VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Nikhilesh C (1BM22CS181)

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

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried outby Nikhilesh C **(1BM22CS181),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Tic-Tac-Toe (Lab 1: 24-09-2024)

Observation Book:

classmate
Date 24/09/2024
Page

Tic - Tac - Toe :

Algorithm :

```
Function play_game ()
    Initialize board as a list of 9 empty strings
    Set current_player to 'X'
    while True do
        Print board
        if current_player is 'X' then
            Repeat
                Prompt "Player X, choose a position (1-9):"
                Read move
                if move is invalid then
                    print "invalid input"
                else if board[move-1] is not empty then
                    print "That position is already taken."
                else
                    break
            End Repeat
            Set board[move - 1] to 'X'
            if check_winner (board, 'X') then
                print board
                print "it's a tie"
                break
            end if
    else
        print "Computer's turn (O) ... "
        move = computer_move (board)
        set board[move] to 'O'
        if check_winner (board, 'O') then
            print board
            print "Computer wins! "
            break
        end if
```

```
if is_board_full(board) then
    print board
    print "It's a tie!"
    break
end if
if current_player is 'X' then
    set current_player to 'O'
else
    set current_player to 'X'
end if
end function

Function computer_move(board)
  for i from 0 to 8 do
     if board[i] is empty then
        set board[i] to 'O'
        if check_winner(board, 'O') then
           return i
        end if
        set board[i] to empty
     end if
  end for
  for i from 0 to 8 do
     if board[i] is empty then
        set board[i] to 'X'
        if check_winner(board, 'X') then
           set board[i] to 'O'
           return i
        end if
        set board[i] to empty
     end if
  end for
```

```
if board[4] is empty then
    return 4.
end if
corners = [0, 2, 6, 8]
for corner in corners do
    if board[corner] is empty then
        return corner
    end if
end for
for i from 0 to 8 do
    if board[i] is empty then
        return i
    end if
end for
end function
```

Code :

```
def print_board (board) :
    print (f"{board[0]} | {board[1]} | {board[2]}")
    print ("--+---+--")
    print (f"{board[3]} | {board[4] | {board[5]}")
    print ("--+---+--")
    print (f"{board[6]} | {board[7] | {board[8]}")

def check_winner (board, player) :
    win_conditions = [ (0,1,2), (3,4,5), (6,7,8),
                       (0,3,6), (1,4,7), (2,5,8),
                       (0,4,8), (2,4,6)]
    return any (board[a] == board[b] == board[c] == player
                for a,b,c in win_conditions)
```

Date
Page

```
def is_board_full(board)
    return all(cell in ('X', 'O') for cell in board)

def computer_move(board):
    for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                if check_winner(board, 'O'):
                    return i
                board[i] = ' '
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'X'
            if check_winner(board, 'X'):
                board[i] = 'O'
                return i
            board[i] = ' '
    if board[4] == ' ':
        return 4
    corners = [0, 2, 6, 8]
    for corner in corners:
        if board[corner] == ' ':
            return corner
    for i in range(9):
        if board[i] == ' ':
            return i

def play_game():
    board = [' ' for _ in range(9)]
    current_player = 'X'
    while True:
        print_board(board)
        if current_player == 'X':
```

```
        try:
            move = int(input("Player X, choose a position
                                 (1-9): ")) -1
        except ValueError:
            print("Invalid input. Try again")
            continue
        if move<0 or move >8:
            print("Invalid move. Try again")
            continue
    else:
        print("Computer's turn (0)...")
        move = computer_move(board)
    board[move] = current_player
    if check_winner(board, current_player):
        print_board(board)
        print(f"{player {current_player} wins!")
        break
    if is_board_full(board):
        print_board(board)
        print("It's a tie!")
        break
    current_player = 'O' if current_player == 'X'
                else 'X'

play_game()
```

Output:

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

Output Screenshots:
Player X winning:

```
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, choose a position (1-9): 3
  |   | X
--+---+--
  |   |
--+---+--
  |   |
Computer's turn (O)...
  |   | X
--+---+--
  | O |
--+---+--
  |   |
Player X, choose a position (1-9): 7
  |   | X
--+---+--
  | O |
--+---+--
X |   |
Computer's turn (O)...
O |   | X
--+---+--
  | O |
--+---+--
X |   |
```

```
Player X, choose a position (1-9): 9
O |   | X
--+---+--
  | O |
--+---+--
X |   | X
Computer's turn (O)...
O |   | X
--+---+--
  | O | O
--+---+--
X |   | X
Player X, choose a position (1-9): 8
O |   | X
--+---+--
  | O | O
--+---+--
X | X | X
Player X wins!
```

Player O winning (Computer winning):

```
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, choose a position (1-9): 6
  |   |
--+---+--
  |   | X
--+---+--
  |   |
Computer's turn (O)...
  |   |
--+---+--
  | O | X
--+---+--
  |   |
Player X, choose a position (1-9): 3
  |   | X
--+---+--
  | O | X
--+---+--
  |   |
```

```
Computer's turn (O)...
  |   | X
--+---+--
  | O | X
--+---+--
  |   | O
Player X, choose a position (1-9): 2
  | X | X
--+---+--
  | O | X
--+---+--
  |   | O
Computer's turn (O)...
O | X | X
--+---+--
  | O | X
--+---+--
  |   | O
Player O wins!
```

Tie:

```
      RESTART: D:/Nikhilesh/BABCL/Seme
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, choose a position (1-9): 5
  |   |
--+---+--
  | X |
--+---+--
  |   |
Computer's turn (O)...
O |   |
--+---+--
  | X |
--+---+--
  |   |
Player X, choose a position (1-9): 7
O |   |
--+---+--
  | X |
--+---+--
X |   |
Computer's turn (O)...
O |   | O
--+---+--
  | X |
--+---+--
X |   |
```

```
Player X, choose a position (1-9): 4
O | X | O
--+---+--
X | X |
--+---+--
X | O |
Computer's turn (O)...
O | X | O
--+---+--
X | X | O
--+---+--
X | O |
Player X, choose a position (1-9): 9
O | X | O
--+---+--
X | X | O
--+---+--
X | O | X
It's a tie!
```

LAB-2: Vacuum Cleaner

Observation book:

Date 01/10/24

<u>Vacuum Cleaner :</u>

<u>Algorithm :</u>

2 Parameters → State, Location
Functions :
* Return state
* Suction ON
* Suction OFF
* Change Room
* Count

Function ON (~~State, Location~~):
    Print ("Suction Turned ON")
    ~~State = "Clean"~~

Function OFF (~~State, Location~~):
    Print ("Suction Turned OFF")

Function ~~Right~~ Turn (Location):
    if Location == 1 :
        Location = 2
    ~~else~~: Print ("Turning right to enter room 2")

~~Function~~
~~Function Left (Location):~~
    ~~Location = 2~~
        Location = 1
        Print ("Turning left to enter room 1")

~~Function Vacuum ():~~
    ~~state = "Dirty"~~
    ~~Location = 1~~

~~Function State (State):~~
    ~~return State~~

Function Vacuum ():
    ~~st~~ State = "Dirty"
    Location = 1
    for i in range (2):
        if State == "Dirty" :
            ON()

State = " Clean "
OFF() → Print (" Room is clean now ")
Turn (Location)
else :
~~Turn (Loc~~
Print (" Room Already clean ")
Turn (Location)

