VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

P Manya (1BM22CS187)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **P Manya (1BM22CS187) ,**who is a 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.

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

Index

GITHUB LINK: **https://github.com/pmanya6/AI-LAB**

Program 1-Tic Tac Toe

Algorithm:

24/9/24 LAB PGM-1 TIC-TAC-TOE

ALGORITHM:

1. Create a 3 x 3 matrix where each cell is
empty with default value = -1
2. Take user input as 0 in one cell.
if already filled ask to fill in other cell
which is empty. board [row][col] empty:
3. check if X's or 0's is in consecutive
manner horizontally, vertically or diagonally
{ if board [row][col] is empty:
place the X
end loop }.
else
enter in empty cell.
4. if row: (0,1,2) or (3,4,5) or (6,7,8)
or column: (0,3,6), (1,4,7), (2,5,8)
or diagonal: (0,4,8), (2,4,6)
return True
else
return False
5. while true
if board [row][col] is empty:
Print 0
end loop
6. random

3. After calling check function
If all values of row is 0
return 1
if all values of a col is 0
return 1
if [0][0] [1][1] & [2][2]
return 1
if [0][2] [1][1] & [2][0]
return 1
else return 0.
if the condition is true (1)
print user wins.
4. generate random()
row, col = random.choice (empty cells present)
generate random row mod 3
generate random col mod 3
if mat [i][j] = -1
return 1
else
generate random()
row, col = random.choice (empty cells)
5. When comp wins print comp won
6. If all cells = filled.
Declare draw

Refer Algo

Code:

```
import random

def initialize_board():
    return [[' ' for _ in range(3)] for _ in range(3)]

def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != ' ':
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != ' ':
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
    return None

def available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check_two_in_a_row(board, player):
    for row in range(3):
        if board[row].count(player) == 2 and board[row].count(' ') == 1:
            return row, board[row].index(' ')
    for col in range(3):
```

```
        if [board[row][col] for row in range(3)].count(player) == 2:
            empty_index = [row for row in range(3) if board[row][col] == ' ']
            if empty_index:
                return empty_index[0], col
    if [board[i][i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][i] == ' ']
        if empty_index:
            return empty_index[0], empty_index[0]
    if [board[i][2 - i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][2 - i] == ' ']
        if empty_index:
            return empty_index[0], 2 - empty_index[0]
    return None

def make_move(board, player, move):
    board[move[0]][move[1]] = player

def computer_move(board):
    move = check_two_in_a_row(board, 'O')
    if move:
        make_move(board, 'O', move)
        return
    move = check_two_in_a_row(board, 'X')
    if move:
        make_move(board, 'O', move)
        return
    moves = available_moves(board)
    if moves:
        move = random.choice(moves)
        make_move(board, 'O', move)

def user_move(board):
```

```
    while True:
        try:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
            if board[row][col] == ' ':
                make_move(board, 'X', (row, col))
                return
            else:
                print("That spot is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter numbers between 0 and 2.")

def play_game():
    board = initialize_board()
    players = ['X', 'O']
    current_player = 0
    for _ in range(9):
        display_board(board)
        if current_player == 0:
            user_move(board)
        else:
            computer_move(board)
        winner = check_winner(board)
        if winner:
            display_board(board)
            print(f"Player {winner} wins!")
            return
        current_player = 1 - current_player
    display_board(board)
    print("It's a draw!")

play_game()
```

```
print("P Manya")
print("1BM22CS187")
```

Output:

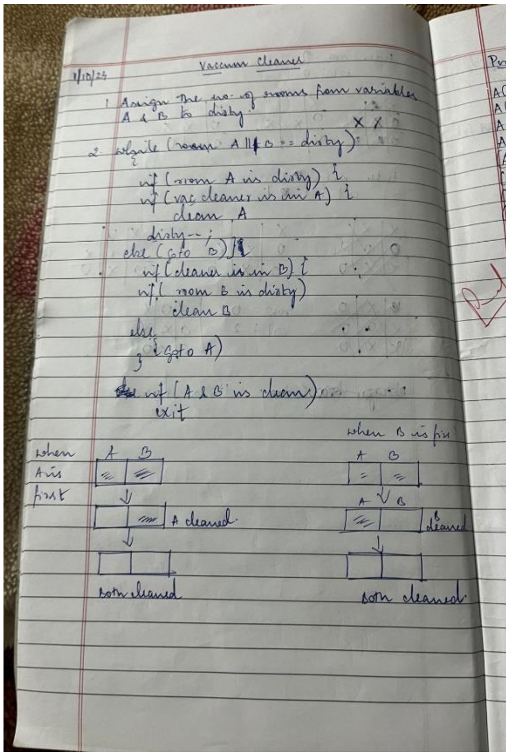
```
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2): 0
Enter column (0-2): 0
X| |
-----
 | |
-----
 | |
-----
X| |
-----
O| |
-----
 | |
-----
Enter row (0-2): 2
Enter column (0-2): 0
X| |
-----
O| |
-----
X| |
-----
```

```
-----
X| |
-----
O| |
-----
X|O|
-----
Enter row (0-2): 2
Enter column (0-2): 2
X| |
-----
O| |
-----
X|O|X
-----
X| |
-----
O|O|
-----
X|O|X
-----
Enter row (0-2): 1
Enter column (0-2): 1
That spot is already taken. Try again.
Enter row (0-2): 1
Enter column (0-2): 2
X| |
-----
O|O|X
```

```
X| |
-----
O|O|X
-----
X|O|X
-----
X|O|
-----
O|O|X
-----
X|O|X
-----
Player O wins!
P Manya
1BM22CS187
```

Program 2 - Vacuum Cleaner

Algorithm:

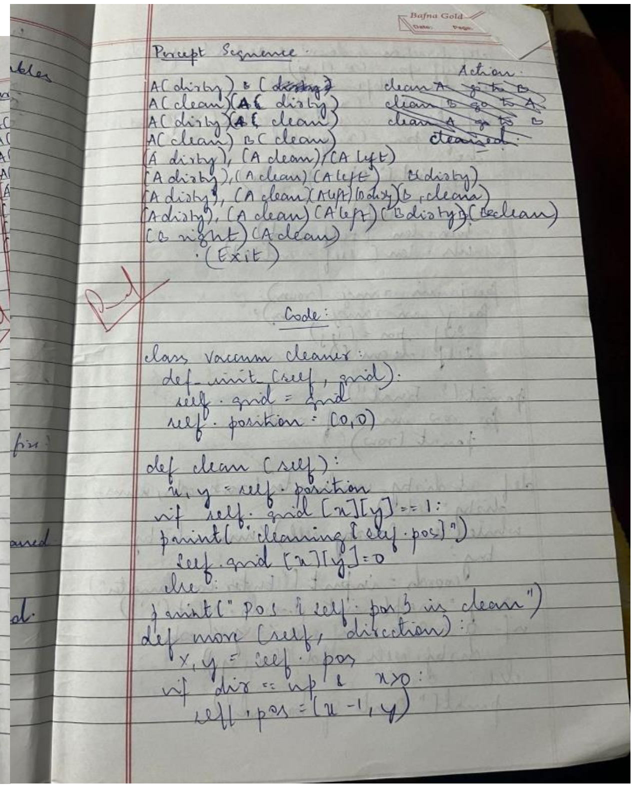


1/10/24 Vacuum Cleaner

1. Assign the no. of rooms from variables A & B to dirty
2. while (room A || B == dirty):
   if (room A is dirty) {
   if (vac cleaner is in A) {
   clean A
   dirty--;
   else (goto B)}
   if (cleaner is in B) {
   if (room B is dirty)
   clean B
   else
   } (goto A)
   if (A & B is clean)
   exit

when A is first
A B
A cleaned
both cleaned

when B is first
A B
B cleaned
both cleaned



Percept Sequence: Action:

A(dirty) & (dirty) clean A ~~go to B~~
A(clean) (A (dirty) clean B ~~go to A~~
A(dirty) (A (clean) clean A ~~go to B~~
A(clean) B(clean) ~~cleaned~~
(A dirty), (A clean) (A left)
(A dirty), (A clean) (A left) (B dirty)
(A dirty), (A clean) (A left) (B dirty) (B clean)
(A dirty), (A clean) (A left) (B dirty) (B clean)
(B right) (A clean)
(Exit)

