VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by
Srushti Sunkad (1BM23CS341)

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

Aug 2025 to Dec 2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Srushti Sunkad (1BM23CS341),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

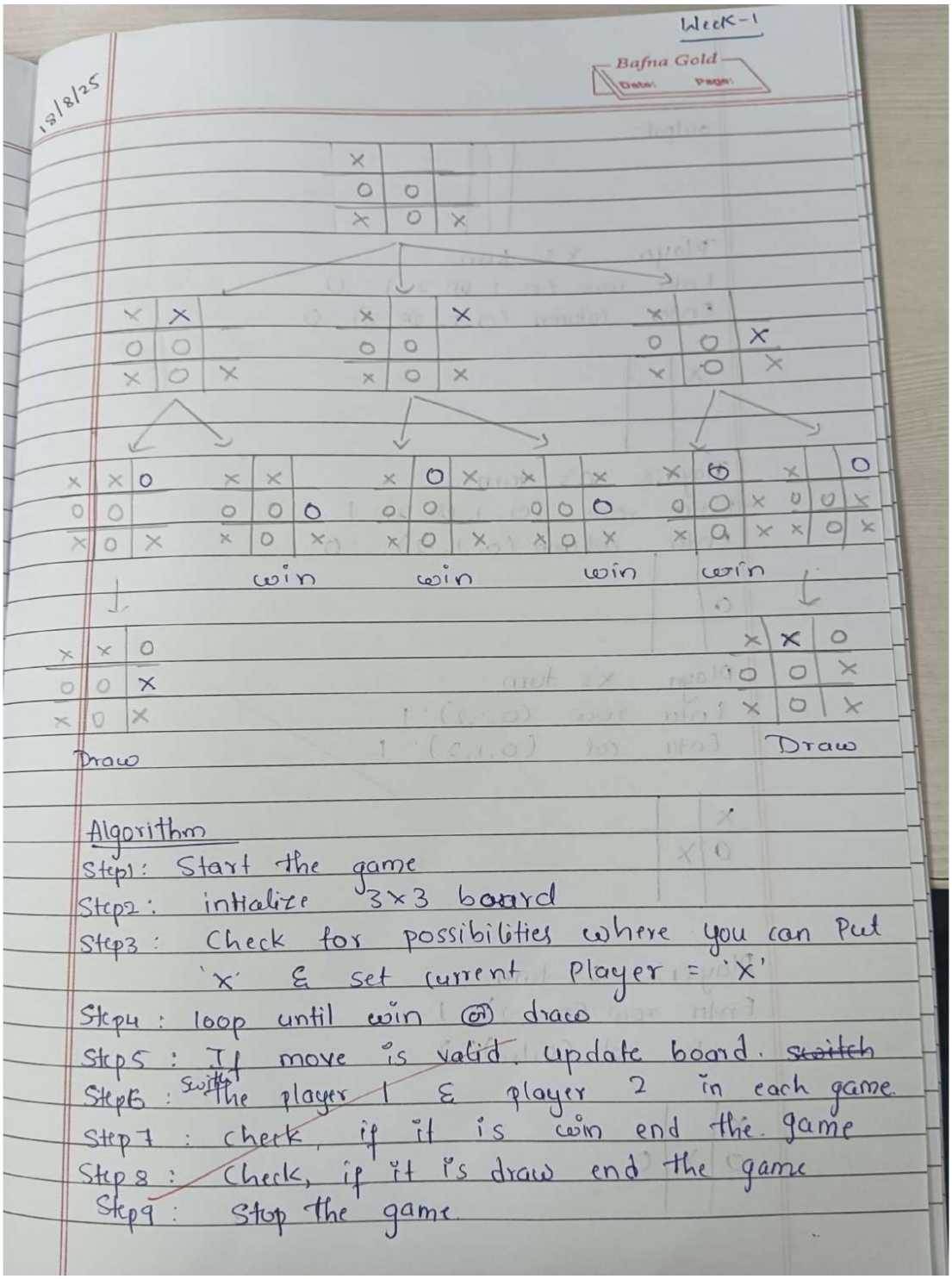
Index

Github Link:
https://github.com/SrushtiSunkad630/Artificial-Intelligence

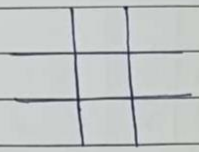
Program 1

Implement Tic –Tac –Toe Game Implement
vacuum cleaner agent

Algorithm:



output:



Player x's turn
Enter row (0, 1 or 2): 0
Enter column (0, 1, or 2): 0

X		

Player's 0's turn
Enter row (0, 1, or 2): 1
Enter column (0, 1, 2) : 0

X		
0		

Player x's turn
Enter row (0, 1, 2): 1
Enter col (0, 1, 2): 1

X		
0	X	

Player's 0's turn
Enter row (0, 1, 2): 1
Enter col (0, 1, 2): 2

X		
0	X	0

Player X's turn
Enter row (0, 1, 2): 2
Enter col (0, 1, 2): 2

X		
O	X	O
		X

Player X wins

25/08/25

Algorithm.

for two rooms.

Step1 : Start the vaccum cleaner implementation

Step2 : Consider two rooms A & B

Step3 : Consider the actions as left, right & sweep as L, R, & S

Step4 : Start the vaccum. check if the dust present by recording a direction of the dust.

Step5 : Send the vaccum in recorded direction

Step1 : Consider room A & B

Step2 : Start the vaccum cleaner

Step3 : Check if dirt present in room A, if Yes record the direction.
if no switch off the vaccum cleaner.

Step4 : ask user to select option
① Clean the room
② Stay in room
③ Go to another room.

Step5 : If user select ① sent a vaccum cleaner in the recorded direction.
if user select ② stay in room
if user select 3 sent to another room.

Step 6 : Repeat Steps from step 3.

Step7 : cost calculation

$O(b^d)$

$b = 4$

$d = 2$

$O(4^2) = O(16)$

Step 8 : Stop.

Bafna Gold
Date: Page:

output :
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B) : A
Cleaning A
Moving vacuum right
cleaning B.
Turning vacuum off
cost : 3
{'A' : 0, 'B': 0}

Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B) : A
Moving vacuum right.
Cleaning B.
Turning vacuum off
cost 2
{A : 0, 'B' : 0}

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

Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
~~Cleaning B.~~ Enter location (A or B): B
Cleaning B
Turning vacuum off
cost 1
{'A': 0, 'B': 0}

Code:
Tic –Tac –Toe Game

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows, columns and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or \
           all([board[j][i] == player for j in range(3)]):
            return True

    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 is_full(board):
    return all(cell in ['X', 'O'] for row in board for cell in row)


def get_move(player):
    while True:
        try:
            move = input(f"Player {player}, enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            if row in [1, 2, 3] and col in [1, 2, 3]:
                return row - 1, col - 1
            else:
                print("Invalid input. Enter numbers between 1 and 3.")
        except ValueError:
            print("Invalid input. Enter two numbers separated by space.")


def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        row, col = get_move(current_player)

        if board[row][col] != " ":
```

```
            print("That spot is taken. Try again.")
            continue

        board[row][col] = current_player

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

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = "O" if current_player == "X" else "X"


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

Output:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 1
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 1 2
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 3
X | O | X
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 2 2
X | O | X
---------
  | O |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 3 3
X | O | X
---------
  | O |
---------
  |   | X
---------
Player O, enter your move (row and column: 1 1): 3 1
X | O | X
---------
  | O |
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1): 2 3
X | O | X
---------
  | O | X
---------
O |   | X
---------
Player X wins!
```

Vacuum Cleaner

```
def vacuum_simulation():
    cost = 0

    # Get initial states and location
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    # Vacuum operation loop
    while True:
        if location == 'A':
            if state_A == 1:
                print("Cleaning A.")
                state_A = 0
                cost += 1
            elif state_B == 1:
                print("Moving vacuum right")
                location = 'B'
                cost += 1
            else:
                print("Turning vacuum off")
                break

        elif location == 'B':
            if state_B == 1:
                print("Cleaning B.")
                state_B = 0
                cost += 1
            elif state_A == 1:
```

```
            print("Moving vacuum left")
            location = 'A'
            cost += 1
        else:
            print("Turning vacuum off")
            break
    print(f"Cost: {cost}")
    print(f"{{'A': {state_A}, 'B': {state_B}}}")


vacuum_simulation()
```

OUTPUT

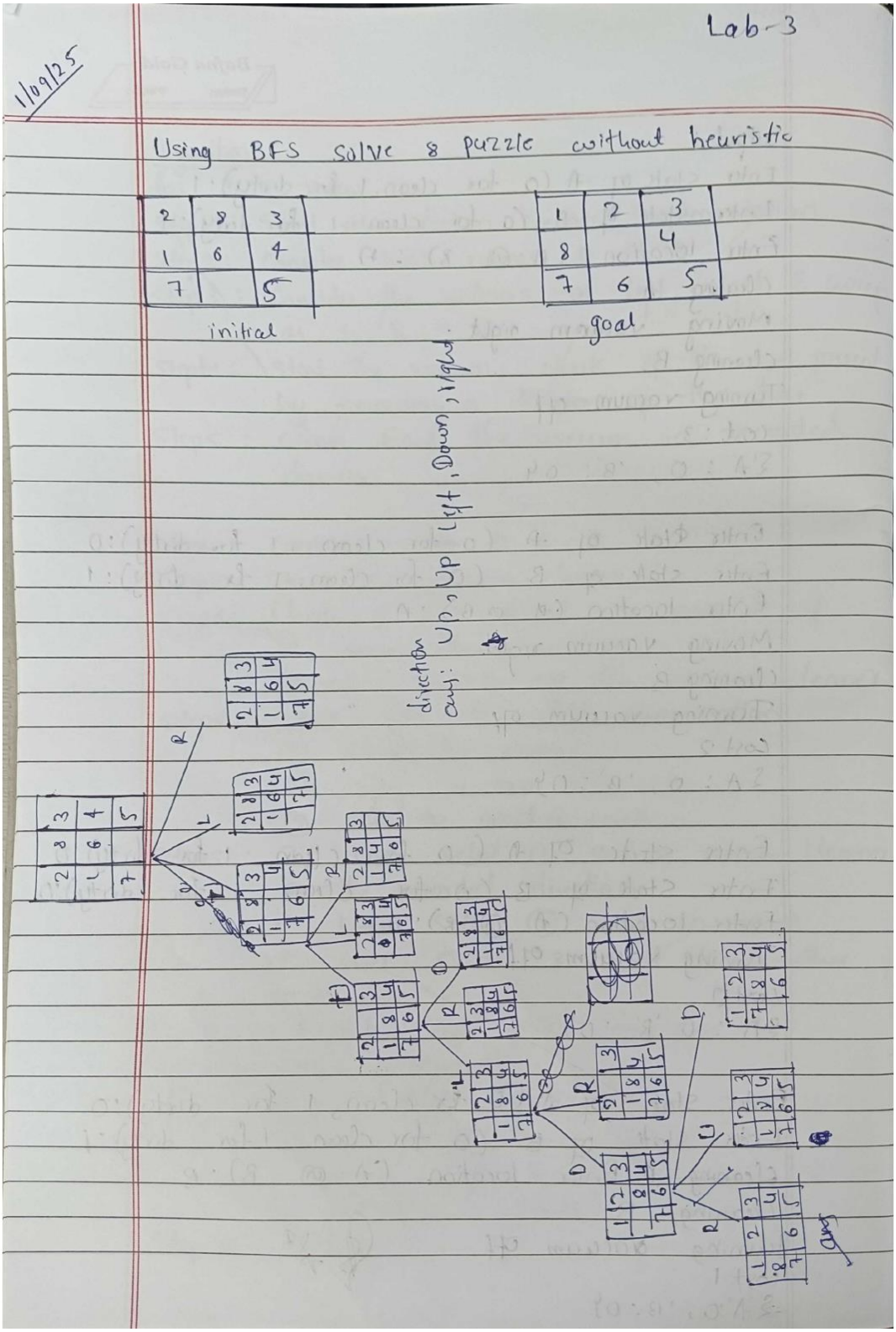
```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaning A.
Turning vacuum off
Cost: 1
{'A': 0, 'B': 0}
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:



Bafna Gold
Date: Page:

Algorithm

BFS

1. Start with initial puzzel
2. Check all states you can reach goal in 1 move
3. If goal is not found, then check for the states you can reach goal in 2 or 3 move. (up, down, right, left).
4. Stops when goal is reached.