Percept Sequence :

① 1, Clean			
"	② 1, Right		
"	"	③ 2, Left	
"	"	"	④ 1, Right

① 1, Check						
① 1, Check	② 1, Clean					
① 1, Check	② 1, Clean	③ 1, Right				
① 1, Check	② 1, Clean	③ 1, Right	④ 2, Check			
① 1, Check	② 1, Clean	③ 1, Right	④ 2, Check	⑤ 2, Left		
① 1, Check	② 1, Clean	③ 1, Right	④ 2, Check	⑤ 2, Left	⑥ 1, Check	
① 1, Check	② 1, Clean	③ 1, Right	④ 2, Check	⑤ 2, Left	⑥ 1, Check	⑦ 1, Right

Program :

```
def ON():
    print("Suction Turned ON")

def OFF():
    print(" Suction Turned OFF ")
```

```
def Turn (Location , Direction ):
    if Direction == "forward":
        if Location == 1 :
            Location = 2
            print ("Turning Right to enter room 2")
        elif Location == 2:
            Location = 3
            print ("Turning Right to enter room 3")
        elif Location == 3:
            Location = 4
            print ("Turning Right to enter room 4")
        else:
            pass
    else:
        if Location == 2 :
            Location = 1
            print (" Turning left to enter room 1")
        elif Location == 3:
            Location = 2
            print ("Turning left to enter room 2")
        elif Location == 4:
            Location = 3
            print (" Turning left to enter room 3")
        else:
            pass.
    return Location

state = " Dirty "
Location = 1
print (" Starting in room 1 ")
print (" The room is dirty ")
```

```
for i in range(4):
    if state == "Dirty":
        ON()
        state = "Clean"
        print("Room is clean now")
        OFF()
        Location = Turn(Location, "forward")
        "# "forward""
for i in range(3):
    if state == "Dirty":
        ON()
        state = "Clean"
        print("Room is clean now")
        OFF()
        Location = Turn(Location, "reverse")
    else:
        print("Room is already clean")
        Location = Turn(Location, "reverse")
```

Output:

```
Starting in room 1
The room is dirty
Suction turned ON
Room is clean now
Suction turned OFF
Turning right to enter room 2
Room is already clean
Turning right to enter room 3
:
```

Output:

```
Enter the number of rows: 2
Enter the number of columns: 2
Enter the number of dirty cells: 2
Enter coordinates for 2 dirty cells (format: row,col row,col ...):
0,0 1,1
Initial grid state:
[1, 0]
[0, 1]
Cleaning position (0, 0)
Position (0, 1) is already clean
Position (1, 0) is already clean
Cleaning position (1, 1)
Final grid state:
[0, 0]
[0, 0]
Nikhilesh 1bm22cs181
```

Lab-3: 8 puzzle problems using DFS and Manhattan distance

Observation book:

classmate
Date 08/10/2024
Page ____

8 - Puzzle :

~~DFS: Manhattan Distance~~ ~~BFS~~: Manhattan Distance :

Algorithm :

* Initialize the goal array in a function.
- This function returns True if current state matches the goal. Else False.

* Define 4 functions : Move left, move right, move up, move down.

* Identify the position of the blank tile.
- If corner, make a move for the neighbouring 2 tiles.
- If center, make a move for the neighbouring 4 Tiles.
- Else, make a move for the neighbouring 3 Tiles.

* For every move, check the Manhattan distance.
If it is

Algorithm :

* Represent the puzzle state as a list where 0 represents the empty space.
* Define a function to check if the current state matches the goal state.
* Define a function to calculate the Manhattan Distance for a given state.
* Define a function to return a list of new

states and their updated positions upon making a move.

* Use a stack to store the states after each move.
* Compare the top element of this stack with the ~~final~~ goal state.
  - If it is a match, then pop the states one by one and add it to the path array.
    - Reverse the path array to print it.
  - If not a match, make the next move and add the new state onto the stack.

* Maintain uniqueness of the states to avoid an infinite loop.

Check with DFS proof

Code :

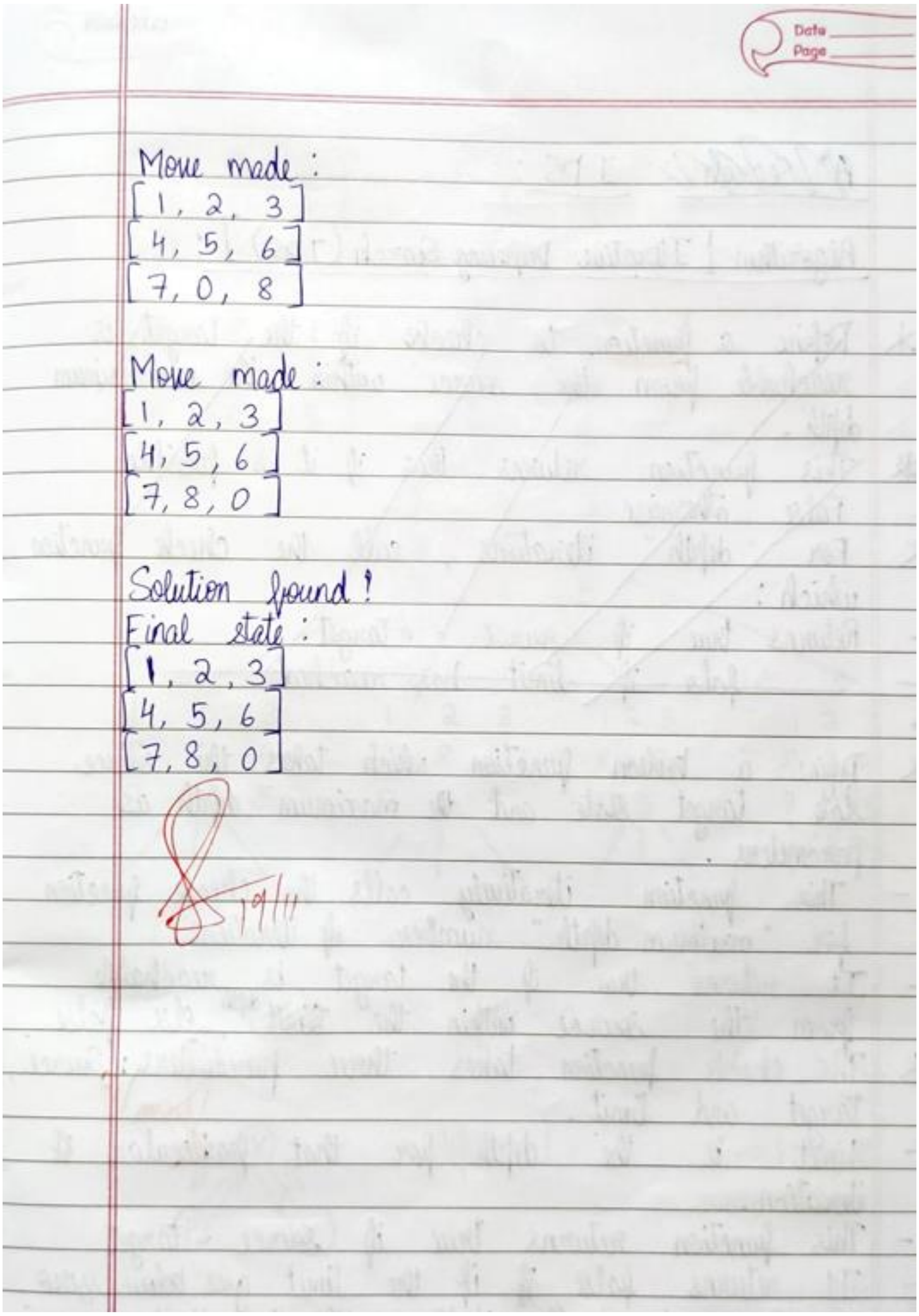
```
import copy

directions = [(-1,0), (1,0), (0,-1), (0,1)]
goal = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]]

def Manhattan(state):
    d = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                gx, gy = divmod(state[i][j]-1, 3)
                d += abs(i-gx) + abs(j-gy)
    return d
```