Code:

```
class vacuum cleaner:
  def init (self, grid):
    self.grid = grid
    self.position = (0,0)

def clean (self):
  x, y = self.position
  if self.grid [x][y] == 1:
  print(" cleaning {self.pos}")
    self.grid [x][y] = 0
  else:
  } print(" pos {self.pos} is clean")
def move (self, direction):
  x, y = self.pos
  if dir == up & x>0:
    self.pos = (x -1, y)
```

Code:

```
class VacuumCleaner:
    def __init__(self, grid):
        self.grid = grid
        self.position = (0, 0)

    def clean(self):
        x, y = self.position
        if self.grid[x][y] == 1:
            print(f"Cleaning position {self.position}")
            self.grid[x][y] = 0
        else:
            print(f"Position {self.position} is already clean")

    def move(self, direction):
        x, y = self.position
        if direction == 'up' and x > 0:
            self.position = (x - 1, y)
        elif direction == 'down' and x < len(self.grid) - 1:
            self.position = (x + 1, y)
        elif direction == 'left' and y > 0:
            self.position = (x, y - 1)
        elif direction == 'right' and y < len(self.grid[0]) - 1:
            self.position = (x, y + 1)
        else:
            print("Move not possible")

    def run(self):
        rows = len(self.grid)
        cols = len(self.grid[0])
        for i in range(rows):
            for j in range(cols):
```

```
                self.position = (i, j)
                self.clean()

        print("Final grid state:")
        for row in self.grid:
            print(row)

def get_dirty_coordinates(rows, cols, num_dirty_cells):
    dirty_cells = set()
    while len(dirty_cells) < num_dirty_cells:
        try:
            coords = input(f"Enter coordinates for dirty cell {len(dirty_cells) + 1} (format: row,col): ")
            x, y = map(int, coords.split(','))
            if 0 <= x < rows and 0 <= y < cols:
                dirty_cells.add((x, y))
            else:
                print("Coordinates are out of bounds. Try again.")
        except ValueError:
            print("Invalid input. Please enter coordinates in the format: row,col")
    return dirty_cells

rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))
num_dirty_cells = int(input("Enter the number of dirty cells: "))

if num_dirty_cells > rows * cols:
    print("Number of dirty cells exceeds total cells in the grid. Adjusting to maximum.")
    num_dirty_cells = rows * cols

initial_grid = [[0 for _ in range(cols)] for _ in range(rows)]
dirty_coordinates = get_dirty_coordinates(rows, cols, num_dirty_cells)
```

```
for x, y in dirty_coordinates:
    initial_grid[x][y] = 1

vacuum = VacuumCleaner(initial_grid)

print("Initial grid state:")
for row in initial_grid:
    print(row)

vacuum.run()
print(“P Manya”)
print(“1BM22CS187”)
```

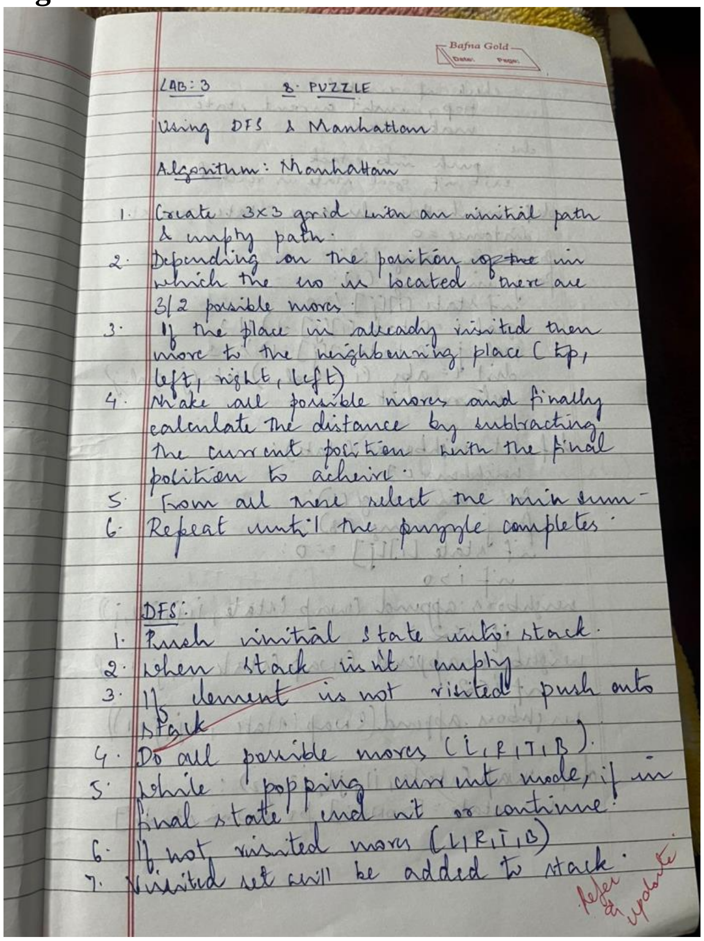
Output:

```
Output
Enter the number of rows: 2
Enter the number of columns: 2
Enter the number of dirty cells: 1
Enter coordinates for dirty cell 1 (format: row,col): 0,1
Initial grid state:
[0, 1]
[0, 0]
Position (0, 0) is already clean
Cleaning position (0, 1)
Position (1, 0) is already clean
Position (1, 1) is already clean
Final grid state:
[0, 0]
[0, 0]
P Manya
1BM22CS187

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

ss

Program 3 - 8 Puzzle Game Using DFS

Algorithm:



Bafna Gold

Date: Page:

LAB: 3 8. PUZZLE

Using DFS & Manhattan

Algorithm: Manhattan

1. Create 3x3 grid with an initial path & empty path.
2. Depending on the position ~~of the~~ in which the no is located there are 3/2 possible moves.
3. If the place is already visited then move to the neighbouring place (top, left, right, left)
4. Make all possible moves and finally calculate the distance by subtracting the current position with the final position to achieve.
5. From all these select the min sum.
6. Repeat until the puzzle completes.

DFS:

1. Push initial state into stack.
2. When stack isn't empty
3. If element is not visited push onto stack
4. Do all possible moves (L, R, T, B).
5. While popping current node, if in final state end it or continue.
6. If not visited move (L, R, T, B)
7. Visited set will be added to stack.

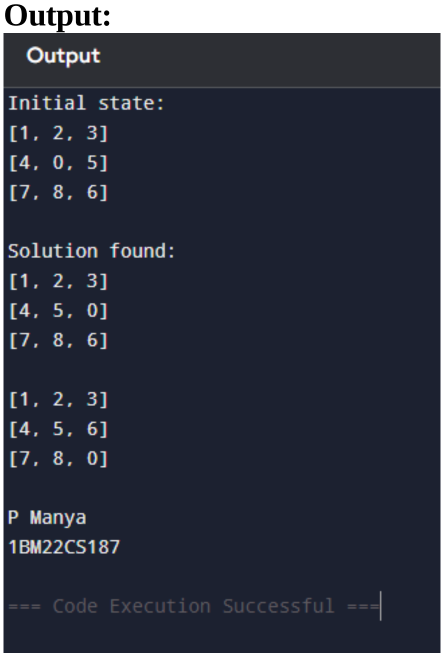
Refer & update

Code:

```
class PuzzleState:
    def __init__(self, board, empty_pos, moves=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.moves = moves
    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]
    def get_possible_moves(self):
        x, y = self.empty_pos
        moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                 new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny], new_board[x * 3
+ y]
                moves.append((new_board, (nx, ny)))
        return moves
def dfs(initial_state):
    stack, visited = [initial_state], set()
    while stack:
        current_state = stack.pop()
        if current_state.is_goal():
            return current_state.moves
        visited.add(tuple(current_state.board))
        for new_board, new_empty_pos in current_state.get_possible_moves():
            new_state = PuzzleState(new_board, new_empty_pos, current_state.moves + [new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None
def print_matrix(board):
```