DFS

1. Start
2. Pick one move & keep going deeper & until you can't go further.
3. If stuck, go back & try another path
4. Maintain a visited set to avoid cycles.
5. If top state is matches to goal state, return path
6. Otherwise generate all other next states
7. Again go back to step 5.

Same problem using DFS.

Bafna Gold
Date: Page:

O/P.

Using BFS.

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

Using DFS.

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

Iterative Deepening Search (IDS) or Iternative Deepening Depth First Search (IDDFS)

Algorithm

function Itr (problem) returns a solution

inputs: problem, a problem

for depth ← 0 to ∞ do

result ← Depth-Limited-Search (problem, depth)

if result ≠ cutoff then return result

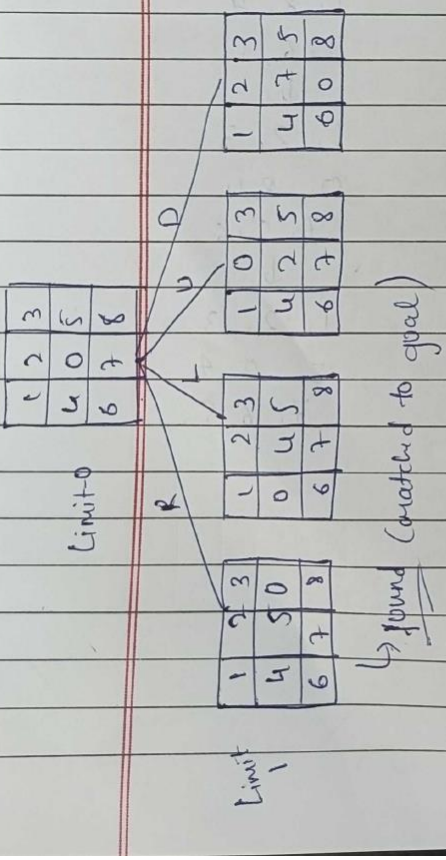
end

1	2	3
4	0	5
6	7	8

initial state

1	2	3
4	5	0
6	7	8

Goal



Bafna Gold
Date: Page:

O/P.

searching with depth limit = 0
searching with dept limit = 1
Solution found in 1 move

Step 0:

1 2 3
4 0 5
6 7 8

Step 1:

1 2 3
4 5 0
6 7 8

Code:
Usig DFS 8 puzzel without heuristic

```
# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_neighbors(state):
    # Find the empty tile (0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break

    neighbors = []
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Swap empty tile with adjacent tile
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def dfs_limited(start, depth_limit):
    stack = [(start, [start])]
    visited = set([start])

    while stack:
        current, path = stack.pop()

        if current == goal:
            return path

        if len(path) - 1 >= depth_limit:  # already reached depth limit
            continue

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor]))

    return None
```

```
# Example start state
start = ((2, 8, 3),
         (1, 6, 4),
         (7, 0, 5))

solution_path = dfs_limited(start, depth_limit=5)

if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:")
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found within 5 moves.")
```

OUTPUT

```
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

```
from copy import deepcopy
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 0],
    [6, 7, 8]
]

# Possible moves of the blank (0) tile: up, down, left, right
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def is_goal(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)

    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            # Swap blank with neighbor
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
```

```
            neighbors.append(new_state)
    return neighbors

def dfs(state, depth, limit, path, visited):
    if is_goal(state):
        return path + [state]
    if depth == limit:
        return None

    for neighbor in get_neighbors(state):
        # To avoid cycles, do not revisit states in current path
        if neighbor not in visited:
            result = dfs(neighbor, depth + 1, limit, path + [state], visited + [neighbor])
            if result is not None:
                return result
    return None

def iterative_deepening_search(initial_state, max_depth=50):
    for depth_limit in range(max_depth):
        print(f"Searching with depth limit = {depth_limit}")
        result = dfs(initial_state, 0, depth_limit, [], [initial_state])
        if result is not None:
            return result
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) for x in row))
    print()
```

```
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]
    ]

    solution = iterative_deepening_search(initial_state)

    if solution:
        print(f"Solution found in {len(solution)-1} moves!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            print_state(state)
    else:
        print("No solution found.")
```

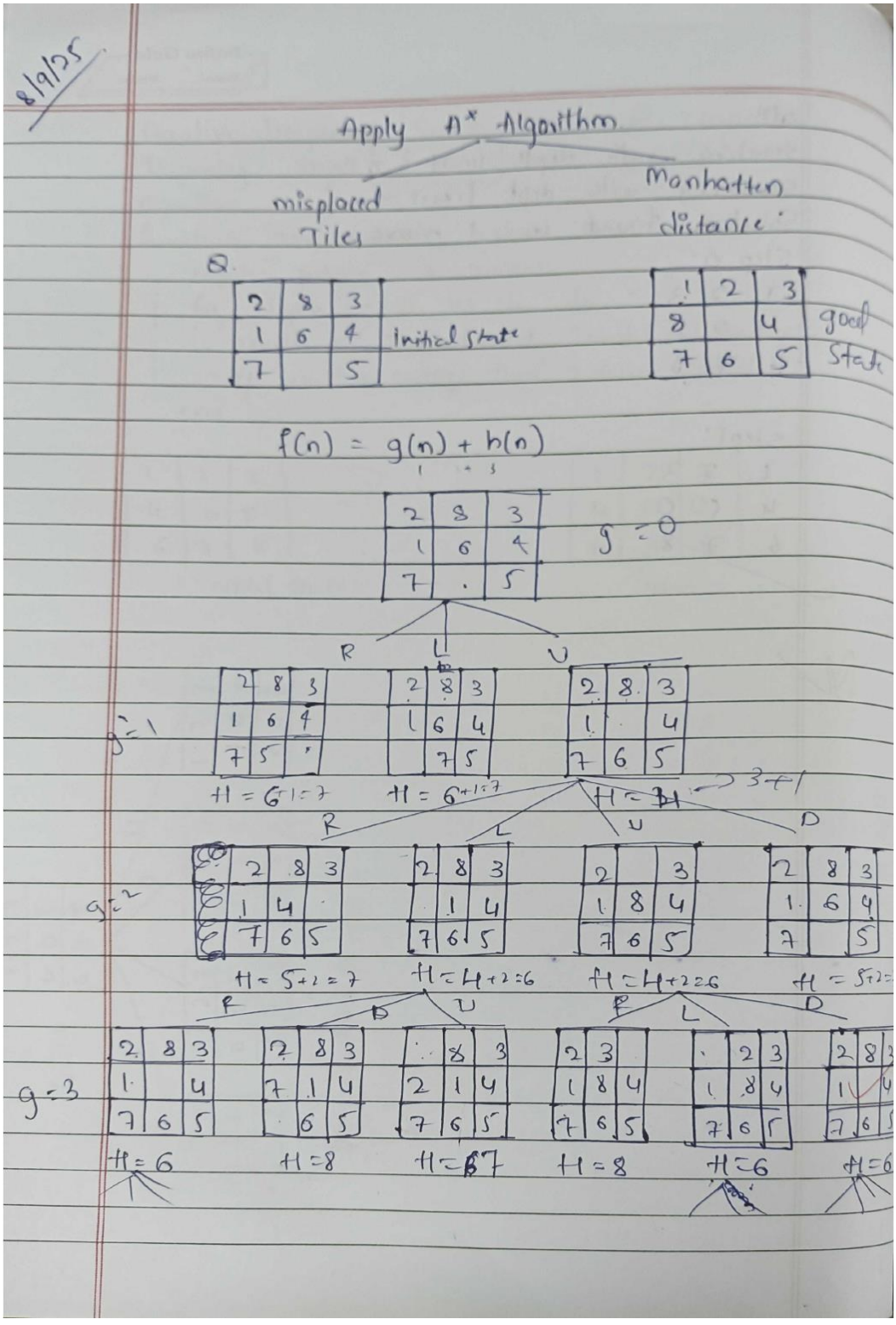
OUTPUT

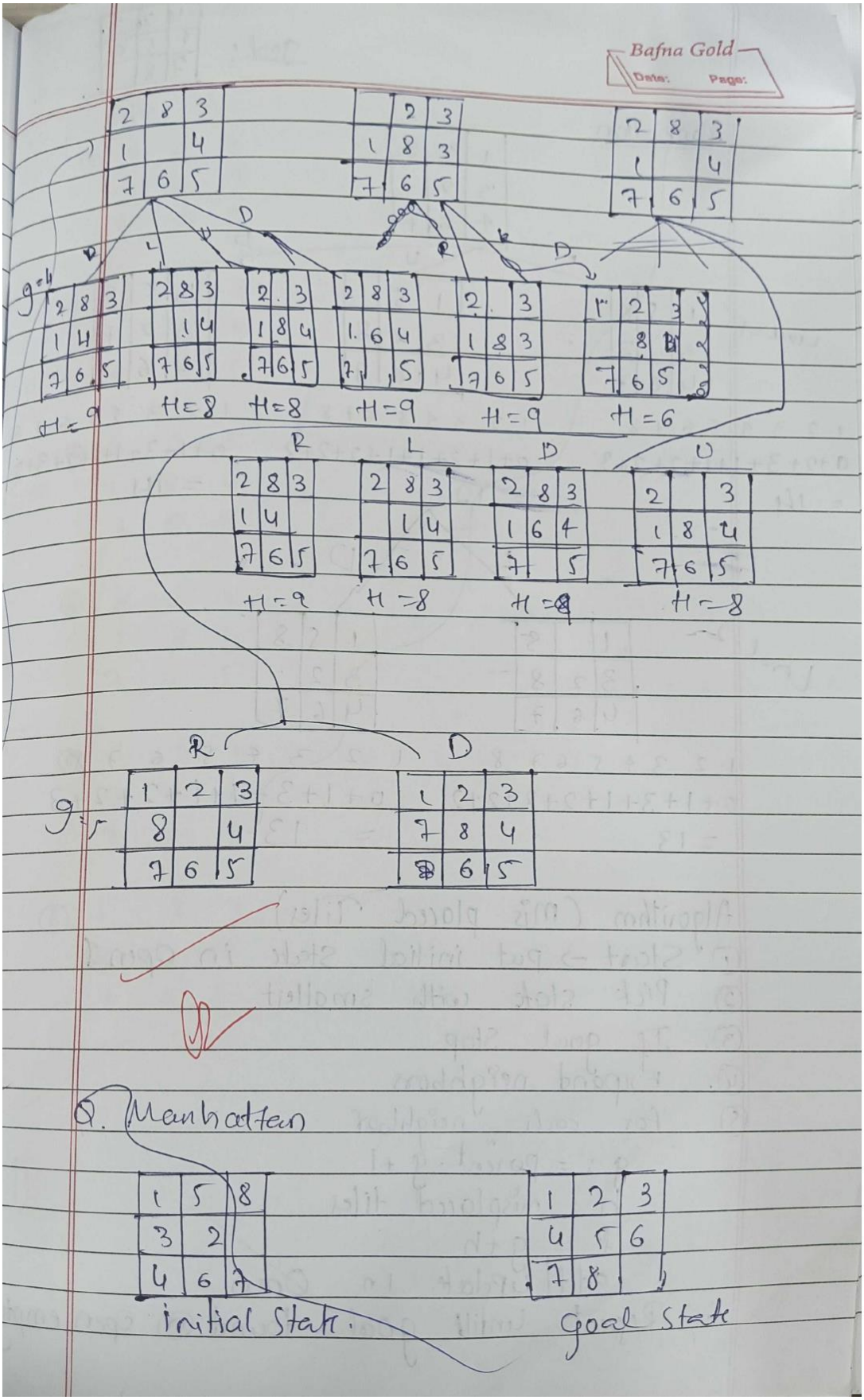
```
Searching with depth limit = 0
Searching with depth limit = 1
Solution found in 1 moves!
Step 0:
1 2 3
4 0 5
6 7 8