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

def goal(state):
    return state == goal

def print_board(state):
    for row in state:
        print(row)
    print("\n")

def dfs(state, depth, moves):
    bx, by = find_blank(state)
    if goal(state):
        return True, state, moves
    if depth == 0:
        return False, None, moves
    possible_moves = []
    for dx, dy in directions:
        nx, ny = bx + dx, by + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = copy.deepcopy(state)
            new_state[bx][by], new_state[nx][ny] =
                new_state[nx][ny], new_state[bx][by]
            md = Manhattan(new_state)
            possible_moves.append((md, new_state))
    possible_moves.sort(key = lambda x: x[0])
    for _, next_state in possible_moves:
        moves.append(next_state)
        print("Move made: ")
```

```
            print_board(next_state)
            found, result, moves = dfs(next_state, depth-1,
                                        moves)
            if found:
                return True, result, moves
            moves.pop()
    return False, None, moves

def solve(initial, depth = 30):
    moves = [initial]
    print("Initial State:")
    print_board(initial)
    found, final, moves = dfs(initial, depth, moves)
    if found:
        print("Solution Found!")
        print("Final state:")
        print_board(final)
    else:
        print("No solution found within the depth limit")

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

solve(initial)
```

Output:

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



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

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

Solution found !
Final state :
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

19/11

Output:

```
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
1 0 3
4 2 6
7 5 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Nikhilesh C 1BM22CS181
```

LAB-4: 8-Puzzle with A* and IDFS

Observation book:

Date 14/10/2024
Page

~~A* Search~~: IDS:

Algorithm [Iterative Deepening Search (IDS)]:

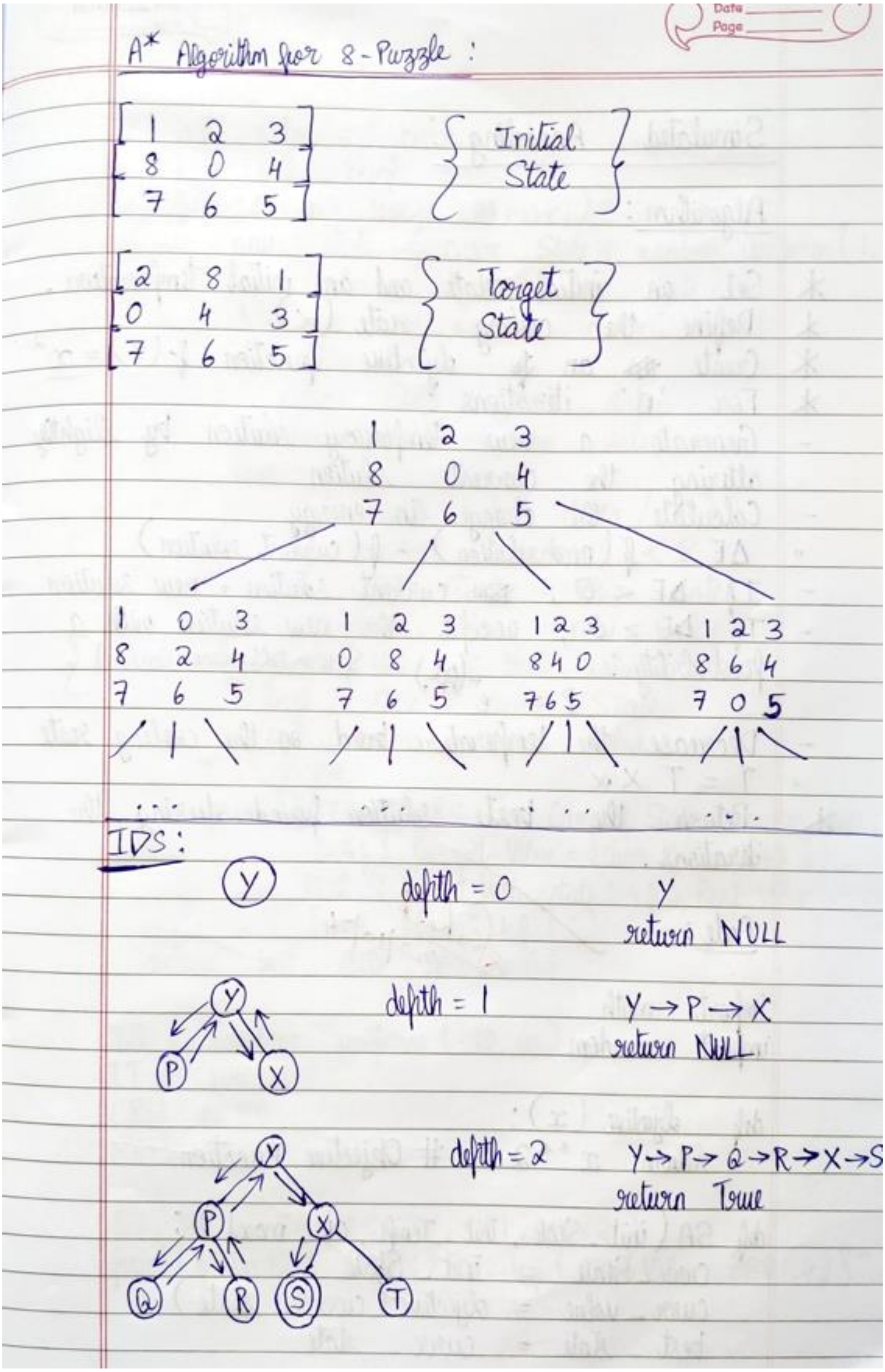
~~* Define a function to check if the target is reachable from the source within the maximum depth.~~
~~- * This function returns true if it is possible~~
~~- False otherwise~~
~~* For "depth" iterations, call the check function which:~~
~~- Returns true if source = target~~
~~- " false if limit has reached~~

* Define a boolean function which takes the source state, target state and the maximum depth as parameters.

- This function iteratively calls the check function for "maximum depth" number of iterations.
- It returns true if the target is reachable from the source within the depth, else false.

* The check function takes three parameters: Source, target and limit.

- Limit is the depth for that particular iteration.
- This function returns true if source = target
- It returns false if the limit goes below zero.
- It recursively calls itself with the limit decreasing by one for each recursion
- The check function returns true if the function it recursively calls is true.
- If none of these conditions are met, it returns false



Date ______ Page ______

A* Algorithm for 8-Puzzle :

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad \left\{ \text{Initial State} \right\}$$

$$\begin{bmatrix} 2 & 8 & 1 \\ 0 & 4 & 3 \\ 7 & 6 & 5 \end{bmatrix} \quad \left\{ \text{Target State} \right\}$$

1	2	3
8	0	4
7	6	5

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

IDS :

depth = 0

Y

return NULL

depth = 1

Y → P → X

return NULL

depth = 2

Y → P → Q → R → X → S

return True

A* algorithm

Code:

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

def flatten(puzzle):
    return [item for row in puzzle for item in row]

def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j

def misplaced_tiles(puzzle):
    flat_puzzle = flatten(puzzle)
    flat_goal = flatten(goal_state)
    return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i] != 0])

def generate_neighbors(puzzle):
    x, y = find_blank(puzzle)
    neighbors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_puzzle = [row[:] for row in puzzle]
            new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
            neighbors.append(new_puzzle)
    return neighbors

def is_goal(puzzle):
    return puzzle == goal_state

def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()

def a_star_misplaced_tiles(initial_state):
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()

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