```
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()
def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (empty_pos // 3, empty_pos % 3))
    print("Initial state:")
    print_matrix(initial_board)
    solution = dfs(initial_state)
    if solution:
        print("Solution found:")
        for step in solution:
            print_matrix(step)
    else:
        print("No solution found.")
if __name___== "__main__":
    main()
print("P Manya")
print("1BM22CS187")
```

Output:



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

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

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

P Manya
1BM22CS187

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

8 Puzzle Game Using Manhattan Distance:

Algorithm:

Bafna Gold
Date: Page:

LAB: 3 8. PUZZLE

Using DFS & Manhattan

Algorithm: Manhattan

1. Create 3x3 grid with an initial path & empty path.
2. Depending on the position of the in which the no is located there are 3/2 possible moves.
3. If the place is already visited then move to the neighbouring place (top, left, right, left)
4. Make all possible moves and finally calculate the distance by subtracting the current position with the final position to achieve.
5. From all move select the min sum.
6. Repeat until the puzzle completes.

DFS:

1. Push initial state into stack.
2. When stack is not empty
3. If element is not visited push onto stack
4. Do all possible moves (L, R, T, B).
5. While popping current node, if in final state end it or continue.
6. If not visited move (L, R, T, B)
7. Visited set will be added to stack.

Refer & update

Code:

```
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i = (state[i][j] - 1) // 3
                goal_j = (state[i][j] - 1) % 3
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def get_neighbors(state):
    i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
    moves = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
    return [swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

def swap(state, i1, j1, i2, j2):
    new_state = [row[:] for row in state]
    new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
    return new_state

def dfs_with_manhattan(state, goal, visited=set()):
    if state == goal:
        return [state]
    visited.add(str(state))
    neighbors = sorted(get_neighbors(state), key=lambda x: manhattan_distance(x, goal))
    for neighbor in neighbors:
        if str(neighbor) not in visited:
            path = dfs_with_manhattan(neighbor, goal, visited)
            if path:
                return [state] + path
    return None
```

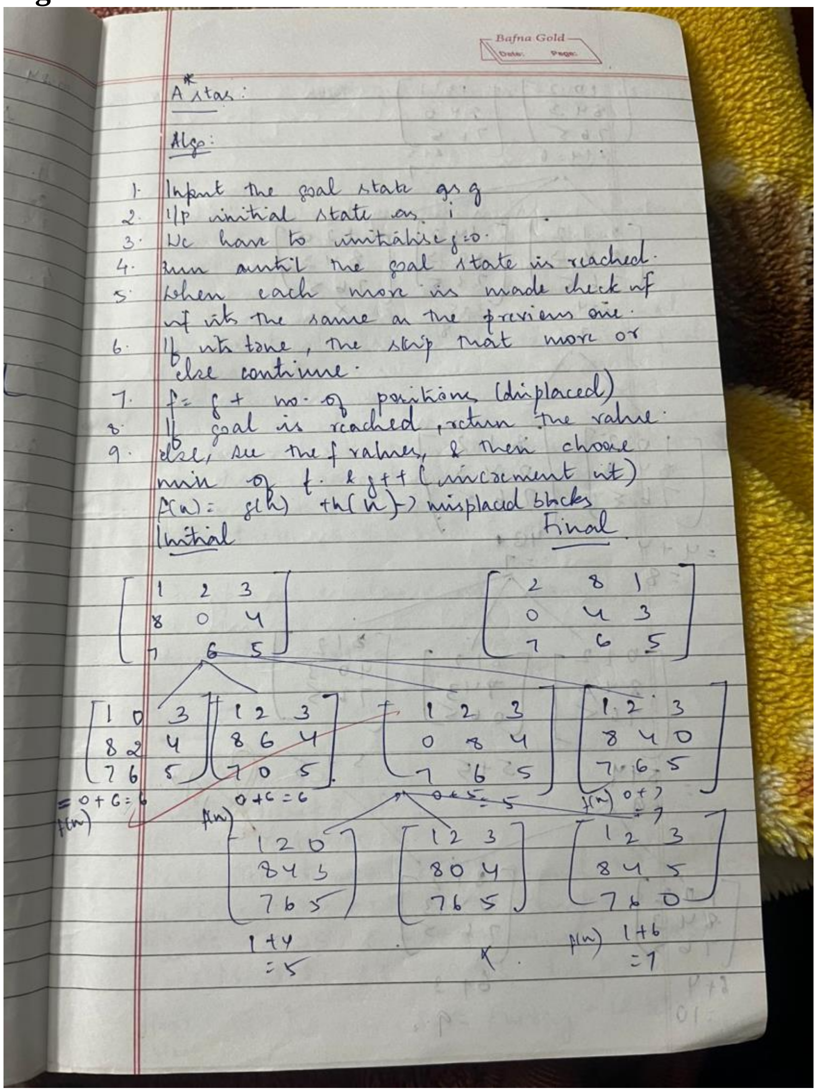
```
# Take user input for initial state
initial_state = [[int(x) for x in input(f"Enter row {i+1}: ").split()] for i in range(3)]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

solution = dfs_with_manhattan(initial_state, goal_state)
if solution:
    print("Solution found:")
    for state in solution:
        print(*state, sep='\n', end='\n\n')
else:
    print("No solution found.")
print("P Manya")
print("1BM22CS187")
```

Output:

```
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]

P Manya
1BM22CS187

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

Program 4 - 8 Puzzle Game Using A*

Algorithm:



Code:

```
import heapq


# Goal state where blank (0) is the first tile
goal_state = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
]
```

```python
# Helper functions
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):
```

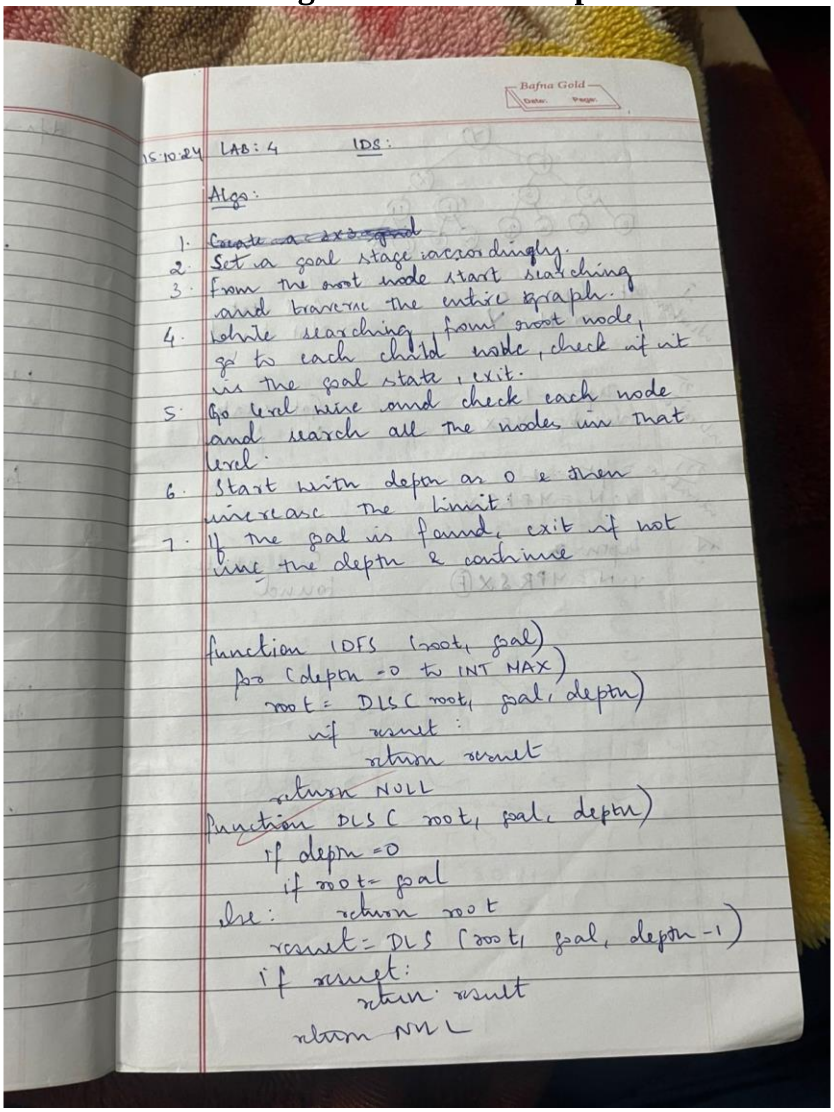
```python
    for row in puzzle:
        print(row)
    print()

def a_star_misplaced_tiles(initial_state):
    # Priority queue (min-heap) and visited states
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()

    while frontier:
        f, g, current_state, path = heapq.heappop(frontier)
        # Print the current state
        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  # No solution found

# Initial puzzle state
```