Step 1:
1 2 3
4 5 0
6 7 8
```

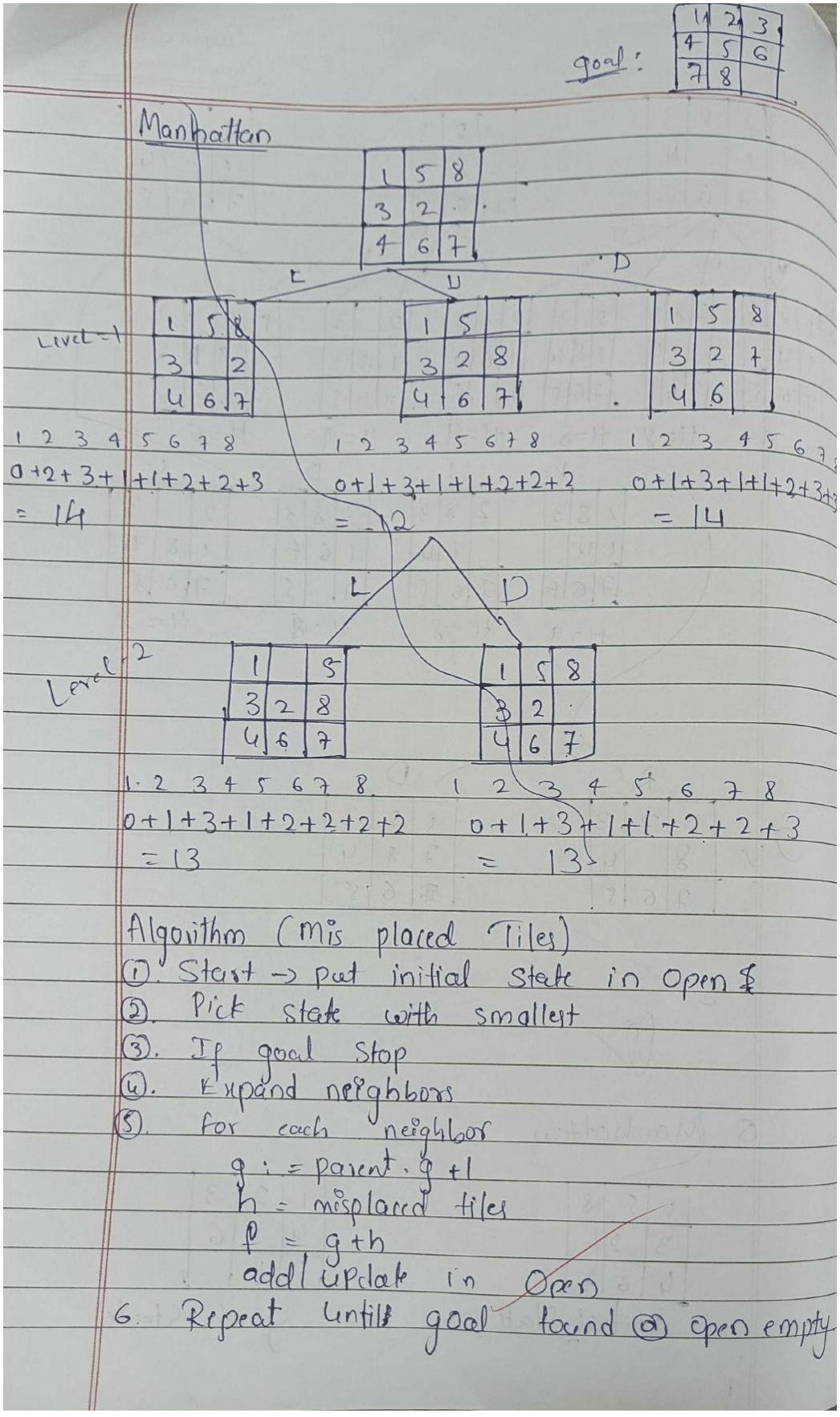
Program 3

Implement A* search algorithm

Algorithm:

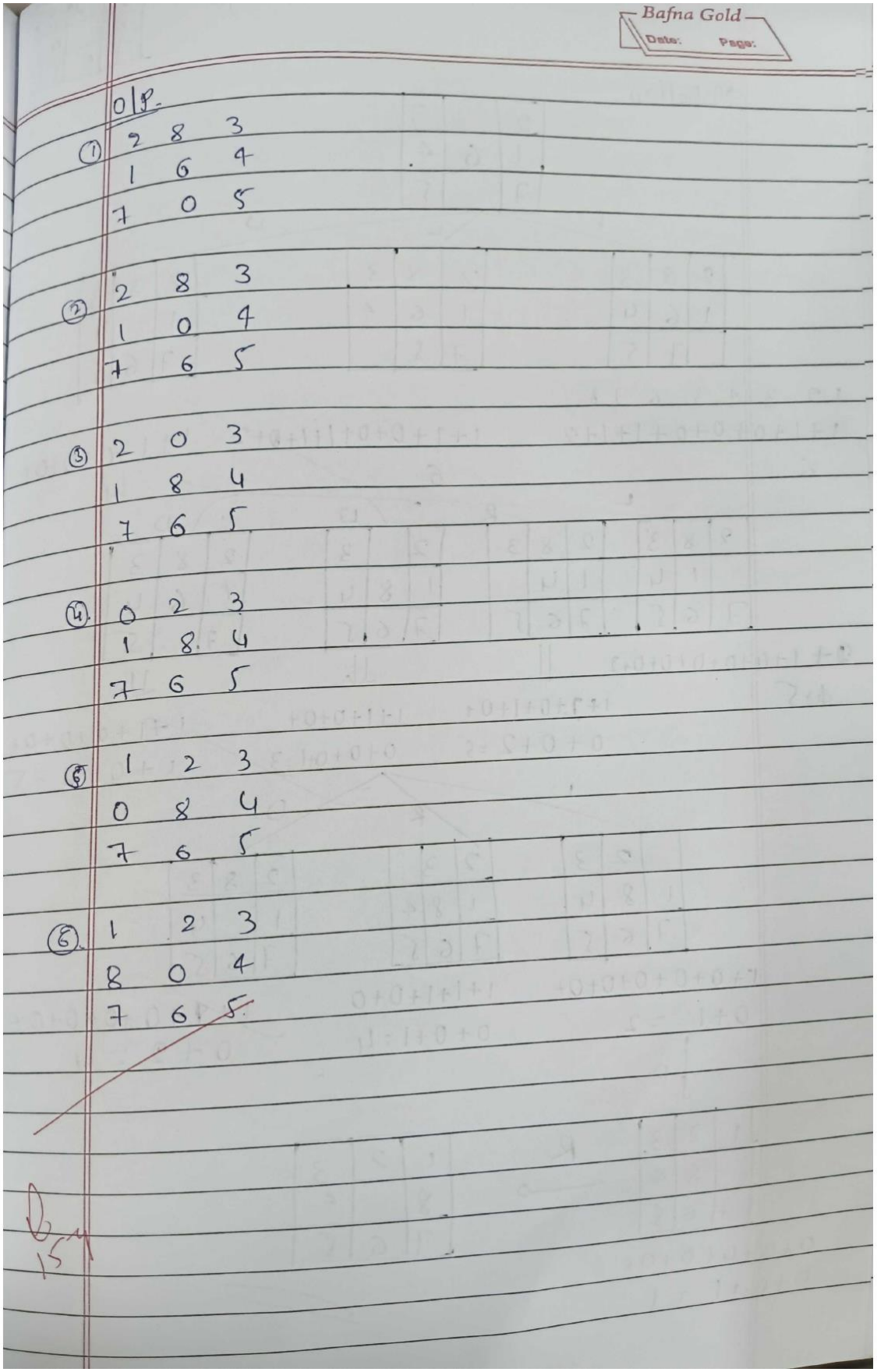



Bafna Gold
Date: Page:
g=4
H=9
H=8
H=8
H=9
H=9
H=6
R
L
D
U
H=9
H=8
H=8
H=8
R
D
g=5
1. Manhattan
initial State.
goal State



Algorithm (mis placed Tiles)

1. Start → put initial state in Open &
2. Pick state with smallest
3. If goal Stop
4. Expand neighbors
5. for each neighbor
   - g := parent.g + 1
   - h = misplaced tiles
   - f = g + h
   - add/update in Open
6. Repeat until goal found @ open empty


Bafna Gold
Date:
Page:
Q.8.
① 2 8 3
1 6 4
7 0 5
② 2 8 3
1 0 4
7 6 5
③ 2 0 3
1 8 4
7 6 5
④ 0 2 3
1 8 4
7 6 5
⑤ 1 2 3
0 8 4
7 6 5
⑥ 1 2 3
8 0 4
7 6 5
15

goal :-

1	2	3
8		4
7	6	5

Mahatfan

2	8	3
1	6	4
7		5

R L U

2	8	3
1	6	4
	7	5

2	8	3
1	6	4
7	5	

2	8	3
1		4
7	6	5

1 2 3 4 5 6 7 8

1+1+0+0+0+1+1+2 = 6

1+1+0+0+1+1+0+2 = 6

1+1+0+0+0+ = 4

L R U D

2	8	3
	1	4
7	6	5

2	8	3
1	4	
7	6	5

2		3
1	8	4
7	6	5

2	8	3
1	6	4
7		5

2+1+0+0+0+0+0+2 = 5

1+1+0+1+0+ = 0+0+2 = 5

1+1+0+0+ = 0+0+0+1 = 3

1+1+0+0+0+ = 1+0+2 = 5

L R D

	2	3
1	8	4
7	6	5

2	3	
1	8	4
7	6	5

2	8	3
1		4
7	6	5

1+0+0+0+0+0+ = 0+1 = 2

1+1+1+0+0 = 0+0+1 = 4

1+1+0+0+0+0+ = 0+2 = 4

D

1	2	3
	8	4
7	6	5

R →

1	2	3
8		4
7	6	5

0+0+0+0+0+0+ = 0+0+1 = 1

Algorithm.

1. Start with current pazzle state
2. Creak an empty list to keep track of file matches
3. For each file from 1 to 8
   - → if the file is in cur. pos. add 0 to the list
   - → otherwise, add 1 to the list.
4. Check the list:
   - → if the list contains all 0's the puzzel is solved = stop.
   - → otherwise, make a move to get closer to the goal.
5. Repeat steps 2 to 4 until, the puzzle is solved.

0+0+2

Output:

Solution path:

2	8	3
1	6	4
7	0	5

⇓

2	8	3
1	0	4
7	6	5

⇓

2	0	3
1	8	4
7	6	5

0	2	3
1	8	4
7	6	5

⇓

7	2	3
0	8	4
7	6	5

⇓

1	2	3
8	0	4
7	6	5

Code:
Misplace Tiles

```
import heapq


# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))


# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]


# Heuristic: Misplaced tiles
def misplaced_tiles(state):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal[i][j]:
                count += 1
    return count


# Find blank position (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Generate neighbors
def get_neighbors(state):
    neighbors = []
```

```
    x, y = find_blank(state)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state)
    return neighbors
# A* Search
def astar(start):
    pq = []
    heapq.heappush(pq, (misplaced_tiles(start), 0, start, []))
    visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + misplaced_tiles(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
```

```
# Example usage
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))

solution = astar(start_state)

# Print solution path
for step in solution:
    for row in step:
        print(row)
    print("------")
```

OUTPUT

```
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
------
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
------
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
------
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
------
```

Manhatten:

```
import heapq
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                # goal position of this tile
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
```

```
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors
def astar(start):
    pq = []
    heapq.heappush(pq, (manhattan_distance(start), 0, start, []))  # (f, g, state, path)
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))
solution = astar(start_state)

if solution is None:
    print("No solution found.")
else:
```

```
print("Solution path:")
for step in solution:
    for row in step:
        print(row)
    print("------")
```

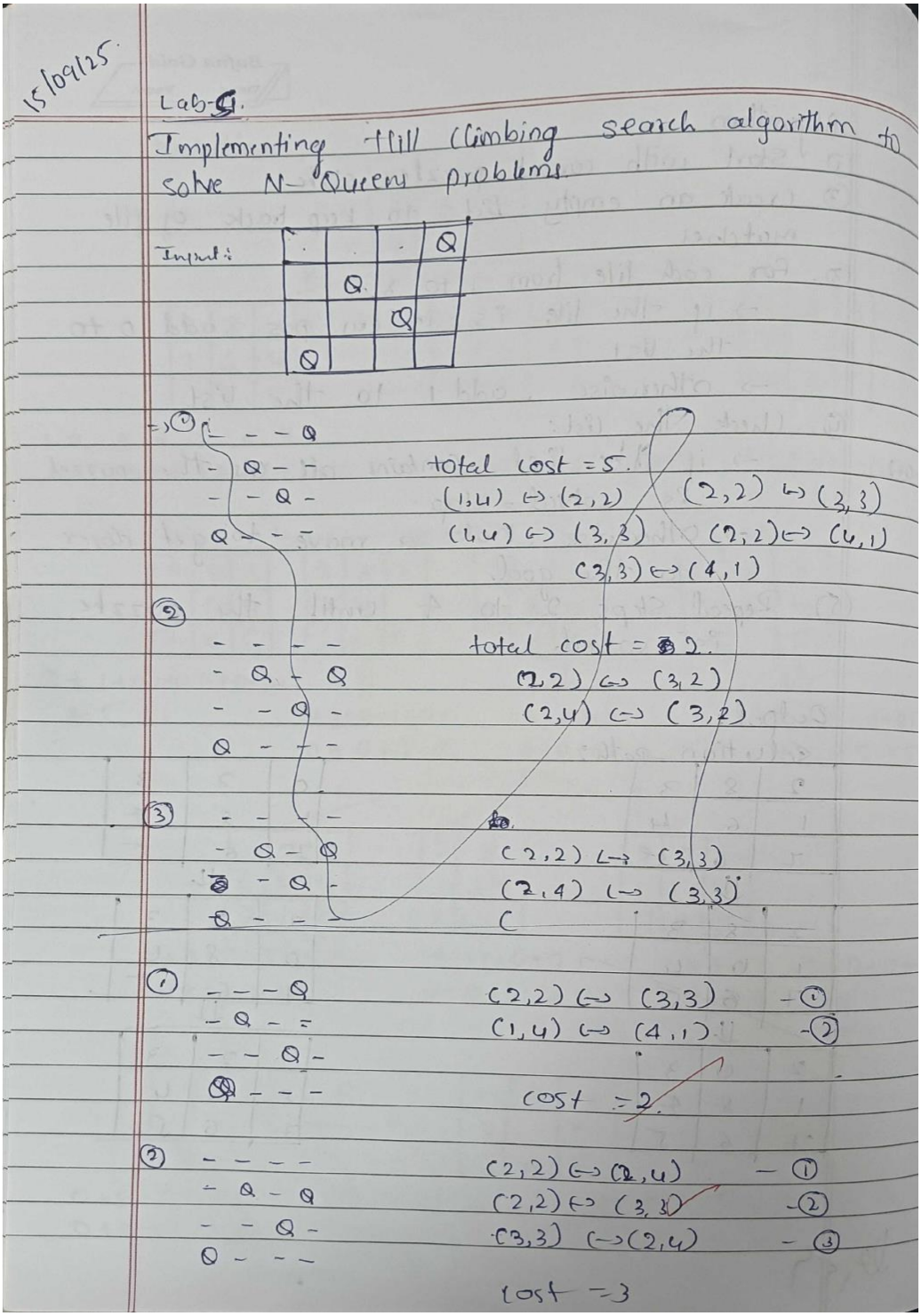
OUTPUT:

```
Solution path:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
------
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
------
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
------
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
------
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
------
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



(3) - - Q - (1,3) ↪ (2,2) - (1)
- Q - Q (2,2) ↪ (2,4) - (2)
- - - - (1,3) ↪ (2,4) - (3)
Q - - - cost = 3.

(4) - - - - (2,4) ↪ (3,3) - (1)
- - - Q (3,2) ↪ (3,1) - (2)
- Q Q - (3,2) ↪ (4,1) - (3)
Q - - - cost = 3.

(5) - Q - - cost = 0
- - - Q
Q - - -
- - Q -

Simulated Annealing

Algorithm
current ← initial state
T ← a large positive value
while T > 0 do
  next ← a random neighbour of current
  ΔE ← current.cost − next.cost
  if ΔE > 0 then
    current ← next
  else
    current ← next with probability $p = e^{\Delta E/T}$
  end if
  decrease T
end while
return current

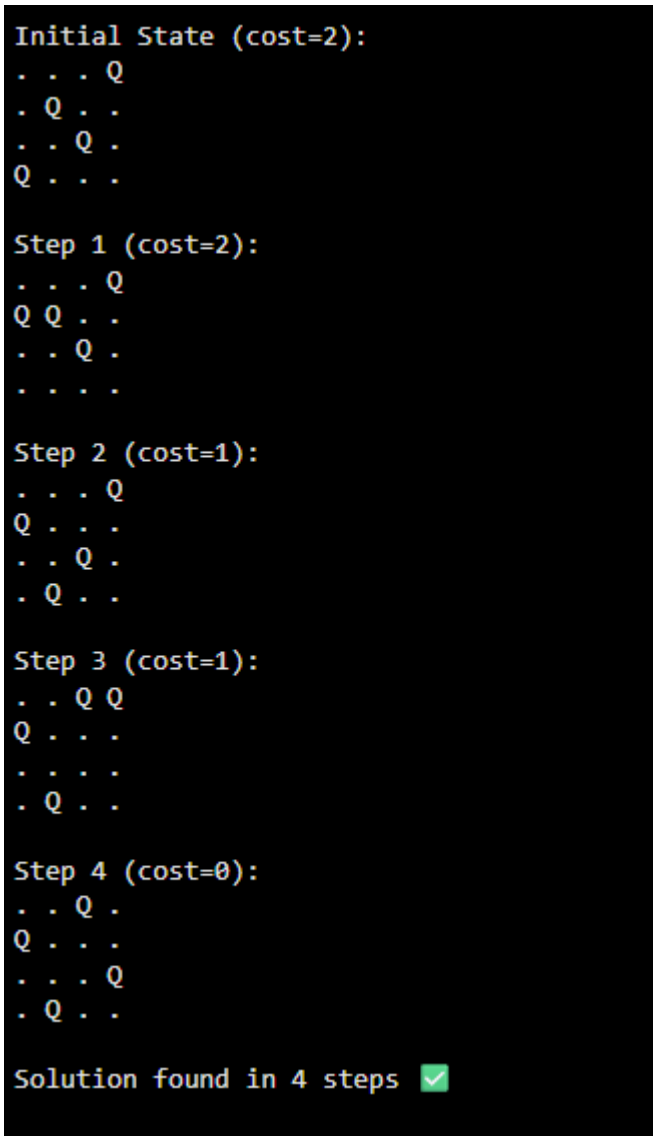
Code:

```
import random


def compute_cost(state):
    n = len(state)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:              # same row
                cost += 1
            elif abs(state[i] - state[j]) == abs(i - j):  # same diagonal
                cost += 1
    return cost



def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):        # pick a column
        for row in range(n):    # try moving queen in this column to another row
            if row != state[col]:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors




def print_board(state):
    n = len(state)
```

```
    for r in range(n):
        line = ""
        for c in range(n):
            line += "Q " if state[c] == r else ". "
        print(line)
    print("")


def hill_climb(initial_state, max_sideways=50):
    current = initial_state
    current_cost = compute_cost(current)
    steps = 0
    sideways_moves = 0

    print("Initial State (cost={}):".format(current_cost))
    print_board(current)

    while True:
        neighbors = get_neighbors(current)
        costs = [compute_cost(n) for n in neighbors]
        min_cost = min(costs)

        if min_cost > current_cost:
            # no better neighbor -> stop
            break

        # pick one of the best neighbors randomly
        best_neighbors = [n for n, c in zip(neighbors, costs) if c == min_cost]
        next_state = random.choice(best_neighbors)
        next_cost = compute_cost(next_state)
```

```
        # handle sideways moves
        if next_cost == current_cost:
            if sideways_moves >= max_sideways:
                break
            else:
                sideways_moves += 1
        else:
            sideways_moves = 0

        current = next_state
        current_cost = next_cost
        steps += 1

        print("Step {} (cost={}):".format(steps, current_cost))
        print_board(current)

        if current_cost == 0:
            print("Solution found in {} steps ☑".format(steps))
            return current

    print("Local minimum reached (cost={}) ❌".format(current_cost))
    return current

initial_state = [3, 1, 2, 0]

final = hill_climb(initial_state, max_sideways=10)
```

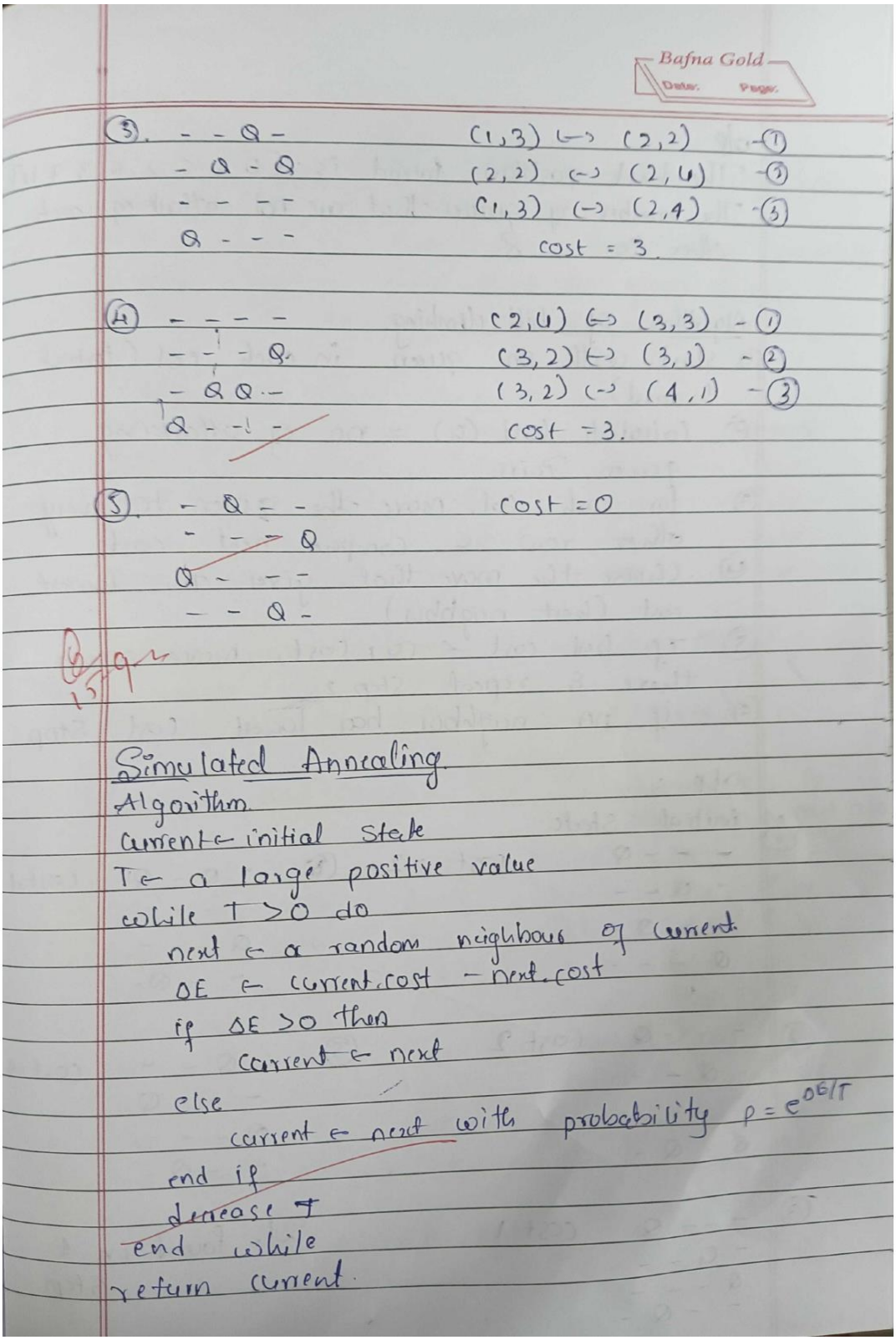
OUTPUT:


Initial State (cost=2):
. . . Q
. Q . .
. . Q .
Q . . .
Step 1 (cost=2):
. . . Q
Q Q . .
. . Q .
. . . .
Step 2 (cost=1):
. . . Q
Q . . .
. . Q .
. Q . .
Step 3 (cost=1):
. . Q Q
Q . . .
. . . .
. Q . .
Step 4 (cost=0):
. . Q .
Q . . .
. . . Q
. Q . .
Solution found in 4 steps

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Annealing

Algorithm

current ← initial State

T ← a large positive value

while T > 0 do

next ← a random neighbour of current

ΔE ← current.cost − next.cost

if ΔE > 0 then

current ← next

else

current ← next with probability $p = e^{\Delta E/T}$

end if

decrease T

end while

return current

o/p

The best position found is: [0 8 5 2 6 3 7 1]

The number of queues that are not attacking each other is : 8

Algorithm of hill climbing.

1. start with one queen in each col (intial board)
2. Calculate cost (G) = no. of attacking queen pairs.
3. for each col, move the queen to every other row & compute next cost.
4. Choose the move that gives the lowest cost (best neighbor)
5. If best cost < cur cost, move queen there & repeat step 2
6. if no neighbor has lowest cost, Stop

o/p :

1. initial state

- - - Q
- Q - -
- - Q -
Q - - -

cost = 2

2. - - - Q
- Q - -
- - - -
Q - Q -

cost 2

3. - - - Q
- Q - -
Q - - -
- - Q -

cost 1

4. - - Q -
- Q - -
Q - - -
- - Q -

cost 1

5. - Q - -
- - - Q
Q - - -
- - Q -

cost=

Sol. found in 4 steps.

Code:

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

def queens_max(x):
    cols = np.round(x).astype(int)
    n = len(cols)

    if len(set(cols)) < n:
        return 1e6
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(cols[i] - cols[j]):
                attacks += 1
    return attacks
n = 8
bounds = [(0, n - 1)] * n
result = dual_annealing(queens_max, bounds)

best_cols = np.round(result.x).astype(int).tolist()
not_attacking = n

print(f"The best position found is: {best_cols}")
print(f"The number of queens that are not attacking each other is: {not_attacking}")
```

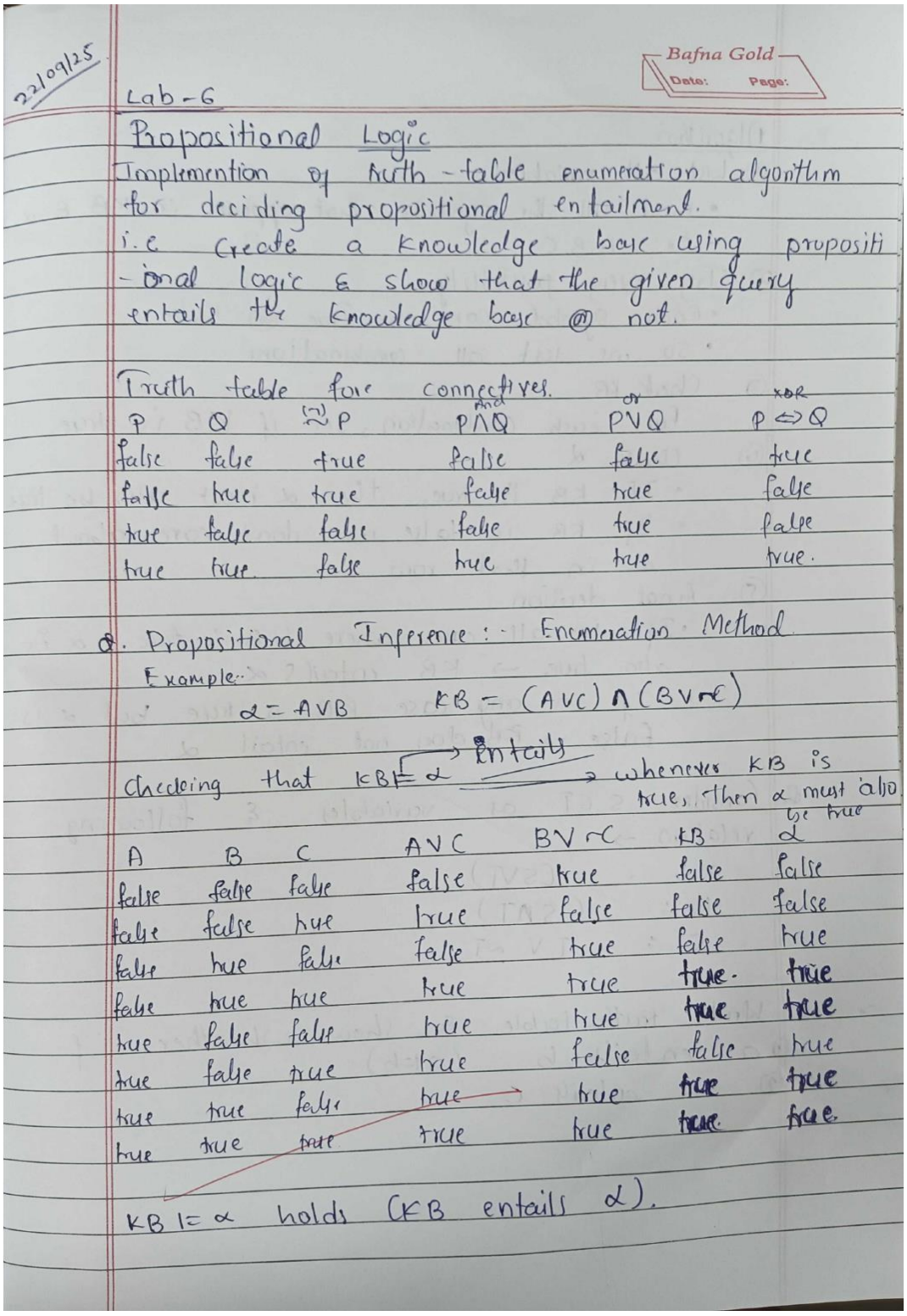
OUTPUT:

```
The best position found is: [7, 4, 6, 1, 3, 5, 0, 2]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:



22/09/25

Bafna Gold
Date: Page:

Lab - 6

Propositional Logic

Implemention of Truth-table enumeration algorithm for deciding propositional entailment.

i.e Create a knowledge base using propositi-onal logic & show that the given query entails the knowledge base or not.

Truth table for connectives.

P	Q	¬P	P ∧ Q (and)	P ∨ Q (or)	P ⇔ Q (XOR)
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true.

Q. Propositional Inference : Enumeration Method

Example:

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

Checking that $KB \models \alpha$ → entails → whenever KB is true, then α must also be true

A	B	C	A ∨ C	B ∨ ¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true.	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true.	true.

$KB \models \alpha$ holds (KB entails α).

Algorithm

① List all variables
- Find all the symbols that appear in KB & α
- Ex:- A, B, C

② Try every possibility
- Each symbol can be True or False
- So we test all combinations

③. Check KB
For each combination, see if KB is true.

④. Check α
- If KB is true, then α must also be true
- If KB is false, we don't care about α in that row.

⑤. Final decision
- If in all cases where KB is true, α is also true → KB entails α.
- If in any case KB is true but α is false → KB does not entail α.

Q. Consider S ET as variables & following relation →

a : (S ∨ T)

b : (S ∧ T)

c : T ∨ ¬T

Write truth table & show wheather if
① a entails b (a ⊨ b)
② a entails c

Bafna Gold
Date: Page:

S	T	~T	a: S∨T	b: S∧T	c: T∨~T
0	0	1	0	0	1
0	1	0	1	0	1
1	0	1	1	0	1
1	1	0	1	1	1

① $a \not\equiv b$ ⇒ not holds. ×

② $a \not\equiv c$ ⇒ holds ✓

SKD
22/9/25

Code:

```
import itertools
from sympy import symbols, sympify

A, B, C = symbols('A B C')

alpha_input = input("Enter alpha (example: A | B): ")
kb_input = input("Enter KB (example: (A | C) & (B | ~C)): ")

alpha = sympify(alpha_input, evaluate=False)
kb = sympify(kb_input, evaluate=False)

GREEN = "\033[92m"
RESET = "\033[0m"

print(f"\nTruth Table for α = {alpha_input}, KB = {kb_input}\n")
print(f"{'A':<6}{'B':<6}{'C':<6}{'α':<10}{'KB':<10}")

entailed = True

for values in itertools.product([False, True], repeat=3):
    subs = {A: values[0], B: values[1], C: values[2]}
    alpha_val = alpha.subs(subs)
    kb_val = kb.subs(subs)

    alpha_str = f"{GREEN}{alpha_val}{RESET}" if kb_val else str(alpha_val)
    kb_str = f"{GREEN}{kb_val}{RESET}" if kb_val else str(kb_val)

    print(f"{str(values[0]):<6}{str(values[1]):<6}{str(values[2]):<6}"
          f"{alpha_str:<10}{kb_str:<10}")
```

```
    if kb_val and not alpha_val:
        entailed = False

if entailed:
    print(f"\n KB |= α holds (KB entails α)\n")
else:
    print(f"\n KB does NOT entail α\n")
```

