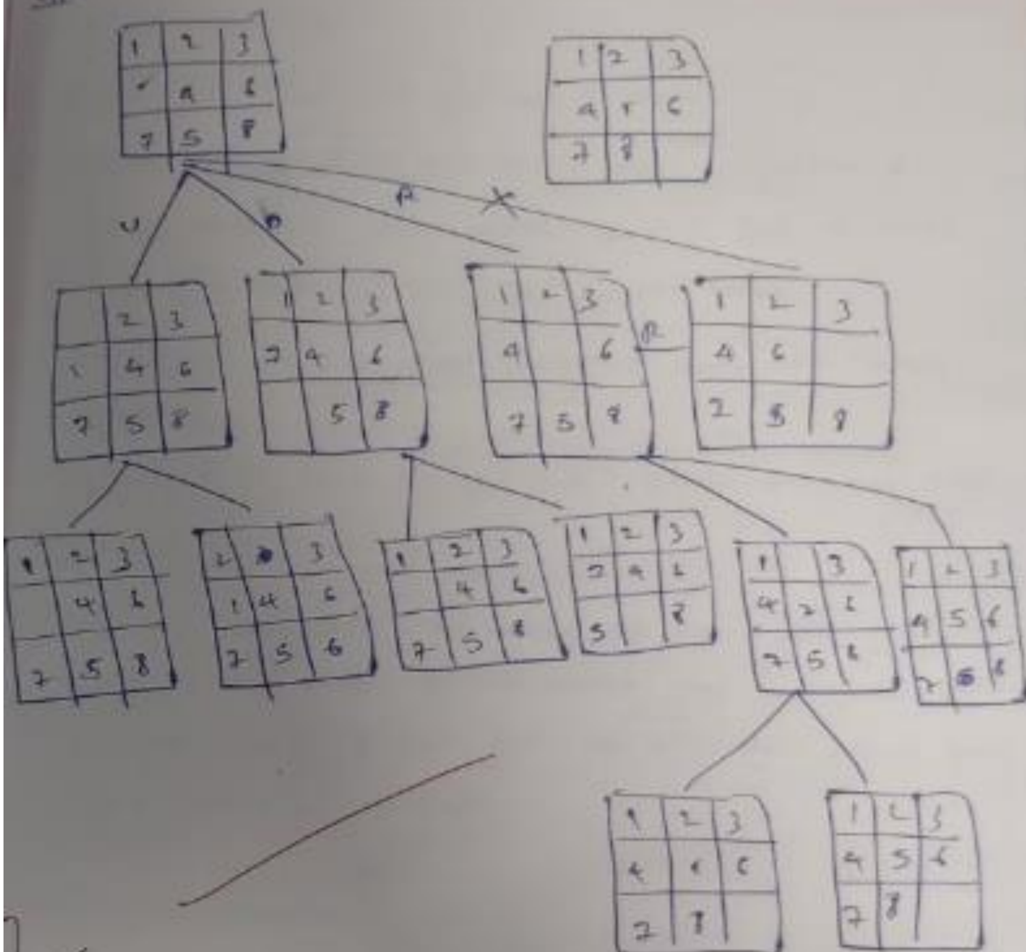
DFS

1) Initialize
   → create a stack and add the initial state
   → create a set to track visited states
   → Set a depth limit
2) while the stack is not empty
   → Pop the top state.
   → print the state and move direction
   → If it's the goal state, return the solution
   → If depth is within the limit, generate all valid neighboring states
   → For each neighbor :
     if not visited, push onto the stack and mark as

visited

3) Return "No Solution" if the stack is empty and goal not found

O/P:



8/10/2024

Code:

```
def dfs(initial_board, zero_pos):
    stack = [(initial_board, zero_pos, [])]
    visited = set()

    while stack:
        current_board, zero_pos, moves = stack.pop()

        if is_goal(current_board):
            return moves, len(moves)  # Return moves and their count

        visited.add(tuple(current_board))

        for neighbor_board, neighbor_pos in get_neighbors(current_board, zero_pos):
            if tuple(neighbor_board) not in visited:
                stack.append((neighbor_board, neighbor_pos, moves + [neighbor_board]))

    return None, 0  # No solution found, return count as 0

# Initial state of the puzzle
initial_board = [1, 2, 3, 0, 4, 6, 7, 5, 8]
zero_position = (1, 0)  # Position of the empty tile (0)

# Solve the puzzle using DFS
solution, move_count = dfs(initial_board, zero_position)

if solution:
    print("Solution found with moves ({} moves):".format(move_count))
    for move in solution:
        print_board(move)
        print()  # Print an empty line between moves
else:
    print("No solution found.")
```

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

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

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

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

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

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

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

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

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

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

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

Implement Iterative deepening search algorithm

Algorithm:

Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos  # Position of the zero tile
        self.moves = moves         # Number of moves taken to reach this state
        self.previous = previous    # For tracking the path

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

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves

def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None

def dls(state, goal_state, depth, visited):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None

    visited.add(tuple(map(tuple, state.board)))  # Mark this state as visited
```

```
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
        if tuple(map(tuple, new_board)) not in visited:
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board)))  # Unmark this state
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)

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

# Perform Iterative Deepening Search
max_depth = 20  # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

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

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

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

Program 3

Implement A* search algorithm

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

LAB-4 15-10-2024

→ 8 puzzle using A* implementation

→ A* misplaced tiles:

(i) Define the initial and goal state

(ii) Define list & $f(n) = g(n) + h(n)$ where $g(n) = 0$ & $h(n)$ = misplaced tiles, where $g(n)$ is cost to reach the each node & $h(n)$ is heuristic value

(iii) while list is not empty, extract node with lowest frequency.