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

solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print("P Manya")
print("1BM22CS187")
```

Output:

```
Output
[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!
P Manya
1BM22CS187

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

8 Puzzle Game Using IDDFS On a Graph



Bafna Gold

Date: Page:

15.10.24 LAB: 4 IDS:

Algo:

1. ~~Create a 3x3 grid~~
2. Set a goal stage accordingly.
3. From the root node start searching and traverse the entire graph.
4. While searching from root node, go to each child node, check if it is the goal state, exit.
5. Go level wise and check each node and search all the nodes in that level.
6. Start with depth as 0 & then increase the limit.
7. If the goal is found, exit if not inc the depth & continue

```
function IDFS (root, goal)
  for (depth = 0 to INT MAX)
     root = DLS( root, goal, depth)
       if result :
           return result
     return NULL
function DLS ( root, goal, depth)
   if depth = 0
     if root = goal
else:      return root
   result = DLS (root, goal, depth -1)
  if result:
        return result
   return NULL
```

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(“P Manya”)

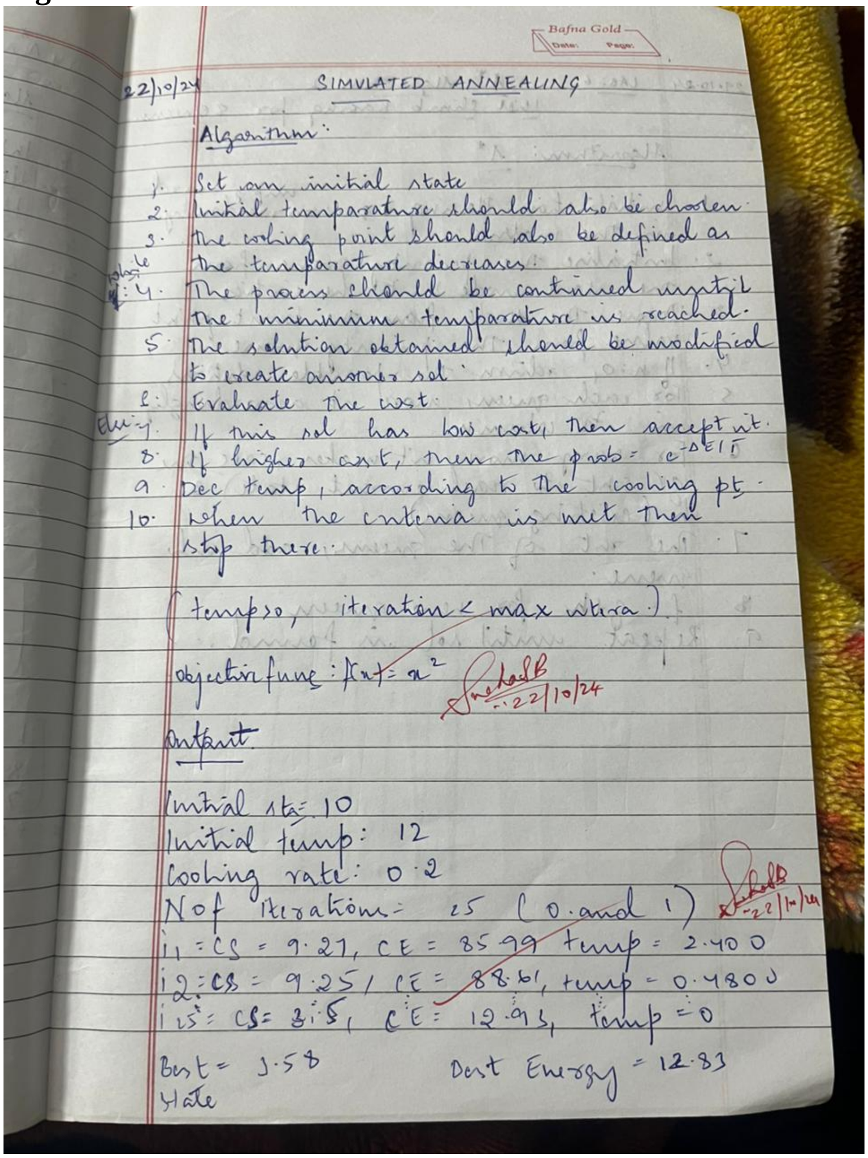
print(“1BM22CS187”)

Output:

```
Output
Enter the number of edges: 14
Enter an edge (format: A B): y p
Enter an edge (format: A B): y x
Enter an edge (format: A B): p r
Enter an edge (format: A B): p s
Enter an edge (format: A B): x f
Enter an edge (format: A B): x h
Enter an edge (format: A B): r b
Enter an edge (format: A B): r c
Enter an edge (format: A B): s x
Enter an edge (format: A B): s z
Enter an edge (format: A B): f u
Enter an edge (format: A B): f e
Enter an edge (format: A B): h l
Enter an edge (format: A B): h w
Enter the start node: y
Enter the goal node: f
Enter the maximum depth for IDDFS: 3
Goal node f found!
P Manya
1BM22CS187

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

Program 5 - Simulated Annealing Algorithm

Algorithm:



Bafna Gold
Date: Page:

22/10/24 SIMULATED ANNEALING

Algorithm:

1. Set an initial state
2. Initial temparature should also be chosen.
3. The cooling point should also be defined as the temparature decreases.
4. The process should be continued until the minimum temparature is reached.
5. The solution obtained should be modified to create another sol.
6. Evaluate the cost.
7. If this sol has low cost, then accept it.
8. If higher cost, then the prob = $e^{-\Delta E/T}$
9. Dec temp, according to the cooling pt.
10. When the criteria is met then stop there.

(temp > 0, iteration < max itera.)

objective func: $f(x) = x^2$

Output:

Initial sta = 10
Initial temp: 12
Cooling rate: 0.2
N of iterations = 25 (0 and 1)
i1 = CS = 9.27, CE = 85.99 temp = 2.400
i2 = CS = 9.25, CE = 88.61, temp = 0.4800
i25 = CS = 3.5, CE = 12.93, temp = 0

Best = 3.58 Best Energy = 12.83
State

Code:

```
import numpy as np
import math
import random

def objective_function(x):
    """Objective function to minimize: f(x) = x^2"""
    return x ** 2

def simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations):
    """Simulated Annealing algorithm to find the minimum of the objective function."""
    current_state = initial_state
    current_energy = objective_function(current_state)
    best_state = current_state
    best_energy = current_energy
    temp = initial_temp

    for iteration in range(max_iterations):
        # Generate a new candidate state by perturbing the current state
        candidate_state = current_state + random.uniform(-1, 1)
        candidate_energy = objective_function(candidate_state)

        # Calculate energy difference
        energy_diff = candidate_energy - current_energy

        # If the candidate state is better, or accepted with a certain probability
        if energy_diff < 0 or random.uniform(0, 1) < math.exp(-energy_diff / temp):
            current_state = candidate_state
            current_energy = candidate_energy

        # Update best state found
        if current_energy < best_energy:
```

```
            best_state = current_state
            best_energy = current_energy

        # Cool down the temperature
        temp *= cooling_rate

        # Print the current state and temperature for debugging
        print(f"Iteration {iteration + 1}: Current State = {current_state:.4f}, Current Energy = {current_energy:.4f}, Temperature = {temp:.4f}")

    return best_state, best_energy

# Get user input for parameters
try:
    initial_state = float(input("Enter the initial state (starting point): "))
    initial_temp = float(input("Enter the initial temperature: "))
    cooling_rate = float(input("Enter the cooling rate (between 0 and 1): "))
    max_iterations = int(input("Enter the number of iterations: "))

    # Validate cooling rate
    if cooling_rate <= 0 or cooling_rate >= 1:
        raise ValueError("Cooling rate must be between 0 and 1.")