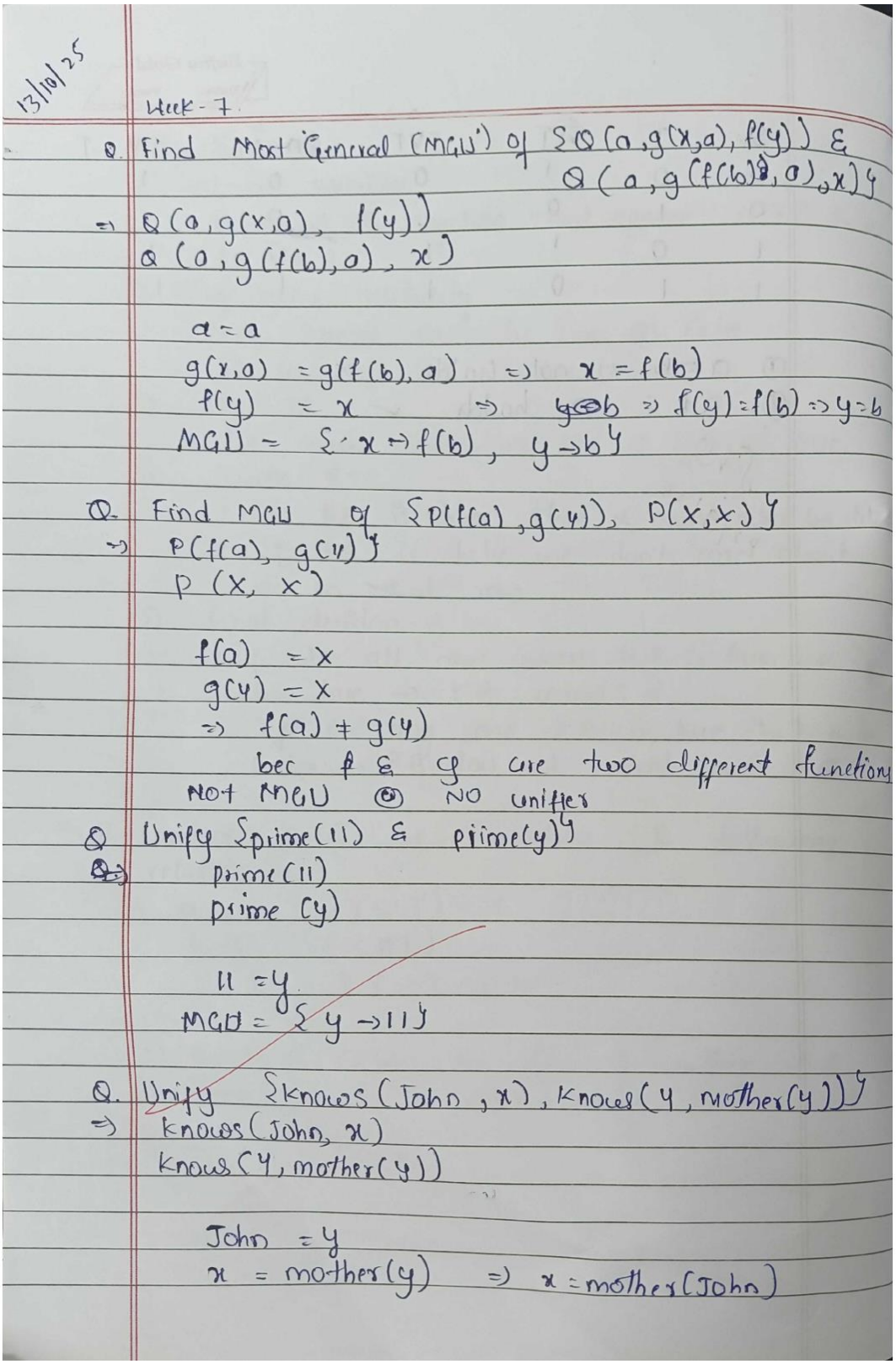
OUTPUT:

```
Enter alpha (example: A | B): A|B
Enter KB (example: (A | C) & (B | ~C)): (A | C) & (B | ~C)

Truth Table for α = A|B, KB = (A | C) & (B | ~C)

A     B     C     α         KB
False False False False     False
False False True  False     False
False True  False True      False
False True  True  TrueTrue
True  False False TrueTrue
True  False True  True      False
True  True  False TrueTrue
True  True  True  TrueTrue

 KB |= α holds (KB entails α)
```

Program 7

Implement unification in first order logic

Algorithm:



MGU = {y → John, x → mother(John)}

Q. Unify {knows(John, x), knows(y, Bill)}

⇒ knows(John, x)
knows(y, Bill)

John = y
x = Bill

MGU = {y → John, x → Bill}

Q. Find Most General Unifier (MGU) of {P(b, x, f(g(z))) & P(z, f(y), f(y))}

⇒ P(b, x, f(g(z)))
P(z, f(y), f(y))

b = z
x = f(y)
f(g(z)) = f(y) → g(z) = y

MGU = {z → b, x → f(g(z)), y → g(z)}

Unification Algo.
Algo: Unif(ψ1, ψ2)

Step 1: If ψ1 or ψ2 is a variable or constant, then:
a). If ψ1 or ψ2 are identical, then return NIL.
b). Else if ψ1 is a variable,
a. then if ψ1 occurs in ψ2, then return FAILURE
b. Else return {(ψ2/ψ1)}
c). Else if ψ2 is a variable,

a. If ψ_2 occurs in ψ_1 then return FAILURE,
b. Else return $\{(\psi_1/\psi_2)\}$.

d). Else return ~~$\{(\psi_1/\psi_2)\}$~~ FAILURE

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

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

Step 4 : Set substitution set(SUBST) to Nil.

step 5 : For i=1 to the no. of ele in ψ_1.

a). Call Unify function with the ith ele. of ψ_1 & ith ele of ψ_2 & put the res into S.

b). If S = failure then returns failure

c). If S ≠ NIL then do,

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

b. SUBSET = APPEND(S, SUBSET).

Step 6 : Return SUBSET).

~~Output : MGU = {'b' : 'z', 'x' : ('f', 'y').~~

Output MGU = {$z \to b$, $x \to f(g(z))$, $y \to g(z)$}

Code:

```
def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr[1:])  # Skip function symbol
    return False
def substitute(expr, subst):
    if isinstance(expr, str):
        # Follow substitution chain until fully resolved
        while expr in subst:
            expr = subst[expr]
        return expr
    # If it's a function term: (f, arg1, arg2, ...)
    return (expr[0],) + tuple(substitute(sub, subst) for sub in expr[1:])
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}
    Y1 = substitute(Y1, subst)
    Y2 = substitute(Y2, subst)

    # Case 1: identical
    if Y1 == Y2:
        return subst

    # Case 2: Y1 is variable
    if isinstance(Y1, str):
        if occurs_check(Y1, Y2):
            return "FAILURE"
        subst[Y1] = Y2
        return subst
```

```
    # Case 3: Y2 is variable
    if isinstance(Y2, str):
        if occurs_check(Y2, Y1):
            return "FAILURE"
        subst[Y2] = Y1
        return subst

    # Case 4: function mismatch
    if Y1[0] != Y2[0] or len(Y1) != len(Y2):
        return "FAILURE"

    # Case 5: unify arguments
    for a, b in zip(Y1[1:], Y2[1:]):
        subst = unify(a, b, subst)
        if subst == "FAILURE":
            return "FAILURE"

    return subst
expr1 = ("p", "X", ("f", "Y"))
expr2 = ("p", "a", ("f", "b"))

output = unify(expr1, expr2)
print(output)
```

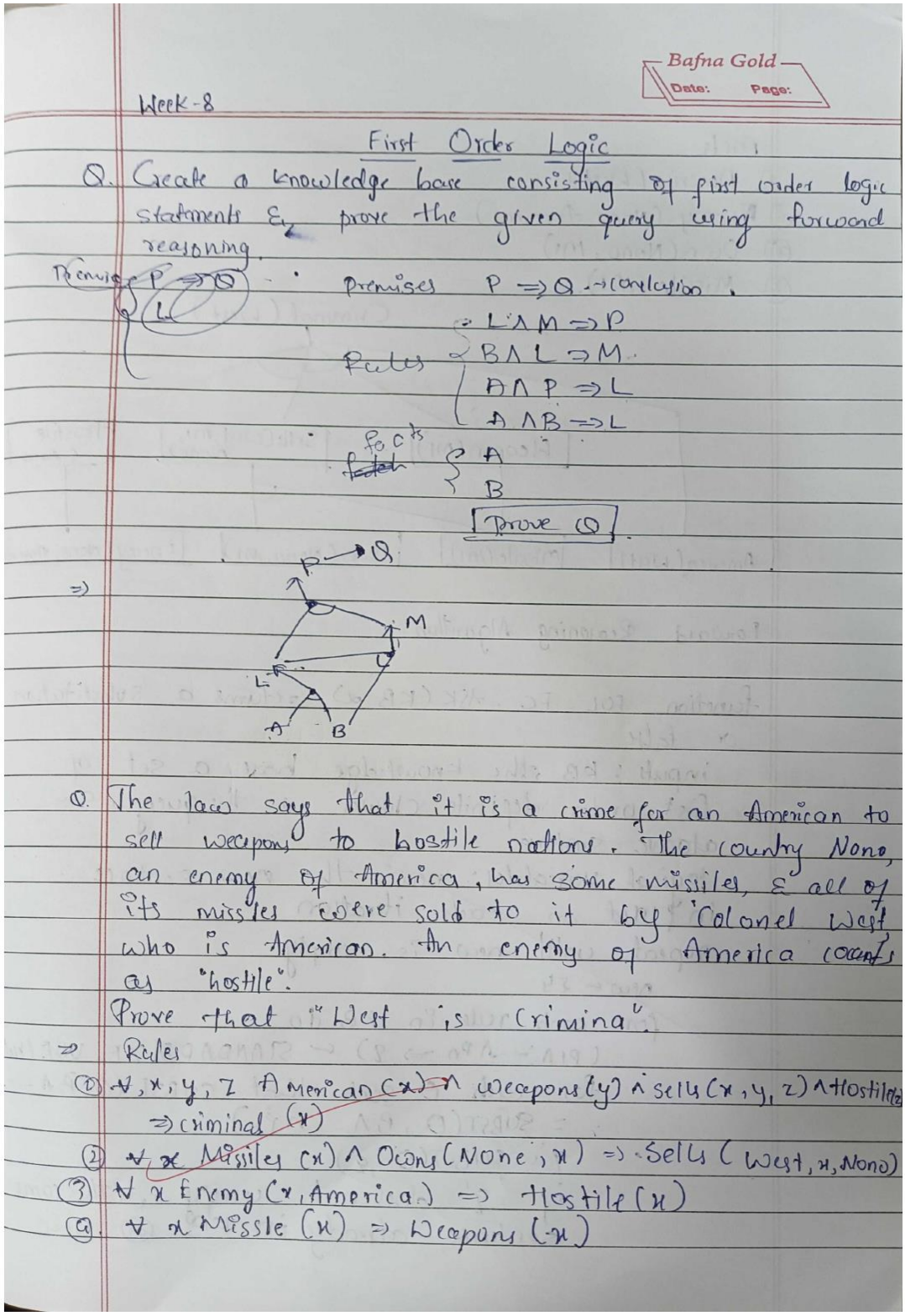
OUTPUT:

```
{'X': 'a', 'Y': 'b'}
```

Program 8

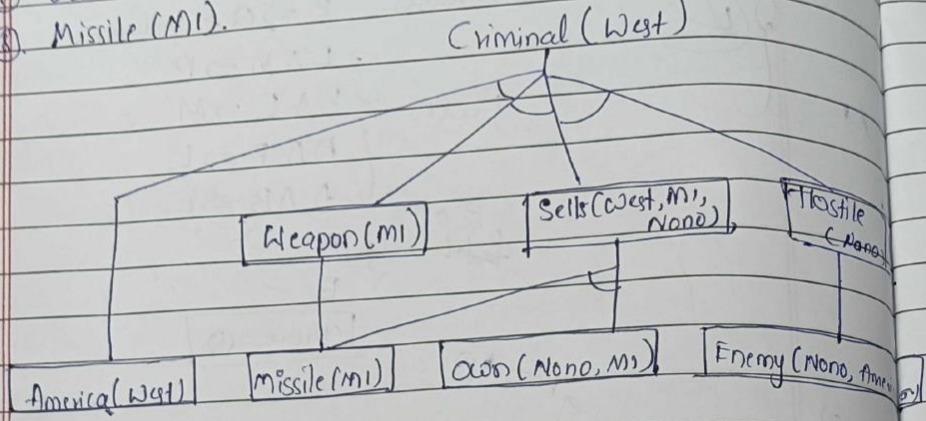
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



Facts

5. America(West)
6. Enemy(Nono, America)
7. Owns(Nono, M1)
8. Missile(M1).



Forward Reasoning Algorithm

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

inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration
repeat until new is empty
new ← {}
for each rule in KB do
(P1 ∧ ... ∧ Pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, P1 ∧ ... ∧ Pn) = SUBST(θ, P'1 ∧ ... ∧ P'n)
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new

Bafna Gold
Date: Page:

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

Output

Adding fact : American(West)
Adding fact : Enemy(Nono, America)
Adding fact : Missile(M1)
Adding fact : Owns(Nono, M1)
Inferred new fact : Weapon(M1) from ['Missile(M1)'] ⇒ Weapon(M1)
Inferred new fact : Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)'] ⇒ Sells(West, M1, Nono)
Inferred new fact : Hostile(Nono) from ['Enemy(Nono, America)'] ⇒ Hostile(Nono)
Inferred new fact : Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'] ⇒ Criminal(West).
Goal Reached : West is Criminal
True.