If current state is goal state return path and add the current state to closed set

(iv) Generate new state by moving up, down, left and right. If new state is not closed set calculate $g(n)$ for new state & $h(n)$, $f(n)$ continue the process until goal state is reached

→ Manhattan Distance:

(i) Define goal state & also create a dictionary

(ii) Create a list containing $f(n)$, $g(n)$ and path with $g(n) = 0$

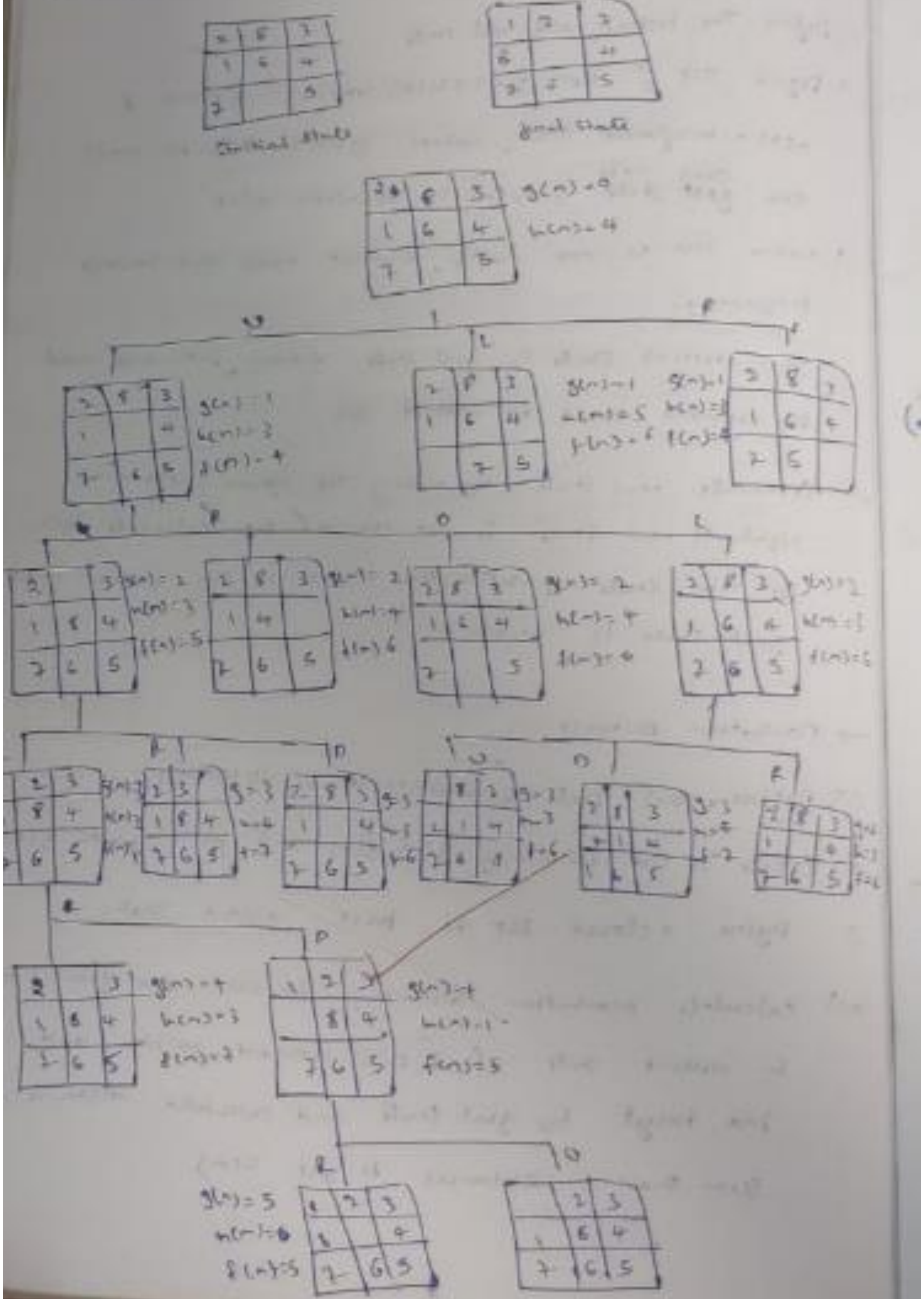
Define a closed set to track visited states.