    # Execute the simulated annealing algorithm
    best_state, best_energy = simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations)

    # Output the best state and energy found
    print(f"Best State: {best_state:.4f}, Best Energy: {best_energy:.4f}")

except ValueError as e:
    print(f"Invalid input: {e}")
```

print("P Manya")

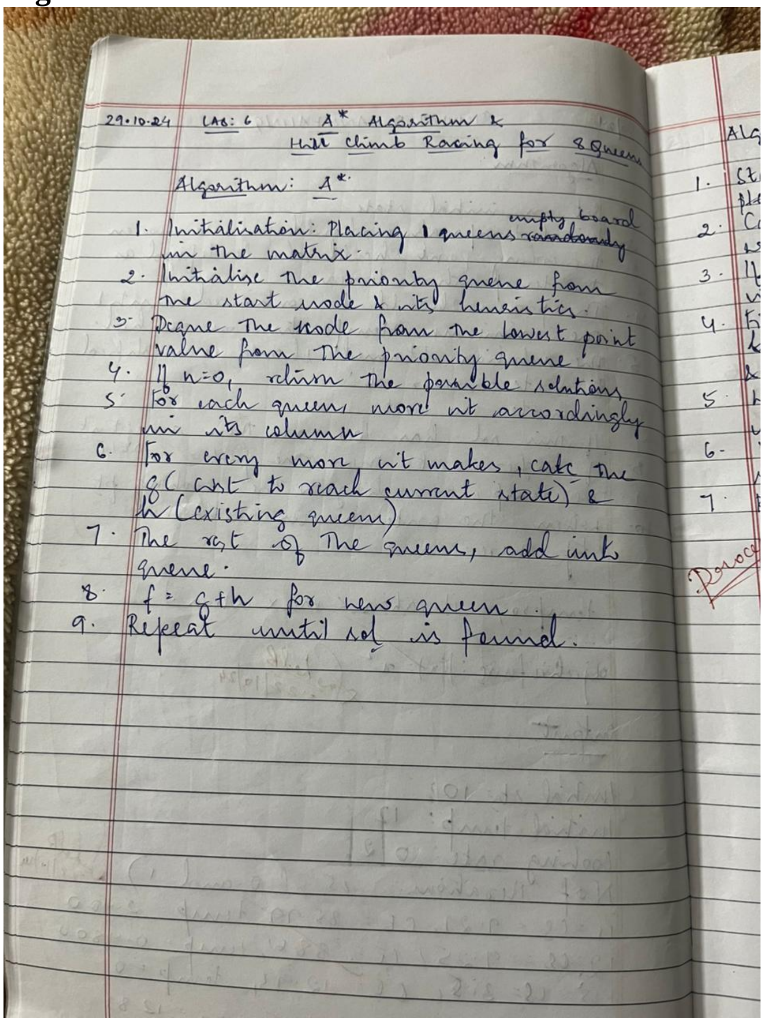
print("1BM22CS187")

Output:

```
Output                                                        Clear
Enter the initial state (starting point): 10
Enter the initial temperature: 12
Enter the cooling rate (between 0 and 1): 0.3
Enter the number of iterations: 20
Iteration 1: Current State = 9.7246, Current Energy = 94.5682, Temperature = 3.6000
Iteration 2: Current State = 9.7246, Current Energy = 94.5682, Temperature = 1.0800
Iteration 3: Current State = 9.4551, Current Energy = 89.3990, Temperature = 0.3240
Iteration 4: Current State = 8.9080, Current Energy = 79.3530, Temperature = 0.0972
Iteration 5: Current State = 8.9080, Current Energy = 79.3530, Temperature = 0.0292
Iteration 6: Current State = 8.9080, Current Energy = 79.3530, Temperature = 0.0087
Iteration 7: Current State = 8.0594, Current Energy = 64.9534, Temperature = 0.0026
Iteration 8: Current State = 7.3893, Current Energy = 54.6018, Temperature = 0.0008
Iteration 9: Current State = 7.3893, Current Energy = 54.6018, Temperature = 0.0002
Iteration 10: Current State = 7.3893, Current Energy = 54.6018, Temperature = 0.0001
Iteration 11: Current State = 7.3893, Current Energy = 54.6018, Temperature = 0.0000
Iteration 12: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 13: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 14: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 15: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 16: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 17: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 18: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 19: Current State = 7.3587, Current Energy = 54.1509, Temperature = 0.0000
Iteration 20: Current State = 6.7606, Current Energy = 45.7062, Temperature = 0.0000
Best State: 6.7606, Best Energy: 45.7062
P Manya
1BM22CS187
```

Program 6 - Implementing A* on 8 Queens

Algorithm:



29.10.24 LAB: 6 A* Algorithm & Hill Climb Racing for 8 Queens

Algorithm: A*

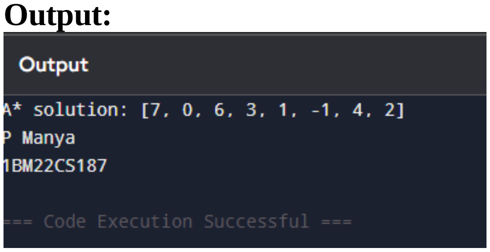
1. Initialisation: Placing 1 queens ~~randomly~~ empty board in the matrix.
2. Initialise the priority queue from the start node & its heuristics.
3. Dequeue the node from the lowest point value from the priority queue
4. If h=0, return the possible solutions
5. For each queen, move it accordingly in its column
6. For every move it makes, calc the g(cost to reach current state) & h (existing queen)
7. The rest of the queens, add into queue.
8. f = g+h for new queen
9. Repeat until sol is found.

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 = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
```

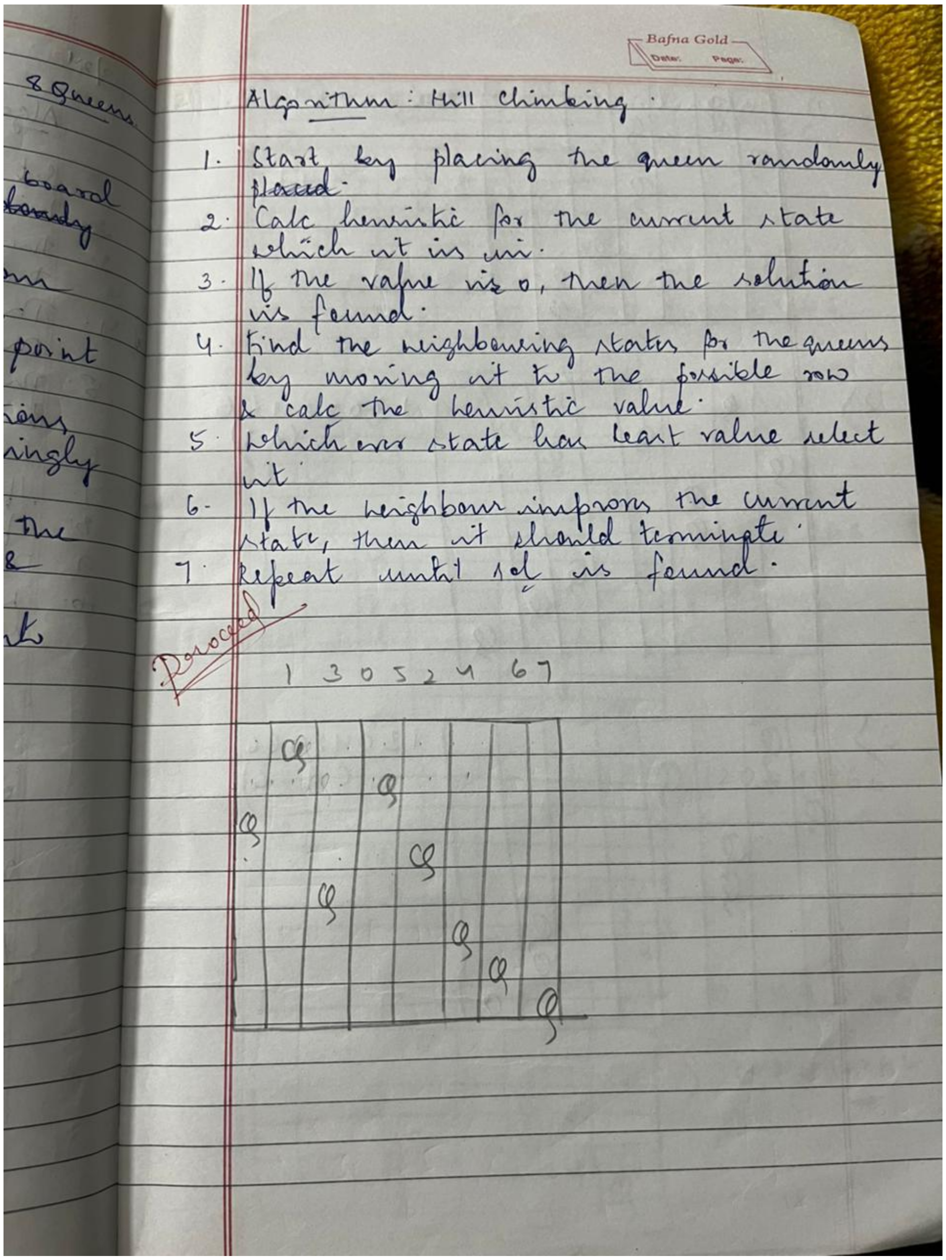
```
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

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

        for col in range(8):
            if current_state[col] == -1:  # Only place a queen if none is present in this column
                for row in range(8):
                    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

solution = a_star_8_queens()
print("A* solution:", solution)
print("P Manya")
print("1BM22CS187")
```