```
        print("Current State:")
        print_puzzle(current_state)
        h = misplaced_tiles(current_state)
        print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
        print("-" * 20)

        if is_goal(current_state):
            print("Goal reached!")
            return path

        visited.add(tuple(flatten(current_state)))
        for neighbor in generate_neighbors(current_state):
            if tuple(flatten(neighbor)) not in visited:
                h = misplaced_tiles(neighbor)
                heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))

    return None

def get_input():
    print("Enter the initial state of the puzzle (3x3 matrix, each row separated by space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i + 1}: ").split()))
        initial_state.append(row)
    return initial_state

initial_state = get_input()
solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print("Nikhilesh 1BM22CS181")
```

Output:

```
Enter the initial state of the puzzle (3x3 matrix, each row separated by space):
Enter row 1: 1 2 0
Enter row 2: 3 4 5
Enter row 3: 6 7 8
Current State:
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

g(n) = 0, h(n) = 2, f(n) = 2
--------------------
Current State:
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 1, h(n) = 1, f(n) = 2
--------------------
Current State:
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 2, h(n) = 0, f(n) = 2
--------------------
Goal reached!
Solution found!
Nikhilesh 1BM22CS181
```

IDFS:

Code:

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        self.adjacency_list[u].append(v)

    def depth_limited_dfs(self, node, goal, limit, visited):
        if limit < 0:
            return False
        if node == goal:
            return True
        visited.add(node)
        for neighbor in self.adjacency_list.get(node, []):
            if neighbor not in visited:
                if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
                    return True
        visited.remove(node)  # Allow revisiting for the next iteration
        return False

    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.depth_limited_dfs(start, goal, depth, visited):
```

```
            return True
        return False


def main():
    graph = Graph()

    # Input number of edges
    num_edges = int(input("Enter the number of edges: "))

    # Input edges
    for _ in range(num_edges):
        edge = input("Enter an edge (format: A B): ").split()
        graph.add_edge(edge[0], edge[1])

    start_node = input("Enter the start node: ")
    goal_node = input("Enter the goal node: ")
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    if graph.iddfs(start_node, goal_node, max_depth):
        print(f"Goal node {goal_node} found!")
    else:
        print(f"Goal node {goal_node} not found within depth {max_depth}.")


if __name__ == "__main__":
    main()
print("Nikhilesh 1bm22cs181")
```

Output:

```
Enter the number of edges: 4
Enter an edge (format: A B): A B
Enter an edge (format: A B): B C
Enter an edge (format: A B): C D
Enter an edge (format: A B): D E
Enter the start node: A
Enter the goal node: E
Enter the maximum depth for IDDFS: 4
Goal node E found!
Nikhilesh 1bm22cs181
```

LAB-5: Simulated Annealing Algorithm

Observation book:

Date 22/10/2024
Page

Simulated Annealing :

Algorithm :

* Set an initial state and an initial temperature.
* Define the cooling rate (α)
* Create ~~an~~ an ~~f~~ objective function $f(x) = x^2$
* For 'n' iterations :
  - Generate a new temporary solution by slightly altering the current solution
  - Calculate the change in energy
    - $\Delta E = f(\text{new solution}) - f(\text{current solution})$
  - If $\Delta E < 0$, ~~...~~ current solution = new solution
  - If $\Delta E > 0$, accept the new solution with a probability : $P = e^{-\left(\frac{\Delta E}{KT}\right)}$ { K → Boltzmann Constant }
  - Decrease the temperature based on the cooling rate
    - $T = T \times \alpha$
* Return the best solution found during the iterations.

Code :

Sneha SB 22/10/24

```
import math
import random

def objective(x):
    return x**2  # Objective Function

def SA(init_State, init_Temp, CR, maxI):
    curr_State = init_State
    curr_value = objective(curr_state)
    best_state = curr_state
```

```
best_value = curr_value
T = init_Temp
for i in range(maxl):
    new_state = curr_State + random.uniform(-1,1)
    new_value = objective(new_state)
    delta = new_value - curr_value
    if delta < 0:
        curr_State = new_state
        curr_Value = new_value
    else:
        P = math.exp(-delta / T)
        if random.random() < P:
            curr_State = new_state
            curr_Value = new_value
    if curr_Value < best_value:
        best_state = curr_State
        best_value = curr_Value
    T *= CR
    print(f"Iteration {i+1}: Current State = {curr_State:
        0.4f}, Current Value = {curr_Value:.4f},
        Best State = {best_state:.4f}, Best Value =
        {best_value:.4f}")
return best_state, best_value

IS = random.uniform(-10,10)
IT = 100
CRl = 0.95
MAX2 = 50

BS, BV = SA(IS,IT,CRl,MAX2)
print(f"\n Best State:{BS:.4f}, Best Value:{BV:.4f}")
```

Output :

Iteration 1 : CS = -1.6967 , CV = 2.8789 , BS = -1.3766 , BV = 1.8950
Iteration 2 : CS = -2.1096 , CV = 4.4503 , BS = -1.3766 , BV = 1.8950
⋮
Best State : -0.0235 , Best Value : 0.0006

22/10/24

Output:

```
Enter the initial state (starting point): 15
Enter the initial temperature: 10
Enter the cooling rate (between 0 and 1): 0.5
Enter the number of iterations: 4
Iteration 1: Current State = 15.8992, Current Energy = 252.7846, Temperature = 5.0000
Iteration 2: Current State = 15.4894, Current Energy = 239.9222, Temperature = 2.5000
Iteration 3: Current State = 15.4894, Current Energy = 239.9222, Temperature = 1.2500
Iteration 4: Current State = 14.4963, Current Energy = 210.1414, Temperature = 0.6250
Best State: 14.4963, Best Energy: 210.1414
Nikhilesh 1BM22CS181
```

LAB-6 - Implementing A* and Hill Climbing Algorithm on 8 Queens.

Observation book:

Date 29/10/2024
Page ______

A* Search Algorithm :

Algorithm :

* Initialize an open set with the starting node and a closed set as empty.
* Define a function to calculate the heuristic cost from the start state to the goal state with starting and ending nodes as parameters.
* While the open set is not empty, remove the node with the lowest total cost (including heuristic).
* If goal state is reached, return the path taken and the cost.
* Else, generate neighbours of the current node using heuristics and iterate through each neighbour.
- This neighbour should be within bounds and not belong to the closed set.
* Calculate the cost from the starting node to the neighbour and update the minimum cost if needed.
- Add this neighbour to the closed set and proceed to step 4.
* Once all the neighbours are processed, if the goal state has not been reached and the open set is empty, then no path exists from the given start state to the goal state.