(iii) Calculate manhattan distance for each tile from 1 to 8 in current state find this current position and find target by goal state and calculate distance

Sum these distances to get $h(n)$

(iv) Expand nodes, while list is ...
node with low frequency and return path

State space tree diagram



O/P (missing tiles)

Enter the start state: 2 8 3 1 6 4 7 0 5

No. of states visited: 12

Solution found at depth: 5 with cost: 5

Sol steps)

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

(manhattan approach)

Enter the start state: 2 8 3 1 6 4 7 0 5

No. of states visited: 34

Solution found at depth: 5 with cost: 5

Sol steps)

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

15/10/2024

Code:
Misplaced Tiles

```
def mistil(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j]:
                count += 1
    return count

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = mistil(state['state'], goal)
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
```

```
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")
    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
```

print_state(goal_state)

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 3, f(n): 4
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: up

Current state:
2 8 3
0 1 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: left

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 3, f(n): 6
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 2, f(n): 6
Move: down
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Manhattan Distance

```
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:  # Ignore the blank space (0)
                # Find the position of the tile in the goal state
                for r in range(3):
                    for c in range(3):
                        if goal[r][c] == tile:
                            target_row, target_col = r, c
                            break
                # Add the Manhattan distance (absolute difference in rows and columns)
                distance += abs(target_row - i) + abs(target_col - j)
    return distance

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = manhattan_distance(state['state'], goal)  # Use Manhattan distance here
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

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

# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)

    print("Current state:")
    print_state(best_state['state'])

    h = manhattan_distance(best_state['state'], goal_state)  # Using Manhattan distance here
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")

    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if h == 0:  # Goal is reached if h == 0
        goal_state_reached = best_state
        break

    visited_states.append(best_state['state'])
```

```
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down
```