13/10/20

P9

Code:

```
from collections import deque
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

    def add_fact(self, fact):
        if fact not in self.facts:
            print(f"Adding fact: {fact}")
            self.facts.add(fact)
            return True
        return False

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def forward_chain(self):
        agenda = deque(self.facts)

        while agenda:
            fact = agenda.popleft()
            if fact in self.inferred:
                continue
            self.inferred.add(fact)

            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
                    if conclusion not in self.facts:
```

```
                print(f"Inferred new fact: {conclusion} from {premises} => {conclusion}")
                self.facts.add(conclusion)
                agenda.append(conclusion)

                if conclusion == 'Criminal(West)':
                    print("\n☑ Goal Reached: West is Criminal")
                    return True
        return False
kb = KnowledgeBase()

kb.add_fact('American(West)')
kb.add_fact('Enemy(Nono, America)')
kb.add_fact('Missile(M1)')
kb.add_fact('Owns(Nono, M1)')

kb.add_rule(premises=['Missile(M1)'], conclusion='Weapon(M1)')

kb.add_rule(premises=['Missile(M1)', 'Owns(Nono, M1)'], conclusion='Sells(West, M1, Nono)')

kb.add_rule(premises=['Enemy(Nono, America)'], conclusion='Hostile(Nono)')
kb.add_rule(premises=['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'],
conclusion='Criminal(West)')
kb.forward_chain()
```

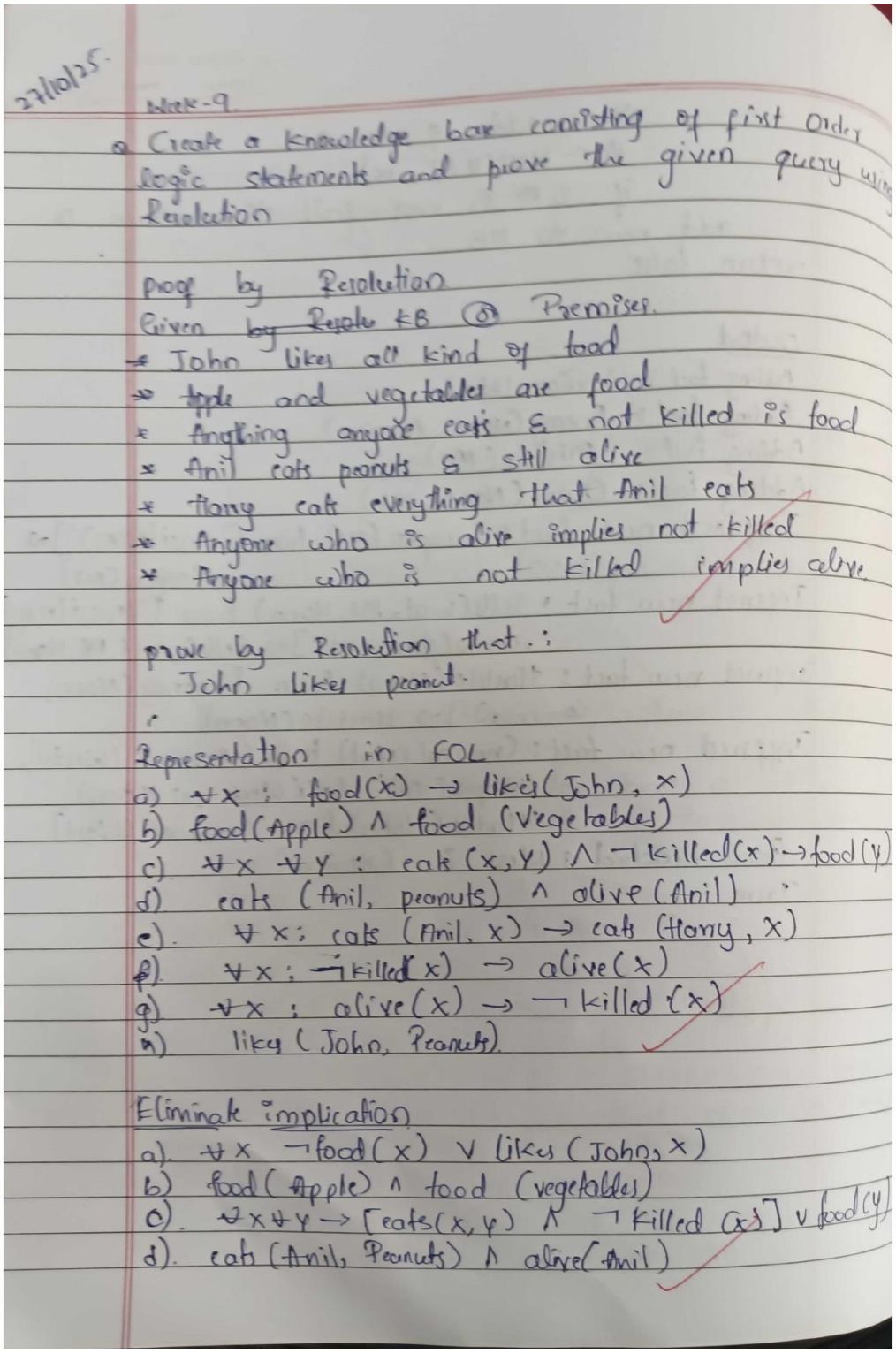
OUTPUT:

```
Adding fact: American(West)
Adding fact: Enemy(Nono, America)
Adding fact: Missile(M1)
Adding fact: Owns(Nono, M1)
Inferred new fact: Weapon(M1) from ['Missile(M1)'] => Weapon(M1)
Inferred new fact: Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact: Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)'] => Sells(West, M1, Nono)
Inferred new fact: Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'] => Criminal(West)

✅ Goal Reached: West is Criminal
True
```

Program 9

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

Algorithm:



27/10/25.

Week-9

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

Proof by Resolution

Given KB (a) Premises.

- John likes all kind of food
- Apple and vegetables are food
- Anything anyone eats & not killed is food
- Anil eats peanuts & still alive
- Harry eats everything that Anil eats
- Anyone who is alive implies not killed
- Anyone who is not killed implies alive.

prove by Resolution that.:
John likes peanut.

Representation in FOL

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(Vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$
e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f) $\forall x : \neg killed(x) \rightarrow alive(x)$
g) $\forall x : alive(x) \rightarrow \neg killed(x)$
h) $likes(John, Peanuts)$

Eliminate implication

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

Bafna Gold
Date: Page:

e). $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f). $\forall x \neg [\neg killed(x)] \vee alive(x)$
g). $\forall x \neg alive(x) \vee \neg killed(x)$
h). Likes (John, Peanuts)

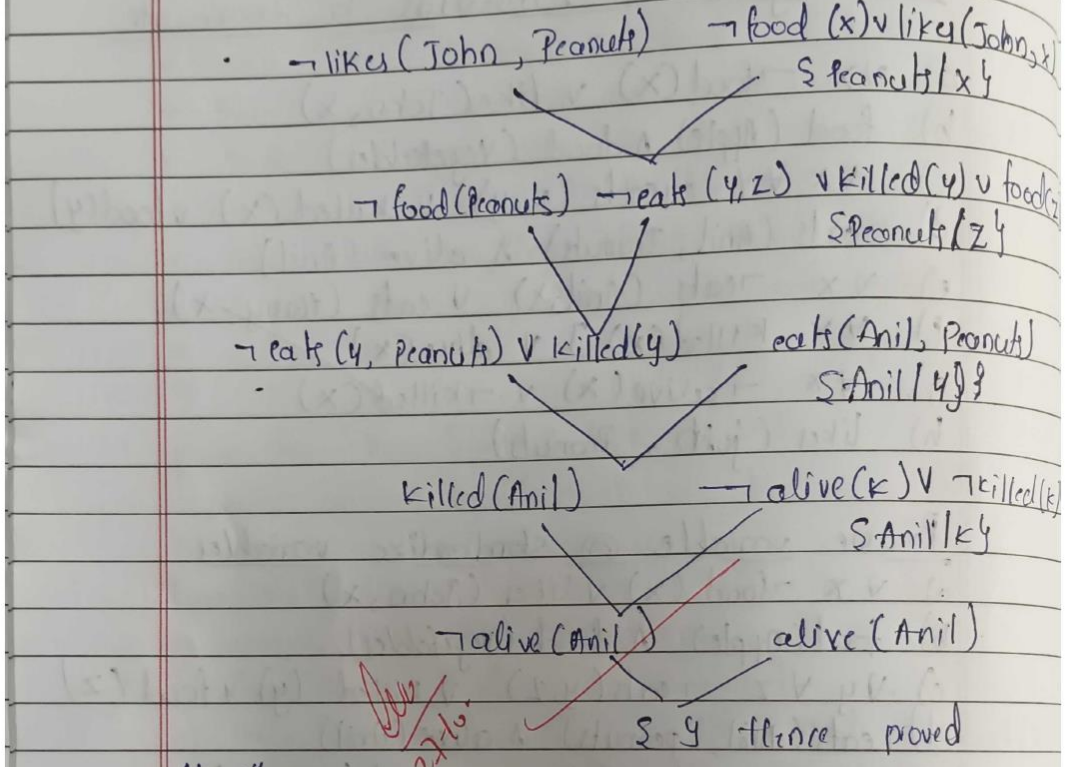
More negation ($\neg$) inwards & rewrite

a). $\forall x \neg food(x) \vee Likes(John, x)$
b). food (Apple) $\wedge$ food (vegetables)
c). $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
d). eats (Anil, Peanuts) $\wedge$ alive (Anil)
e). $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f) $\forall x killed(x)] \vee alive(x)$
g). $\forall x \neg alive(x) \vee \neg killed(x)$
h). likes (john, Peanuts)

Rename variable @ standardize variables
a). $\forall x \neg food(x) \vee likes(John, x)$
b). food (Apple) $\wedge$ food (vegetables)
c) $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$
d). eats (Anil, peanuts) $\wedge$ alive (Anil)
e) $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$.
f). $\forall g killed(g)] \vee alive(g)$
g) $\forall k \neg alive(k) \vee \neg killed(k)$
h). likes (John, Peanuts)

Drop universe
a). $\neg food(x) \vee likes(John, x)$
b). food (Apple)
c). food (Vegitables)
d). $\neg eats(y, z) \vee killed(y) \vee food(z)$
e). eats (Anil, Peanuts

f). alive (Ani)
g). ¬eats (Anil, w) ∨ eats (Harry, w)
h). killed (g) ∨ alive (g)
i). ¬ alive (k) ∨ ¬ killed (k)
j). likes (John, Peanuts).

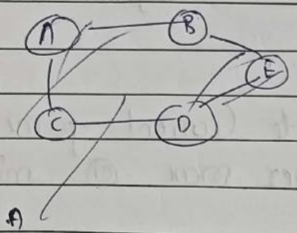


Algorithm
Algorithm ①. Input: 1). Knowledge Base (KB)
2). Query (Q)
② Convert KB & ¬Q to clausal form:
- Eliminate implications
- Move negations inward
- Standardize variables
- Skolemize (remove ∃ quantifiers)
- Drop universal quantifiers
- Convert to CNF.

Week 10

Bafna Gold
Date: Page:

Ex:



3) Apply resolution:
- Repeatedly resolve pairs of clauses that contain complementary literals.
- Add new clauses to the KB
- Stop if:
  - Empty clause (⊥) is derived → Q is true ✓
  - No new clause can be added → Q is false.

4). Output True/False for query Q.