~~Hill climbing Search Algorithm : [For 8-queens]~~

A* Search Algorithm for 8-queens Problem :

* Initialize the open set with initial state and its cost.
* While the open set is not empty :

* Select the state with the lowest-total cost
  - $f(n) = g(n) + h(n)$
  - $g(n) \rightarrow$ No. of queens placed so far
  - $h(n) \rightarrow$ No. of queens remaining to be placed.
* If this state is the goal state [$h(n) = 0$], then return the solution.
* Generate the next possibilities by placing a queen in the next row.
  - Compute $f(n)$ at each step.
* If a deadend is reached, backtrack.

Hill Climbing Search Algorithm [8-queens]:

* Place 8 queens randomly on the chess board
* Calculate the number of attacking pairs (conflicts)
* If there are 0 attacking pairs, then the goal state has been reached.
* ~~Gen~~ Generate neighbouring states by moving one queen to a different column in the same row.
* Select the neighbour with the fewest conflicts.
* If ~~a~~ ~~neighbour~~ the best neighbour has fewer conflicts than the current state, move to that neighbour.
* Go back to step 2 if the goal state is not reached.

Output A*:

<P.T.O>

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

<u>Output Hill Climbing Search :</u>

```
. Q . . . . . .
Q . . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q
```

```
. Q . . . . . .
. . . Q . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q
Q . . . . . . .
```

.
.
.

<P.T.O>

A* algorithm:

Code:

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g  # cost to reach this state
        self.h = h  # heuristic cost to reach goal
```

```
        self.f = g + h  # total cost

    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens(initial_state):
    open_list = []
    closed_set = set()
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

# Get user input for the initial state
initial_state = []
for i in range(8):
    while True:
        try:
            row = int(input(f"Enter row for queen {i+1} (0 to 7): "))
            if 0 <= row < 8 and row not in initial_state:
                initial_state.append(row)
                break
```

```
            else:
                print("Invalid row. Enter a number between 0 and 7, and each row must be unique.")
        except ValueError:
            print("Invalid input. Please enter an integer between 0 and 7.")

# Execute the A* algorithm
solution = a_star_8_queens(initial_state)
if solution:
    print("A* solution:", solution)
else:
    print("A* solution: No solution found.")
print("Nikhilesh 1bm22cs181")
```

output:

```
A* solution: [0, 2, 4, 6, 1, 3, 5, 7]
1BM22CS181
>>>
```

Hill climbing:

Code:

```
import random

def heuristic(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def hill_climbing_8_queens(initial_state):
    state = initial_state  # Start with user-provided initial state
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state
```

```
        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
                    new_state[col] = row
                    h = heuristic(new_state)
                    if h < next_h:
                        next_h = h
                        next_state = new_state

        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum

        state = next_state

# Get user input for the initial state
initial_state = []
for i in range(8):
    while True:
        try:
            row = int(input(f"Enter row for queen {i+1} (0 to 7): "))
            if 0 <= row < 8 and row not in initial_state:
                initial_state.append(row)
                break
            else:
                print("Invalid row. Enter a number between 0 and 7, and each row must be
unique.")
        except ValueError:
            print("Invalid input. Please enter an integer between 0 and 7.")

# Execute the Hill Climbing algorithm
solution = hill_climbing_8_queens(initial_state)
if solution:
    print("Hill Climbing solution:", solution)
else:
    print("Hill Climbing solution: No solution found.")
print("1BM22CS181")
```

Output:

```
Hill Climbing solution: [0, 2, 4, 6, 1, 3, 5, 7]
1BM22CS181
>>>
```

LAB-7: Entailment Using Literals

Observation book:

Date 12/11/2024
Page

Propositional Logic :

Knowledge Base :

1. Alice is the mother of Bob.
2. Bob is the father of Charlie.
3. A father is a parent.
4. A mother is a parent.
5. All parents have children.
6. If someone is a parent, their children are siblings.
7. Alice is married to David.

Hypothesis :

Charlie is a sibling

Premise from Knowledge Base :

* Bob is the father of Charlie (A)
* A father is a parent (B)
* Alice is the mother of Bob (C)
* A mother is a parent (D)
* If someone is a parent, their children are siblings (E)

Entailment Process :

* From the first entailment, we can conclude that Bob is Charlie's parent
* From the third entailment, we can conclude that Alice is Bob's parent.
* From the above, we can conclude that Alice is a parent of Charlie. (By Transitivity)

* From the fifth premise, we can conclude that Charlie is a sibling of Bob.

Hence, the hypothesis "Charlie is a sibling of Bob" is entailed by the knowledge base, because it logically follows from the premisis

Code:

```
import re

# Helper function to parse user input into logical predicates
def parse_input(input_sentence, knowledge_base):
    # Convert the sentence to lowercase for consistency
    input_sentence = input_sentence.lower()

    # Match patterns for predicates and facts (e.g., 'X is the mother of Y' or 'X is married to Y')

    # Fact or Rule: "X is the mother of Y"
    mother_match = re.match(r"(\w+) is the mother of (\w+)", input_sentence)
```

```
    # Fact or Rule: "X is the father of Y"
    father_match = re.match(r"(\w+) is the father of (\w+)", input_sentence)

    # General rule: "All X have children"
    parent_match = re.match(r"all (\w+) have children", input_sentence)

    # Rule for parent-child relation and siblings
    parent_rule_match = re.match(r"if someone is a parent, their children are siblings",
input_sentence)

    # General fact: "X is married to Y"
    married_match = re.match(r"(\w+) is married to (\w+)", input_sentence)

    # Parsing rules and facts
    if mother_match:
        mother, child = mother_match.groups()
        # Add the mother-child relationship to knowledge base
        knowledge_base["Mother"].append((mother.capitalize(), child.capitalize()))

    elif father_match:
        father, child = father_match.groups()
        # Add the father-child relationship to knowledge base
        knowledge_base["Father"].append((father.capitalize(), child.capitalize()))

    elif parent_match:
        parent = parent_match.group(1)
        # Rule: All X are parents with children
        knowledge_base["ParentRule"].append((parent.capitalize(), "HasChildren"))

    elif parent_rule_match:
        # General rule: If someone is a parent, their children are siblings
        knowledge_base["ParentSiblingRule"].append(("Parent", "Siblings"))

    elif married_match:
        spouse1, spouse2 = married_match.groups()
        # Add the married relationship to knowledge base
        knowledge_base["Married"].append((spouse1.capitalize(), spouse2.capitalize()))

# Function to check if two children are siblings
def are_siblings(child1, child2, knowledge_base):
    # Check if both children share the same parent
    parents = set()
    for mother, child in knowledge_base["Mother"]:
        if child == child1:
            parents.add(mother)
        if child == child2:
            parents.add(mother)

    for father, child in knowledge_base["Father"]:
```

```
        if child == child1:
            parents.add(father)
        if child == child2:
            parents.add(father)

    return len(parents) > 1  # If both children share a parent, they are siblings

# Function to check the hypothesis "Charlie is a sibling of Bob"
def check_hypothesis(hypothesis, knowledge_base):
    # Parse the hypothesis
    hyp_match = re.match(r"(\w+) is a sibling of (\w+)", hypothesis.lower())
    if hyp_match:
        child1, child2 = hyp_match.groups()
        # Check if the children are siblings
        if are_siblings(child1.capitalize(), child2.capitalize(), knowledge_base):
            return True
    return False

# Main function for user input and entailment reasoning
def main():
    # Create an empty knowledge base
    knowledge_base = {
        "Mother": [],
        "Father": [],
        "ParentRule": [],
        "ParentSiblingRule": [],
        "Married": []
    }

    print("Enter knowledge base rules. Type 'done' when finished.")