Output:



Implementing Hill Climbing on 8 Queens

Algorithm:


Algorithm: Hill climbing.

1. Start by placing the queen randomly ~~placed~~.
2. Calc heuristic for the current state which it is in.
3. If the value is 0, then the solution is found.
4. Find the neighbouring states for the queens by moving it to the possible row & calc the heuristic value.
5. Which ever state has least value select it.
6. If the neighbour improves the current state, then it should terminate.
7. Repeat until sol is found.

Proceed

1 3 0 5 2 4 6 7

Code:

```
import random

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 hill_climbing_8_queens():
    # Random initial state
    state = [random.randint(0, 7) for _ in range(8)]
    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
```

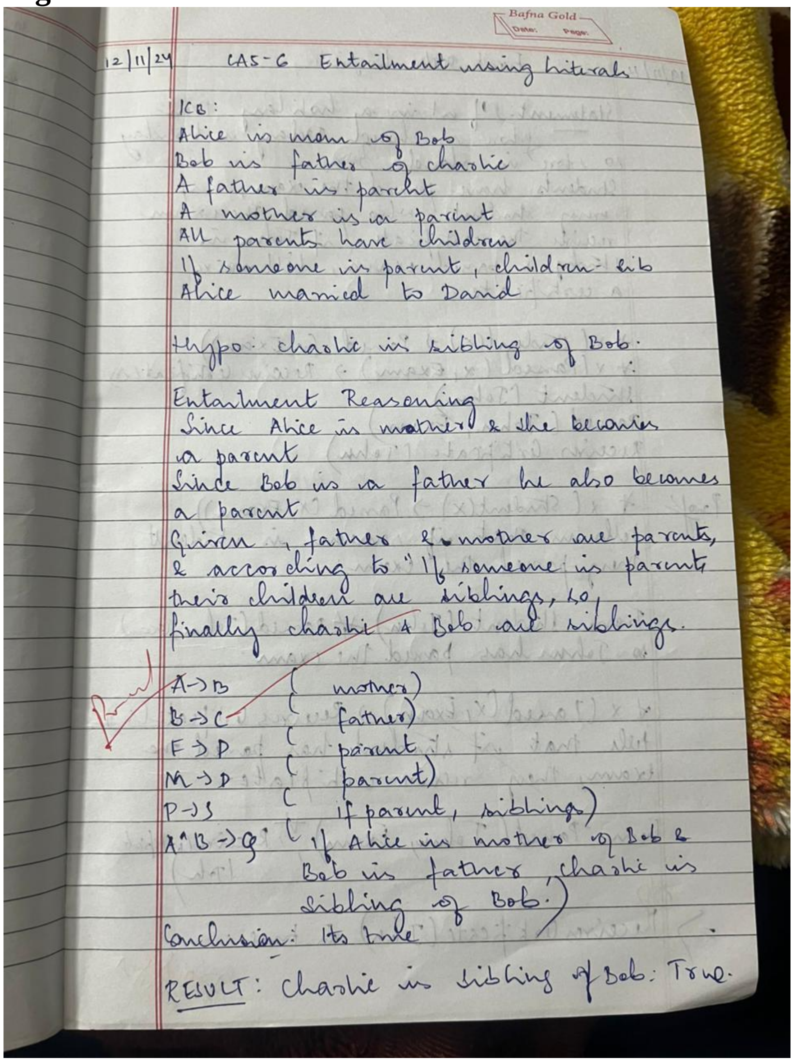
```
        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum

        state = next_state

def hill_climbing_with_random_restarts(max_restarts=100):
    for _ in range(max_restarts):
        solution = hill_climbing_8_queens()
        if solution:
            return solution
    return None  # No solution found after max_restarts

solution = hill_climbing_with_random_restarts()
if solution:
    print("Hill Climbing solution:", solution)
else:
    print("No solution found after maximum restarts.")

print("P Manya")
print("1BM22CS187")
```

Output:

```
Output
Hill Climbing solution: [1, 3, 5, 7, 2, 0, 6, 4]
P Manya
1BM22CS187

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

Program 7 - Entailment Using Literals

Algorithm:



Bafna Gold
Date: Page:

12/11/24 CAS-6 Entailment using literals

KB:
Alice is mom of Bob
Bob is father of charlie
A father is parent
A mother is a parent
All parents have children
If someone is parent, children sib
Alice married to David

Hypo: charlie is sibling of Bob.

Entailment Reasoning
Since Alice is mother & she becomes a parent
Since Bob is a father he also becomes a parent
Given, father & mother are parents, & according to "If someone is parent their children are siblings, so, finally charlie & Bob are siblings.

A->B (mother)
B->C (father)
F->P (parent
M->P (parent)
P->S (if parent, siblings)
A^B->Q (if Alice is mother of Bob & Bob is father, charlie is sibling of Bob.)

Conclusion: Its true

RESULT: Charlie is sibling of Bob: True.

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 rule: ").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("P Manya")
print("1BM22CS187")
```

Output:

```
Output                                                    Clear
Enter rule: Alice is the mother of Bob.
Enter rule: Bob is the father of Charlie.
Enter rule: A father is a parent.
Enter rule: A mother is a parent.
Enter rule: If someone is a parent, their children are siblings.
Enter rule: All parents have children.
Enter rule: Alice is married to David.
Enter rule: done

Current Knowledge Base:
Mother: [('Alice', 'Bob')]
Father: [('Bob', 'Charlie')]
ParentRule: [('Parents', 'HasChildren')]
ParentSiblingRule: [('Parent', 'Siblings')]
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.
P Manya
1BM22CS187

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

Program 8 - FOL using Unification

Algorithm:

19/11/24 First Order Logic

Statement: If it is a holiday, then store is closed. Today is holiday so store is closed. }×

Students have passed the exam, & if any student who passed the exam receive the certificate, if John is study & passed exam, will receive a certificate.

∀ x(Student(x) → Passed (x, Exam))
∀ x(Passed (x, Exam) → Receive Certificate(x))
Student (John)
Passed (John, Exam)
Receives Certificate (John)

Proof: ∀ x (Student(x) → Passed (X, Exam))
tells us that if someone is student they passed the exam.

From Student (John) ∀ Passed (John, Exam)
so John has passed the exam.

∀ x (Passed (x, Exam) → Receives Certificate (x))
tells that if student has passed the exam, they receive certificate.

From Passed (John, Exam), $\xrightarrow{\forall}$ Receives Certificate (John)

O/P
⇒ Receives Certificate (John) is true.

19/11/24

1. Two t
2. check
3. check
4. One
5. Both
6. Recei

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
```