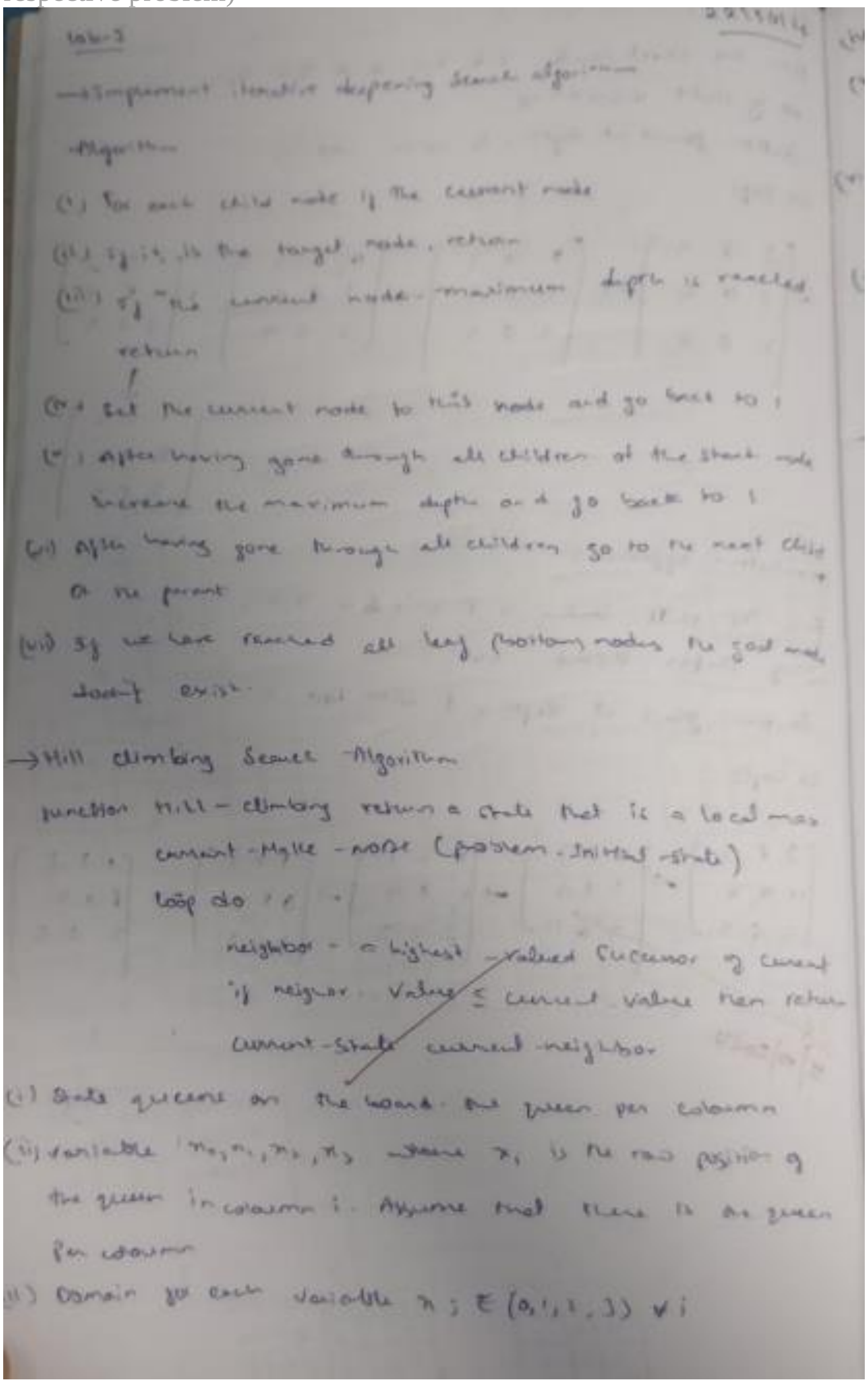
```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)



Lab-3

→ Implement iterative deepening search algorithm

Algorithm

(i) For each child node of the current node

(ii) If it is the target node, return

(iii) If the current node maximum depth is reached return

(iv) Set the current node to this node and go back to 1

(v) After having gone through all children of the start node increase the maximum depth and go back to 1

(vi) After having gone through all children go to the next child of the parent

(vii) If we have reached all leaf (bottom) nodes the goal node doesn't exist.

→ Hill climbing Search Algorithm

function Hill-climbing return a state that is a local max