    # Allow the user to input knowledge base facts, rules, or actions
    while True:
        user_input = input("Enter fact/rule/action: ").strip()
        if user_input.lower() == "done":
            break
        parse_input(user_input, knowledge_base)

    # Print the current knowledge base
    print("\nCurrent Knowledge Base:")
    for category, items in knowledge_base.items():
        print(f"{category}: {items}")

    # Ask for the hypothesis (the statement to check)
    hypothesis = input("\nEnter hypothesis to check: ").strip()

    # Check if the hypothesis is entailed
    if check_hypothesis(hypothesis, knowledge_base):
        print(f"\nConclusion: The hypothesis '{hypothesis}' is entailed by the knowledge base.")
    else:
```

print(f"\nConclusion: The hypothesis '{hypothesis}' is NOT entailed by the knowledge base.")

# Run the program
main()
print("Nikhilesh 1BM22CS181")

Output :

```
Enter knowledge base rules. Type 'done' when finished.
Enter fact/rule/action: Alice is the mother of Bob
Enter fact/rule/action: Bob is the father of Charlie
Enter fact/rule/action: A father is a parent
Enter fact/rule/action: A mother is a parent
Enter fact/rule/action: All parents have children
Enter fact/rule/action: If someone is a parent,their children are siblings
Enter fact/rule/action: Alice is married to David
Enter fact/rule/action: done

Current Knowledge Base:
Mother: [('Alice', 'Bob')]
Father: [('Bob', 'Charlie')]
ParentRule: [('Parents', 'HasChildren')]
ParentSiblingRule: []
Married: [('Alice', 'David')]

Enter hypothesis to check: Charlie is a sibling of Bob

Conclusion: The hypothesis 'Charlie is a sibling of Bob' is entailed by the knowledge base.
```

LAB-8: FOL using Unification.

Observation book:

classmate
Date 19/11/2024
Page

First Order Logic: [Unification]

"If every dog has a tail, and all dogs that are friendly are also playful, and Fido is a friendly dog, then Fido is playful and has a tail"

* Dog(x) : " x is a dog "
* Tail(x) : " x has a tail "
* Friendly(x) : " x is friendly "
* Playful(x) : " x is playful "

FOL representation:

1 * $\forall x\ (Dog(x) \rightarrow Tail(x))$
- For every x, if x is a dog, then x has a tail

2 * $\forall x\ ((Dog(x) \wedge Friendly(x)) \rightarrow Playful(x))$
- For every x, if x is a dog and x is friendly, then x is playful.

3 * Friendly (Fido)
- Fido is friendly

4 * Dog (Fido)
- Fido is a dog

* Unification:

* From 1 & 4, substituting x with Fido,
Dog(Fido) $\rightarrow$ Tail(Fido)
- By Modus Ponens, Tail (Fido) —— 5
(Fido has a tail)

* From 2, 3 & 4, substituting x with Fido,

(Dog (Fido) $\land$ Friendly (Fido)) $\rightarrow$ Playful (Fido)

- By Modus Ponens, Playful (Fido) — 6
  (Fido is playful)

* From 5 and 6,

Playful (Fido) $\land$ Tail (Fido)

- Fido is playful and Fido has a tail

code::

```
import re
# Define a simple function for extracting predicates from sentences
def extract_predicate(sentence):
# Regular expression to find patterns like Predicate(Argument)
pattern = r"([A-Za-z]+)\((\w+)\)"
match = re.search(pattern, sentence)
if match:
predicate = match.group(1)
subject = match.group(2)
return predicate, subject
return None, None
# Function for unification
def unify(fact, query):
# Check if the fact and query are the same
if fact == query:
return True
# Extract predicate and subject from fact and query
fact_predicate, fact_subject = extract_predicate(fact)
query_predicate, query_subject = extract_predicate(query)
# If predicates match, unify the subjects
if fact_predicate == query_predicate:
if fact_subject == query_subject:
return True
else:
# Here, we could handle variable substitution (unification)
return False
return False
# Function to deduce the goal using given rules
def deduct(rules, goal):
# Try to find unification for the goal from the rules
for rule in rules:
if unify(rule, goal):
print(f"Unification successful: {rule} matches with {goal}.")
return True
return False
# Main function to handle user input
def main():
# Step 1: Get the rules (facts/implications) from the user
print("Enter the rules (facts/implications). Type 'done' to finish entering rules.")
rules = []
while True:
rule_input = input("Enter rule: ")
if rule_input.lower() == 'done':
break
else:
rules.append(rule_input.strip())
# Step 2: Get the goal (query) from the user
goal_input = input("Enter the goal (query) to prove: ").strip()
# Step 3: Try to deduce the goal using the given rules
print("\nAttempting to deduce the goal...")
```

```
if deduct(rules, goal_input):
print(f"Conclusion: The goal '{goal_input}' is true based on the rules.")
else:
print(f"Conclusion: The goal '{goal_input}' cannot be proven with the provided rules.")
# Run the program
main()
print("Nikhilesh 1bm22cs181")
```

Output: Output:

```
Enter the rules (facts/implications). Type 'done' to finish entering rules.
Enter rule: all birds can fly
Enter rule: bluey is a bird
Enter rule: done
Enter the goal (query) to prove: bluey can fly

Attempting to deduce the goal...
Unification successful: all birds can fly matches with bluey can fly.
Conclusion: The goal 'bluey can fly' is true based on the rules.
Nikhilesh 1bm22cs181
```

LAB-9: FOL(forward chaining),Min-max(tic-tac-toe), Alpha-Beta Pruning(8-Queens)

Observation book:

Date 03/12/24

First Order Logic [Forward Chaining]:

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

Prove that "Robert is criminal"

Predicates:

* American (x) : x is an American citizen
* Hostile (x, y) : x is a hostile nation to y
* Sold (x, m, y) : x sold missile m to y
* Criminal (x) : x is a criminal

~~Logic al~~ Logical Axioms:

* It is a crime for an American to sell weapons to hostile nations.

- $\forall x\ \forall m\ \forall y$ (American(x) $\land$ Hostile$(y$, America$)$ $\land$ Sold$(x, m, y) \rightarrow$ Criminal(x))

* Robert is an American.

- American (Robert)

* Country A is hostile to America.

- Hostile (A, America)

* Robert sold missiles to country A:

- $\exists m$ Sold (Robert, m, A)

Forward Chaining:

* Using the above facts, we can use forward chaining to combine them and arrive at:

American(Robert) ∧ Hostile (A, America) ∧ Sold (Robert, m, A) → Criminal (Robert).

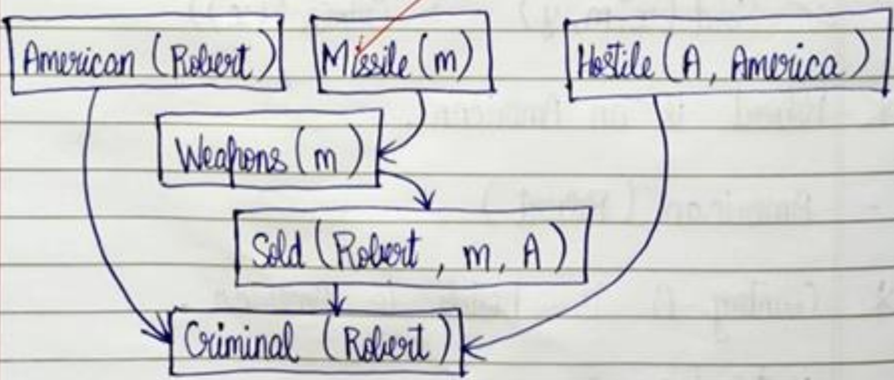
Conclusion:

By forward chaining, we can conclude that "Robert is a criminal"

Note:

* If question says forward chaining with proof, show each derivation:
- Missile (x) ⇒ Weapon (x)
- Draw the derivation tree level by level.

Derivation Tree:



classmate
Date 3/12/2024
Page

Min-Max Algorithm (Tic-Tac-Toe) :

* Minimizing the possible loss for a worst-case scenario by maximizing the player's minimum gain.

Algorithm :