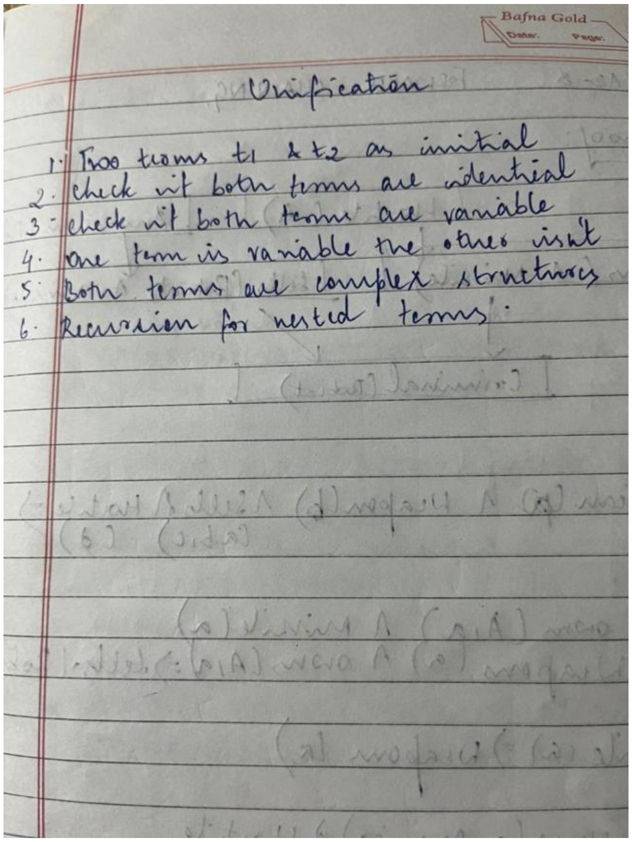
```
# Function to deduce the goal using given rules
def deduct(rules, goal):
    # Try to find unification for the goal from the rules
    for rule in rules:
        if unify(rule, goal):
            print(f"Unification successful: {rule} matches with {goal}.")
            return True
    return False

# 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("P Manya")
print("1BM22CS187")
```

Output:

```
Conclusion: ReceiveCertificate(John) is true.
P Manya
1BM22CS187


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

Program 9 – Unification



Unification

1. Two terms t1 & t2 as initial
2. check if both terms are identical
3. check if both terms are variable
4. One term is variable the other isn't
5. Both terms are complex structures
6. Recursion for nested terms.

Code:

```
def is_variable(term):
    """
    Check if a term is a variable.
    Variables are typically single lowercase letters.
    """
    return isinstance(term, str) and term.islower()


def unify(expr1, expr2, subst={}):
    """
    Unify two expressions expr1 and expr2 under the given substitution subst.
    """
    if subst is None:
        return None  # Failure case
```

```
    if expr1 == expr2:
        return subst  # Expressions are identical

if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    if isinstance(expr1, tuple) and isinstance(expr2, tuple):
        if len(expr1) != len(expr2):
            return None  # Different arity
        # Recursively unify each component
        for arg1, arg2 in zip(expr1, expr2):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None  # Failure
        return subst
    return None  # No unification possible


def unify_variable(var, term, subst):
    """
    Unify a variable with a term, updating the substitution.
    """
    if var in subst:
        return unify(subst[var], term, subst)  # Apply substitution to var
    if term in subst:
        return unify(var, subst[term], subst)  # Apply substitution to term
    if occurs_check(var, term, subst):
        return None  # Circular substitution detected
    # Add var -> term to the substitution
    subst = subst.copy()
    subst[var] = term
    return subst
```

```
def occurs_check(var, term, subst):
    """

    Check if var occurs in term (directly or indirectly) to prevent circular substitutions.
    """
    if var == term:
        return True
    if isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    if term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def parse_input(expr):
    """
    Parse user input into a structured format (nested tuples for functions and terms).
    Example: "f(X, g(y))" -> ('f', 'X', ('g', 'y'))
    """
    expr = expr.strip()
    if '(' not in expr:
        return expr  # Simple variable or constant
    func_name = expr[:expr.index('(')].strip()
    args = expr[expr.index('(') + 1:expr.rindex(')')].split(',')
    args = [parse_input(arg.strip()) for arg in args]
    return (func_name, *args)

def format_output(expr):
    """
    Convert the nested tuple representation back into a string for output.
    Example: ('f', 'X', ('g', 'y')) -> "f(X, g(y))"
    """
```

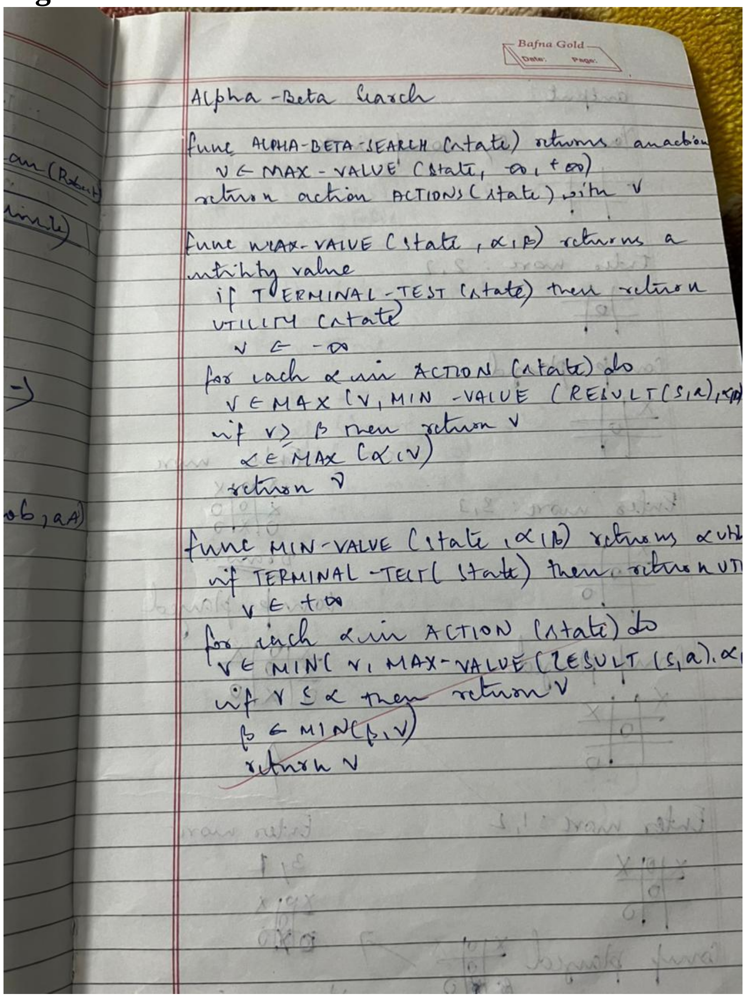
```
    if isinstance(expr, str):
        return expr
    return f"{expr[0]}({', '.join(format_output(arg) for arg in expr[1:])})"

# Main Program
if _name___== "_main_":
    print("Enter the first term:")
    expr1 = parse_input(input().strip())
    print("Enter the second term:")
    expr2 = parse_input(input().strip())
    print("Unifying. .... ")
    result = unify(expr1, expr2)
    if result is None:
        print("Unification failed")
    else:
        print("Unification succeeded with substitution:")
        for var, term in result.items():
            print(f"{var} -> {format_output(term)}")


print("P Manya")
print("1BM22CS187")
```

Output:

```
Output
Enter the first term:
f(X, g(y))
Enter the second term:
f(a, h(x))
Unifying.
Unification succeeded with substitution:
a -> X
g -> h
y -> x
P Manya
1BM22CS187

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

Program 10 - Tic Tac Toe using Min-Max.