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

(i) State queens on the board. One queen per column

(ii) variable x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column

(iii) Domain for each variable $x_i \in (0, 1, 2, 3)$ ∀ i

(iii) Initial State: a random state

(iv) Goal State: 4 queens on the board, no pair of queens are attacking each other

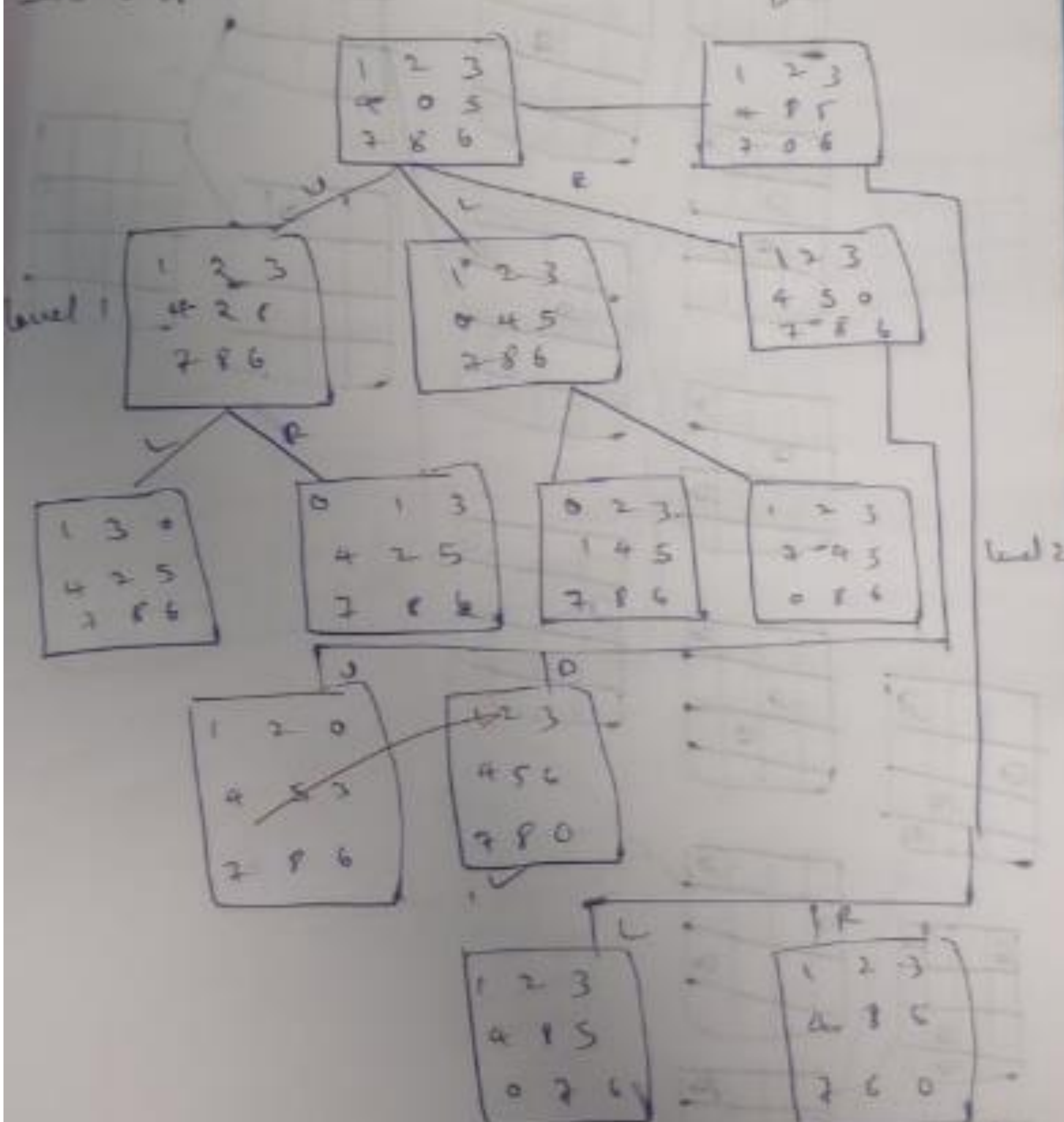
(v) Neighbour relation:

Swap the row of two queens

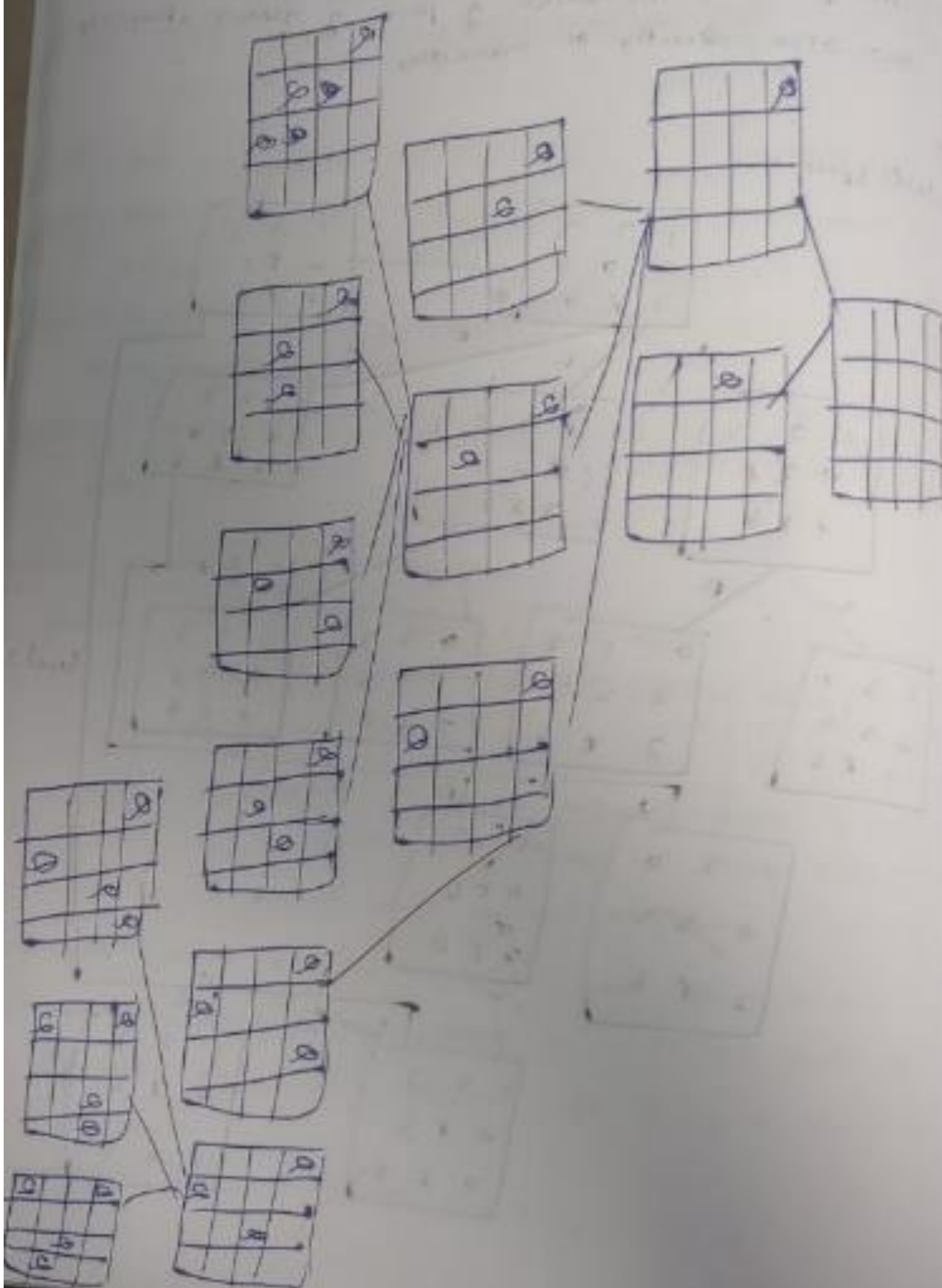
(vi) Cost function: the number of pairs of queens attacking each other directly or indirectly

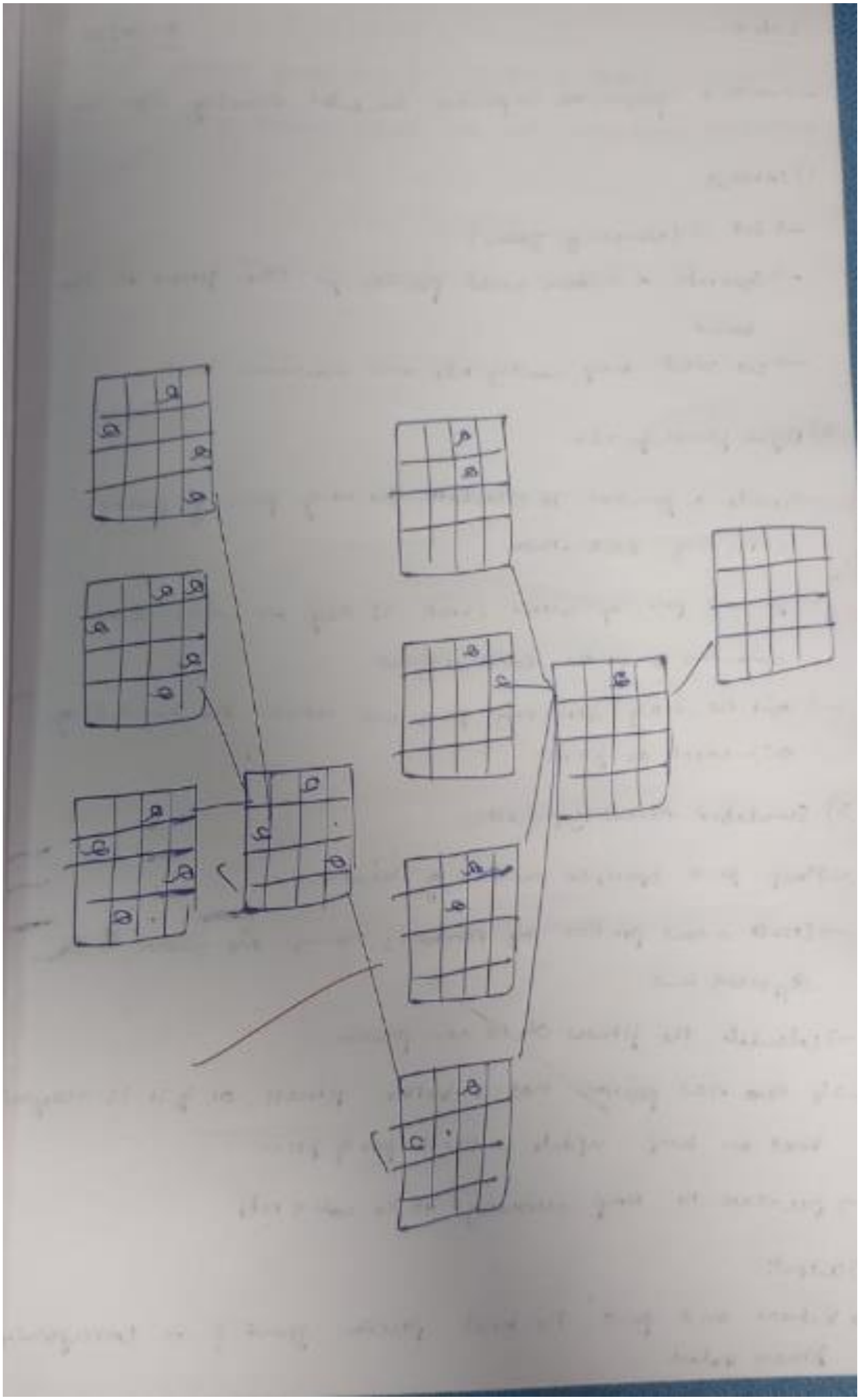
2)

→ State space tree



2)
→ State space tree





Code:

```
import random

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]:  # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print(current_conflicts)
                            print_board(neighbor_board)
                            print(neighbor_conflicts)
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost+=1
                            found_better = True
                            break
                if found_better:
                    break

            # If no better neighbor found, stop searching
            if not found_better:
                break

        # If a solution is found (zero conflicts), return the board
```

```
        if current_conflicts == 0:
            return current_board, current_conflicts, cost

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'  # Place a queen
        print(' '.join(row))
    print()
print("===============")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

```
===============
Q . . .
. . . Q
. . Q .
. Q . .

4
Q . . .
Q . . .
. . Q .
. Q . .

3
Q . . .
Q . . .
. . Q .
. Q . .

3
. . Q .
Q . . .
. . Q .
. Q . .

2
. . Q .
Q . . .
. . Q .
. Q . .

2
. . . Q
Q . . .
. . Q .
. Q . .

1
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

Lab-6

→ write a program to implement Simulated Annealing Algorithm

1) Initialize

→ Set n (number of queens)

→ Generate a random initial position for the queens on the board

→ set initial temp, cooling rate and maximum iterations

2) Define fitness function:

→ Create a function to calculate the no. of pairs of queens attacking each other.

→ for each pair of queens, check if they are in the same column ~~or~~ on the same diagonal

→ count the no of attacking pairs and return the negative of this count as fitness.

3) Simulated Annealing process:

→ loop for a specified number of iterations:

→ Create a new position by randomly moving one queen to a different row.

→ Calculate the fitness of the new position

→ If the new position has a better fitness, or if it is accep

based on temp, update current pos & fitness

→ Decrease the temp according to the cooling rate.

4) Output:

→ Return and print the best position found & its correspo

fitness value

o/p

- The best position found is : [1 6 4 7 0 3 5 2]

The number of queens that are not attacking each other is 28

29/10/2024

Code:

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 8) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

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

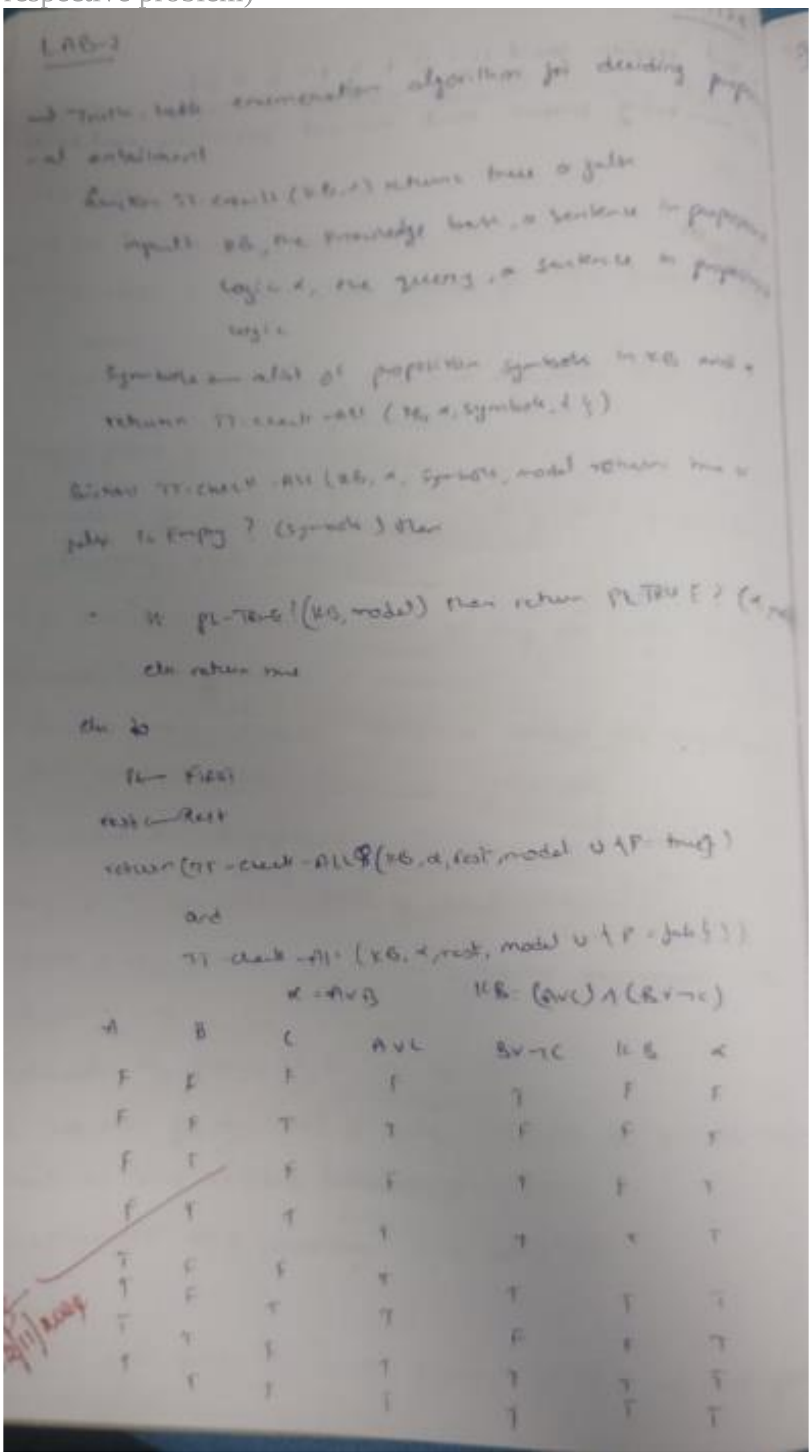
```
The best position found is: [0 8 5 2 6 3 7 4]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)