```
                 board  player
Function check ( b , p )
    for i in range(3) do
        if all (board[i][j] == p for j in range(3)) do
            return True
        end if
        if all (board[j][i] == p for j in range(3)) do
        end if  return True
    end for
    if all (board[i][i] == p for i in range(3)) do
        return True
    end if
    if all (board[i][2-i] == p for i in range(3)) do
        return True
    end if
    return False

Function FullBoard (board)
    for r in board do
        if " " in r do
            return False
        end if
    end for
    return True
```

< P.T.O >

```
Function Evaluate (b)
    if check (b, X) do
        return 1
    end if
    if check (b, O) do
        return -1
    end if
    return 0

Function minmax (b, d, flag)
    score ← evaluate (b)
    if score == 1 or score == -1 or FullBoard(b) do
        return score
    end if
    if flag do
        best ← -float ('inf')
        for i in range (3) do
            for j in range (3) do
                if b[i][j] == " " do
                    b[i][j] ← "X"
                    best ← max (best, minmax(b, d+1, False))
                    b[i][j] ← " "
                end if
            end for
        end for
        return best
    else do
        best ← float ('inf')
        for i in range (3) do
            for j in range (3) do
                if b[i][j] == " " do
                    b[i][j] ← "O"
```

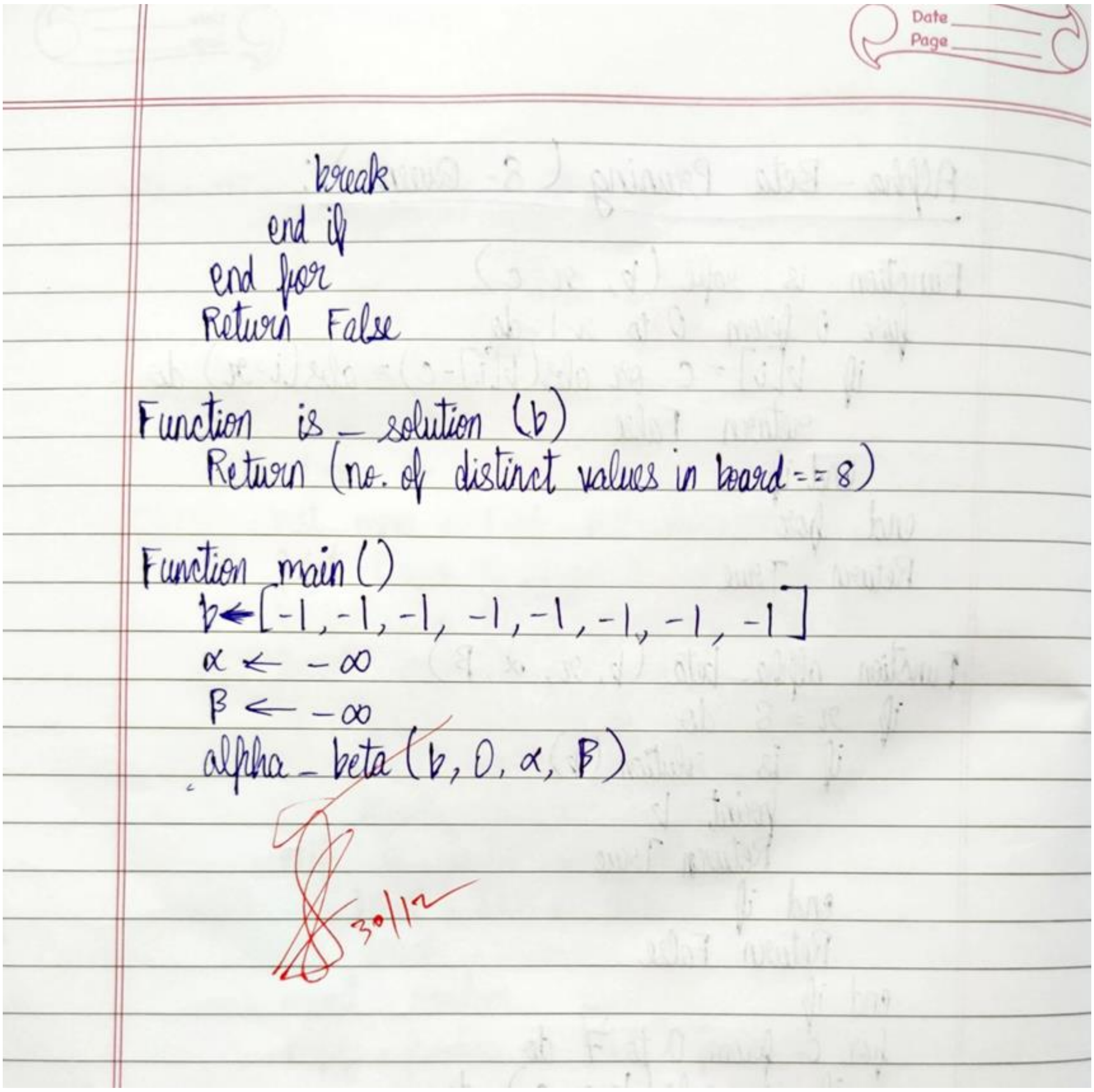
```
best ← min (best, minmax (b, d+1, True))
b[i][j] ← " "
end if
end for
end for
return best
end if

Function Find_best_move (b)
best_val ← - float ('inf')
best_move ← (-1, -1)
for i in range (3) do
for j in range (3) do
if b[i][j] = EMPTY do
b[i][j] ← X
move_val ← minmax (b, 0, False)
b[i][j] ← Empty
if move_val > best_val do
best_move ← (i, j)
best_val ← move_val
end if
end if
end for
end for
return best_move

Function print_board (b)
for r in bo do
print (" | " . join(r))
print ("-" * 5)
```

<P.T.O>