Algorithm:



Alpha-Beta Search

func ALPHA-BETA-SEARCH (state) returns an action
v ← MAX-VALUE (state, -∞, +∞)
return action ACTIONS (state), with v

func MAX-VALUE (state, α, β) returns a utility value
if TERMINAL-TEST (state) then return UTILITY (state)
v ← -∞
for each a in ACTION (state) do
v ← MAX (v, MIN-VALUE (RESULT(s,a), α, β)
if v ≥ β then return v
α ← MAX (α, v)
return v

func MIN-VALUE (state, α, β) returns a util
if TERMINAL-TEST (state) then return UT
v ← +∞
for each a in ACTION (state) do
v ← MIN(v, MAX-VALUE (RESULT (s,a), α,
if v ≤ α then return v
β ← MIN(β, v)
return v

Code:

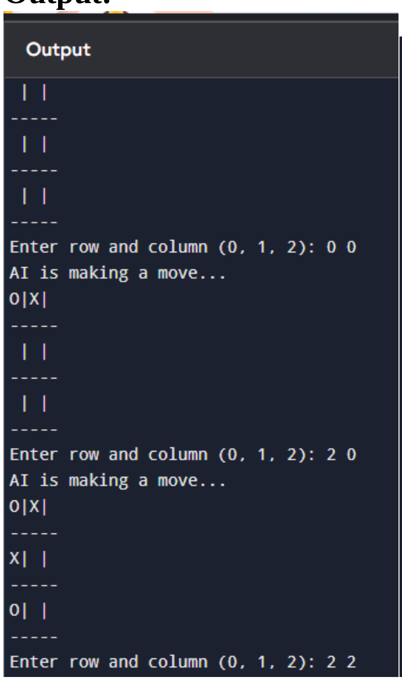
```
import math
# Constants for players
HUMAN = 'O'  # Minimizer
AI = 'X'     # Maximizer
# Initialize empty board
def create_board():
    return [[' ' for _ in range(3)] for _ in range(3)]
# Check if there are any moves left on the board
def is_moves_left(board):
    for row in board:
        if ' ' in row:
            return True
    return False
# Check for a win condition
def evaluate(board):
    # Rows, columns, diagonals check
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != ' ':
            return 1 if row[0] == AI else -1
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != ' ':
            return 1 if board[0][col] == AI else -1
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return 1 if board[0][0] == AI else -1
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return 1 if board[0][2] == AI else -1
    return 0  # No winner
# Minimax algorithm with Alpha-Beta Pruning
def minimax(board, depth, is_maximizing, alpha, beta):
    score = evaluate(board)
    # Terminal condition
```

```
    if score == 1:  # AI wins
        return score - depth  # Prefer quicker wins
    if score == -1:  # Human wins
        return score + depth  # Prefer slower losses
    if not is_moves_left(board):  # Draw
        return 0
    if   is_maximizing:
        best  =  -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = AI
                    best = max(best, minimax(board, depth + 1, False, alpha, beta))
                    board[i][j] = ' '
                    alpha = max(alpha, best)
                    if beta <= alpha:
                        break
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = HUMAN
                    best = min(best, minimax(board, depth + 1, True, alpha, beta))
                    board[i][j] = ' '
                    beta = min(beta, best)
                    if beta <= alpha:
                        break
        return best
# Find the best move for the AI
def find_best_move(board):
```

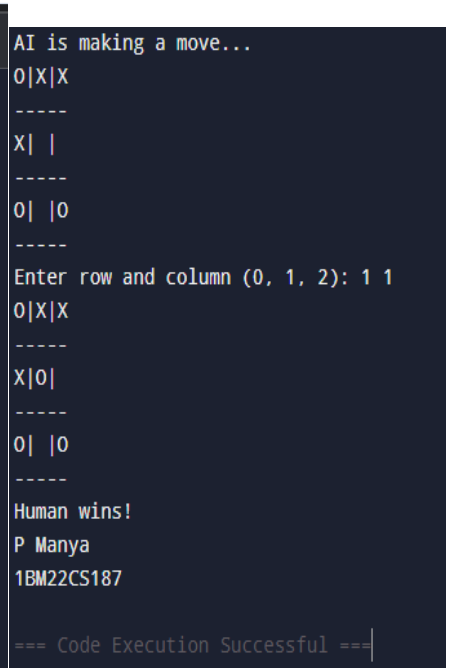
```
    best_val = -math.inf
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = AI
                move_val = minimax(board, 0, False, -math.inf, math.inf)
                board[i][j] = ' '
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move
# Print the board
def print_board(board):
    for row in board:
        print('|'.join(row))
    print('-' * 5)
# Example usage
if __name___== '__main__':
    board = create_board()
    while is_moves_left(board):
        print_board(board)
        # Human makes a move
        row, col = map(int, input("Enter row and column (0, 1, 2): ").split())
        if board[row][col] == ' ':
            board[row][col] = HUMAN
        else:
            print("Invalid move! Try again.")
            continue
        if evaluate(board) != 0 or not is_moves_left(board):
            break
        # AI makes a move
```

```
        print("AI is making a move...")
        ai_move = find_best_move(board)
        board[ai_move[0]][ai_move[1]] = AI
        if evaluate(board) != 0 or not is_moves_left(board):
            break

    # Final result
    print_board(board)
    result = evaluate(board)
    if result == 1:
        print("AI wins!")
    elif result == -1:
        print("Human wins!")
    else:
        print("It's a draw!") print("P
Manya") print("1BM22CS187")
```

Output:



```
Output
 | |
-----
 | |
-----
 | |
-----
Enter row and column (0, 1, 2): 0 0
AI is making a move...
O|X|
-----
 | |
-----
 | |
-----
Enter row and column (0, 1, 2): 2 0
AI is making a move...
O|X|
-----
X| |
-----
O| |
-----
Enter row and column (0, 1, 2): 2 2
```



```
AI is making a move...
O|X|X
-----
X| |
-----
O| |O
-----
Enter row and column (0, 1, 2): 1 1
O|X|X
-----
X|O|
-----
O| |O
-----
Human wins!
P Manya
1BM22CS187

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

Alpha-Beta pruning For 8 Queens.

Algorithm:

```
N-Queens:

def is_safe (board, row, col):
    i to 0, to row-1:
if b[i] == col or abs (b[i] -col) = abs(i-row):
    return False
Return True

def alphabeta (board, row, α, β):
    if row ==8:
Return True

For col 0 to 7:
  is_safe (board, row, col):
    board[row]= col
    if αβ (b, row+1, α, β):
    Return True
board [row] = -1

alpha = max(alpha, row)
  if alpha >= beta:
    Break.
Return False.

Solve_8_queens():
Initialise [-1,-1,-1,-1,-1,-1,-1,-1]
if alphabeta(board, 0, -∞, ∞):
Return board
Return None

Sol = solve_8_queens()
if sol is Not none:
    Print sol
  else No sol.
```

Code:

```
def is_safe(board, row, col):
    """
    Check if it's safe to place a queen at board[row][col].
    """
    # Check for queen in the same column
    for i in range(row):
        if board[i][col] == 1:
            return False
    # Check for queen in the left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Check for queen in the right diagonal
    for i, j in zip(range(row, -1, -1), range(col, len(board))):
        if board[i][j] == 1:
            return False
    return True
def solve_with_alpha_beta(board, row, alpha, beta):
    """
```

```
    Solve the 8-Queens problem using Alpha-Beta Pruning.
    """
    if row >= len(board):  # All queens placed successfully
        return True

    for col in range(len(board)):
        if is_safe(board, row, col):
            # Place the queen
            board[row][col] = 1
            # Recursive call to place the next queen
            if solve_with_alpha_beta(board, row + 1, alpha, beta):
                return True
            # Backtrack if placing the queen here leads to failure
            board[row][col] = 0
        # Update alpha and beta for pruning (though not strictly necessary for 8-Queens)
        alpha = max(alpha, col)
        if beta <= alpha:
            break  # Prune
    return False
def solve_8_queens():
```

```
    """
    Solves the 8-Queens problem and prints the solution.
    """
    n = 8
    board = [[0 for _ in range(n)] for _ in range(n)]
    # Start solving with Alpha-Beta Pruning
    if solve_with_alpha_beta(board, 0, -float('inf'), float('inf')):
        print("Solution:")
        for row in board:
            print(' '.join('Q' if cell == 1 else '.' for cell in row))
    else:
        print("No solution found.")
# Execute the solver
if __name__== "__main__":
    solve_8_queens()
print("P Manya")
print("1BM22CS187")
```

Output:

Output

```
Solution:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
P Manya
1BM22CS187

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