O/P

```
from itertools import product

def evaluate(expression, model):
    if isinstance(expression, str):
        return model[expression]
    elif isinstance(expression, tuple):
        op, *args = expression
        if op == 'not':
            return
```

O/P: Model: {'P': True, 'Q': True}

KB is True, Query is True

Model: {'P': True, 'Q': False}

KB is False, Query is False

Model: {'P': False, 'Q': True}

KB is False, Query is True

KB entails the query

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

```
import itertools

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

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

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

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

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

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

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

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

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

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

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

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

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

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

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

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

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

Program 7

Implement unification in first order logic

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

LAB-8

→ Unification 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 $\{(\Psi_2/\Psi_1)\}$

c) Else if Ψ_2 is a variable

a. If Ψ_2 occurs in Ψ_1 then return Failure

b. Else return $\{(\Psi_1/\Psi_2)\}$

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 and 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 number of elements in Ψ_1

a) Call unify function with the ith element of Ψ_1 & ith element of Ψ_2 & put the result into S

b) If S = failure then returns Failure

c) If S ≠ NIL then do

a. Apply S to the remainder of both L1 & L2

b. SUBST = APPEND (S, SUBST).

Step 5: Return SUBST

Ex: $\Psi_1 = P(b, x, f(g(z)))$

$\Psi_2 = P(z, f(y), f(y))$

$\{ b : z,\ f(y) : x,\ f(g(z)) : f(y) \}$

$\Psi_1 = P(f(x), g(y))$

$\Psi_2 = P(x, x)$

Unification algorithm failed

Substitution $(f(x))(x)$

$P(f(x), g(y))$ and $P(f(x), f(x))$

19/11/24

Code:

```
import re

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

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

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

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

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

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

def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

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

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

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

    return parse_term(input_str)
```

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

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

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

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

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

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

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z)))']]
Expression 2: ['p', '(z', ['f', '(y)'], ['f', '(y))']]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)'], '(g(z)))': '(y))'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y))']]
Expression 2: ['p', '(a', ['f', '(z))']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y))']]
Expression 2: ['p', '(x', 'x)']
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y))']}
Do you want to test another pair of expressions? (yes/no): no
```

Program 8

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

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

26/11/24

Lab-10

→ Convert FOL to Resolution

Algorithm:

(i) Convert all sentences to CNF

(ii) Negate conclusion S and convert results to CNF.