```
if _name_ == _main_ do
    b ← [[ EMPTY, EMPTY, EMPTY],
         [ EMPTY, EMPTY, EMPTY],
         [ EMPTY, EMPTY, EMPTY]]
    print ("Initial Board : ")
    print_board (b)
    while True do
        best_move ← Find_best_move (b)
        print (Player X plays <best_move>)
        b[best_move[0]][best_move[1]] ← X
        print_board (b)
        if Evaluate (b) = 1 do
            print (Player X wins)
            break
        elif is_board_full (b)
            print (It's a draw)
            break
        import random
        empty_positions ← [(i, j) for i in range(3) for
                           j in range(3) if b[i][j] =
                           EMPTY]
        if empty_positions do
            o_move ← random. choice (empty_positions)
            print (Player O plays <o_move> )
            b[o_move[0]][o_move[1]] ← O
            print_board (b)
            if evaluate (b) = -1 do
                print (Player O wins)
                break
            elif is_board_full (b) do
                print ("Its a draw")
                break
```

Alpha - Beta Pruning (8- Queens):

```
Function is_safe (b, r, c)
  for i from 0 to r-1 do
     if b[i] = c or abs(b[i]-c) = abs(i-r) do
        return False
     end if
  end for
  Return True

Function alpha_beta (b, r, α, β)
   if r = 8 do
      if is_solution (b)
         print b
         Return True
      end if
      Return False
   end if
   for c from 0 to 7 do
      if is_safe (b, r, c) do
         b[r] ← c
         if alpha_beta (b, r+1, α, β) do
            Return True
         end if
         b[r] ← -1
      end if
      if r % 2 = 0 do
         α ← max (α, value)
      else do
         β ← min (β, value)
      end if
      if α >= β do
```



break
end if
end for
Return False

Function is_solution (b)
Return (no. of distinct values in board == 8)

Function main()
b ← [-1, -1, -1, -1, -1, -1, -1, -1]
$\alpha \leftarrow -\infty$
$\beta \leftarrow -\infty$
alpha_beta (b, 0, α, β)

30/12

Code:

FOL (Forward Chaining)

```
class Fact:
    def __init__(self, predicate, *args):
        self.predicate = predicate
        self.args = tuple(args)
```

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

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

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

class Rule:
    def __init__(self, conditions, conclusion):
        self.conditions = conditions  # A list of Facts
        self.conclusion = conclusion  # A single Fact

    def is_satisfied(self, known_facts):
        return all(condition in known_facts for condition in self.conditions)

    def __str__(self):
        conditions_str = " ^ ".join(str(c) for c in self.conditions)
        return f"{conditions_str} -> {self.conclusion}"

class ForwardChaining:
    def __init__(self):
        self.facts = set()  # Set of known facts
        self.rules = []  # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                if rule.is_satisfied(self.facts) and rule.conclusion not in self.facts:
                    # Printing the logical statement applied when the rule is applied
                    print(f"Applying rule: {rule.conditions} -> {rule.conclusion}")
                    self.facts.add(rule.conclusion)
                    new_facts = True

    def display_facts(self):
        print("\nFinal Set of Statements proving that Robert is a criminal:")
        for fact in self.facts:
            print(f"{fact.predicate.capitalize()} of {', '.join(fact.args)} is true.")

if __name__ == "__main__":
    fc = ForwardChaining()
```

```
    # Hardcoding facts as per the problem statement
    fc.add_fact(Fact("crime", "american", "hostile_nation"))  # It is a crime for an American to sell weapons to a hostile nation
    fc.add_fact(Fact("american", "robert"))  # Robert is an American
    fc.add_fact(Fact("sold_missiles", "robert", "country_a"))  # Robert sold missiles to Country A
    fc.add_fact(Fact("enemy", "country_a", "america"))  # Country A is an enemy of America

    # Rule: If an American sells weapons to a hostile nation, they are a criminal
    conditions = [
        Fact("american", "robert"),
        Fact("sold_missiles", "robert", "country_a"),
        Fact("enemy", "country_a", "america")
    ]
    conclusion = Fact("criminal", "robert")
    fc.add_rule(Rule(conditions, conclusion))

    # Perform inference (forward chaining)
    print("Performing inference...\n")
    fc.infer()

    # Display the results: final set of facts proving Robert is a criminal
    fc.display_facts()
print("Nikhilesh C – 1BM22CS181")
```

Output:

```
Performing inference...

Applying rule: [<__main__.Fact object at 0x000001E54819B640>, <__main__.Fact obj
ect at 0x000001E54819B6D0>, <__main__.Fact object at 0x000001E54819BA60>] -> cri
minal(robert)

Final Set of Statements proving that Robert is a criminal:
Enemy of country_a, america is true.
Crime of american, hostile_nation is true.
Sold_missiles of robert, country_a is true.
Criminal of robert is true.
American of robert is true.
Nikhilesh C - 1BM22CS181
```

MINIMAX (TIC-TAC-TOE):

Code:

```
import math

def minimax(board, depth, is_maximizing_player):
    if game_over(board):
        return evaluate(board)

    if is_maximizing_player:
        best = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
```

```
            best = max(best, minimax(board, depth + 1, False))
            undo_move(board, move)
        return best
    else:
        best = math.inf
        for move in available_moves(board):
            make_move(board, move, 'O')
            best = min(best, minimax(board, depth + 1, True))
            undo_move(board, move)
        return best

def evaluate(board):
    if player_wins(board, 'X'):
        return 1
    if player_wins(board, 'O'):
        return -1
    return 0

def game_over(board):
    return player_wins(board, 'X') or player_wins(board, 'O') or no_more_moves(board)

def available_moves(board):
    moves = []
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                moves.append((row, col))
    return moves

def make_move(board, move, player):
    row, col = move
    board[row][col] = player

def undo_move(board, move):
    row, col = move
    board[row][col] = " "

def player_wins(board, player):
    # Check rows and columns
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for j in range(3)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in
range(3)):
        return True
    return False

def no_more_moves(board):
    return all(board[row][col] != " " for row in range(3) for col in range(3))
```

```
def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = 'X'
    best_move = None

    if current_player == 'X':
        best_score = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
            score = minimax(board, 0, False)
            undo_move(board, move)
            if score > best_score:
                best_score = score
                best_move = move
        make_move(board, best_move, 'X')

    print("Board after the best move:")
    for row in board:
        print(row)

if __name__ == "__main__":
    main()
print("nikhilesh  1bm22cs181")
```

Output :

```
Board after the best move:
['X', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
nikhilesh  1bm22cs181
```

Alpha-beta(8 Queens)

Code:

```
# Function to check if placing a queen at (row, col) is safe
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):  # Check for column and
diagonal conflicts
            return False
    return True

# Backtracking function for N-Queens
def solve_n_queens(board, row):
    if row == 8:  # All queens have been placed
```

```
        print_board(board)  # Print the board if solution is found
        return True

    for col in range(8):  # Try placing a queen in each column of the current row
        if is_safe(board, row, col):  # Check if placing a queen at (row, col) is safe
            board[row] = col  # Place the queen in the current column

            # Recursively attempt to place the next queen in the next row
            if solve_n_queens(board, row + 1):
                return True  # Solution found, propagate up

            board[row] = -1  # Backtrack: Remove the queen from the current position

    return False  # No solution found in the current row and column configurations

# Function to print the board in a readable format
def print_board(board):
    for row in range(8):
        line = ['Q' if board[row] == col else '.' for col in range(8)]
        print(" ".join(line))
    print()

# Main function to start solving the N-Queens problem
def main():
    board = [-1] * 8  # Initialize the board (no queens placed)
    if not solve_n_queens(board, 0):  # Start solving from the first row
        print("No solution found.")

# Call the main function
main()

print("Nikhilesh 1BM22CS181")
```

Output:

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Nikhilesh 1BM22CS181
>>
```