(iii) add Negated conclusion S to the premise clauses

(iv) 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 resolvement 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

1) ¬food → likes (John, x)

2.) food (Apple)

3.) ¬eats (y, z) ∨ killed (y) ∨ food (z)

4.) eats (Anil, peanuts)

5.) alive (Anil)

6.) ¬eats (Anil, w) ∨ eats (Harry, w)

7.) killed (g) alive (g)

8.) ¬ alive (x) ∨ ¬ killed (x)

9.) likes (John, peanuts)

¬ likes (John, peanuts) ¬ food (x) ∨ likes (John, x)

¬ food (peanuts) ¬ eats (Y, Z) ∨ killed ∨ food Z

(Peanuts / Z)

¬ eats (Y, peanuts) ∨ eats (Anil, peanuts)

killed (Y)

killed (Anil) ¬ alive (x) ∨ ¬ killed

¬ alive (Anil) alive

O/P:

Proved by contradiction John likes peanuts

3/12/2024

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

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

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

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

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

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

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

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
```

forward_chaining()

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 9

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

Code:

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

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

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

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

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

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

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

```
Does John like peanuts? Yes
```

Program 10
Implement Alpha-Beta Pruning.

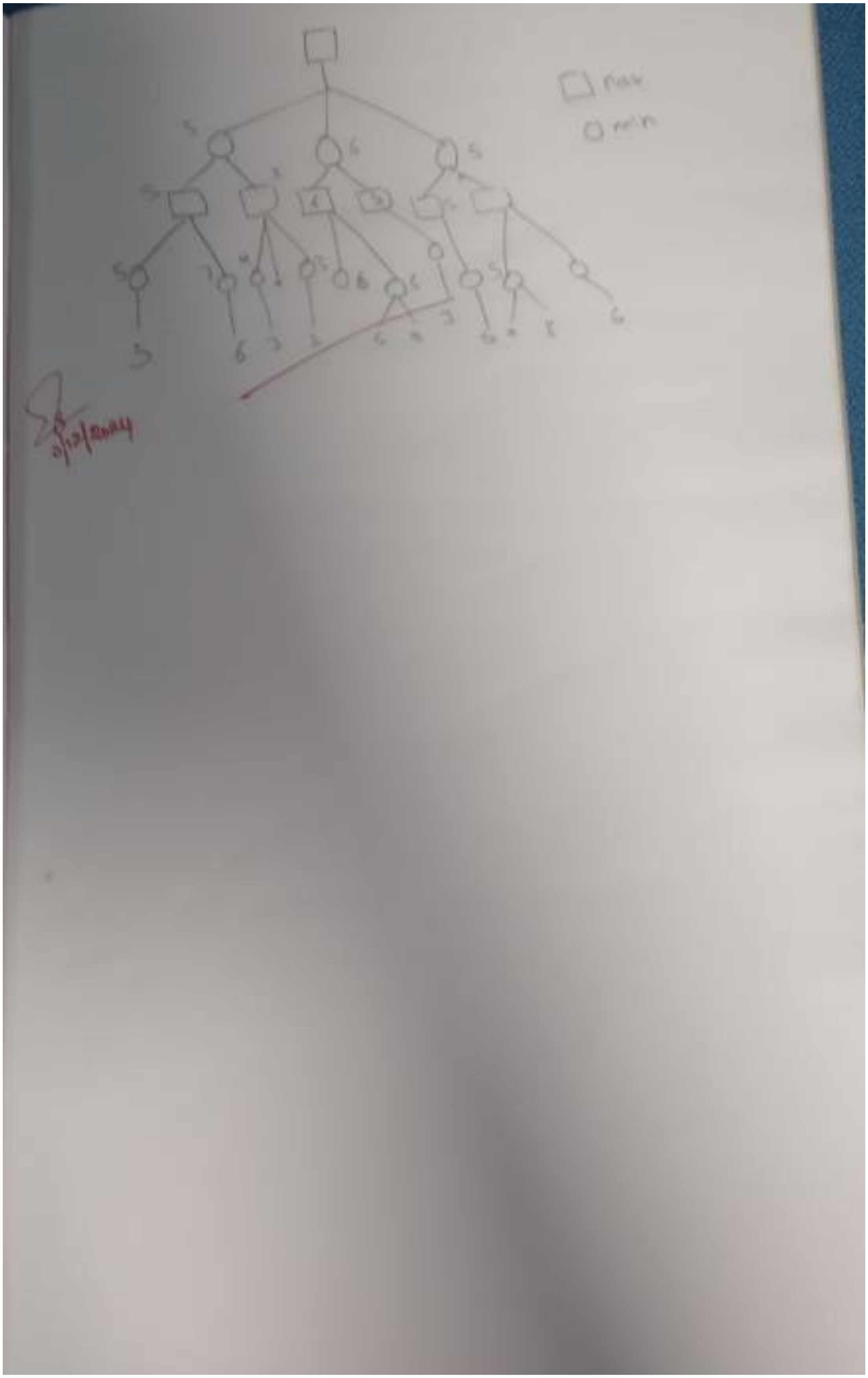
Algorithm:
(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

LAB-11

→ 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 calculations.
→ In both MIN and MAX Node, we return when $\alpha \geq \beta$ which compares with its parent node only.
→ Both minimax and alpha (α) - Beta (β) cut-off same path
→ Alpha (α) = Beta (β) gives optimal solutions at it takes less time to get The value from the root node

* Alpha (α) - Beta (β) gives optimal Solution as it takes less time to get the value for the root node

Output:
6


max
min

Code:

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

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

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

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1])  # Combine two nodes
            else:
                next_level.append(current_level[i])  # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0]  # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
```

```
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

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

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

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

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

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

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