Code:

```
from itertools import combinations

def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses

def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.append(new_clause)
    return resolvents

def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
```

```
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)], clause_id[frozenset(cj)]
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} = Resolve({cid_i}, {cid_j}) → {res or '{}'}")
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n☑ Query is proved by resolution (empty clause found).")
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)


def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in [(r[0], r[1], r[2:][0], r[2:][1]) for r in derived]}

    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
```

```
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)
```

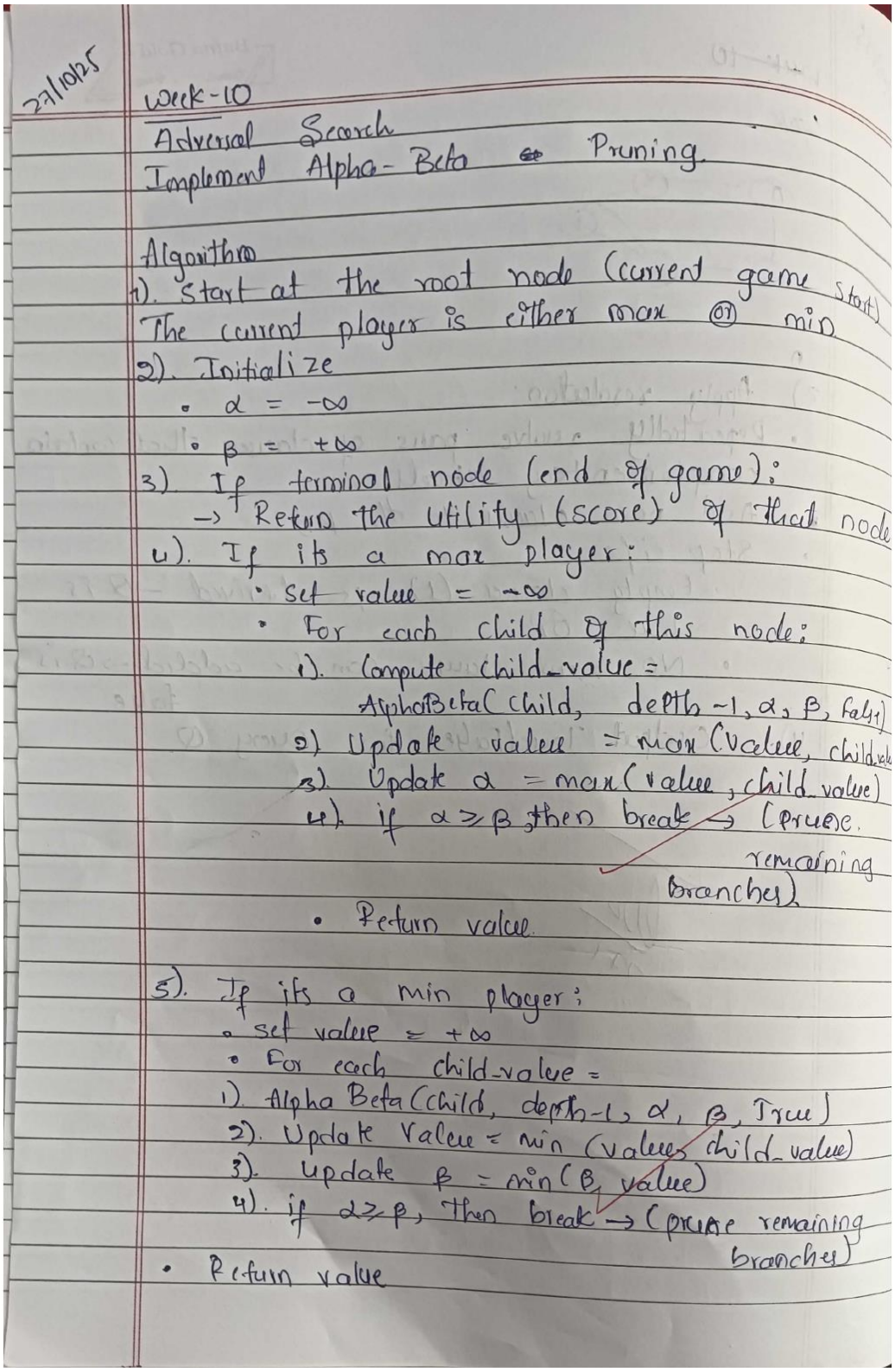
OUPUT:

```
=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: P
Enter clause 2: ~P v Q
Enter clause 3: ~Q
Enter query to prove: Q
[Step 1] R1 = Resolve(C1, C2) → {'Q'}
[Step 2] R2 = Resolve(C2, C4) → {'~P'}
[Step 3] R3 = Resolve(C1, R2) → {}

✅ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R3: {}
    C1
    R2: {'~P'}
        C2
        C4
True
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



27/10/25

Week-10

Adversal Search

Implement Alpha-Beta Pruning.

Algorithm

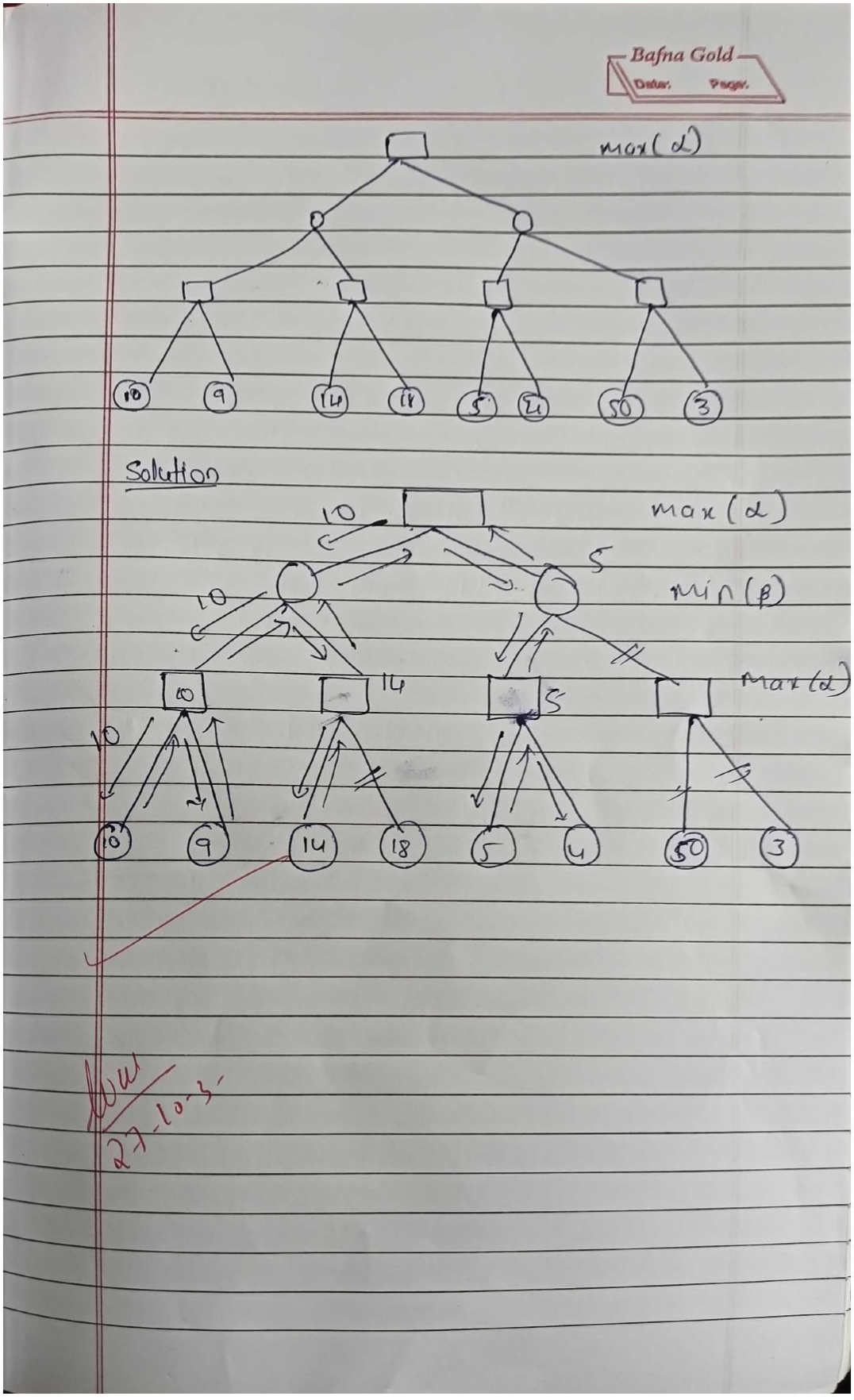
1). Start at the root node (current game state)
The current player is either max or min

2). Initialize
- $\alpha = -\infty$
- $\beta = +\infty$

3). If terminal node (end of game):
→ Return the utility (score) of that node

4). If its a max player:
- set value = $-\infty$
- For each child of this node:
  1). Compute child_value =
  AlphaBeta(child, depth-1, α, β, False)
  2) Update value = max(value, child_value)
  3). Update α = max(value, child_value)
  4). if $\alpha \geq \beta$, then break → (prune remaining branches)
- Return value

5). If its a min player:
- set value = $+\infty$
- For each child_value =
1). AlphaBeta(child, depth-1, α, β, True)
2). Update value = min(value, child_value)
3). Update β = min(β, value)
4). if $\alpha \geq \beta$, then break → (prune remaining branches)
- Return value


Bafna Gold
max (α)
10
9
14
18
5
4
50
3
Solution
max (α)
10
5
min (β)
10
14
5
Max (α)
10
9
14
18
5
4
50
3
27-10-3-

Code:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.value = None
        self.pruned = False

def alpha_beta(node, depth, maximizing, values, alpha, beta, index):
    # Terminal node
    if depth == 3:
        node.value = values[index[0]]
        index[0] += 1
        return node.value

    if maximizing:
        best = float('-inf')
        for i in range(2): # 2 children
            child = Node(f"{node.name} {i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, False, values, alpha, beta, index)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best
    else:
        best = float('inf')
```

```
        for i in range(2):
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, True, values, alpha, beta, index)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best

def print_tree(node, indent=0):
    prune_mark = " [PRUNED]" if node.pruned else ""
    val = f" = {node.value}" if node.value is not None else ""
    print(" " * indent + f"{node.name}{val}{prune_mark}")
    for child in node.children:
        print_tree(child, indent + 4)

# --- main ---
print("=== Alpha-Beta Pruning with Tree ===")
values = list(map(int, input("Enter 8 leaf node values separated by spaces: ").split()))

root = Node("R")
alpha_beta(root, 0, True, values, float('-inf'), float('inf'), [0])

print("\n--- Game Tree ---")
print_tree(root)

print("\nOptimal Value at Root:", root.value)
```

OUTPUT:

```
=== Alpha-Beta Pruning with Tree ===
Enter 8 leaf node values separated by spaces: 3 5 6 9 1 2 0 7

--- Game Tree ---
R = 5
    R0 = 5
        R00 = 5
            R000 = 3
            R001 = 5
        R01 = 6 [PRUNED]
            R010 = 6
    R1 = 2 [PRUNED]
        R10 = 9
            R100 = 9
            R101 = 1
        R11 = 2
            R110 = 2
            R111 = 0

Optimal Value at Root: 5
```

INDEX

Name : Srashti Sunkad

Standard Section F Roll No. 341

Subject Artificial Intelligence.

Sl No.	Date	Title	Page No.	Teacher Sign / Remarks
1	18/8/25	Tic-Tac-Toe	10	
2	25/8/25	Vacuum-Cleaner.	10	
3	1/09/25	Solving 8-Puzzel Using BFS	10	
	1/09/25	Solving 8-Puzzel using DFS		
	1/09/25	Solving 8-Puzzel using IDS		
4.	8/9/25.	A* Algorithm - Manhattan	10	
		A* Algorithm - Miss Placed tiles.		
5	15/09/25	Hill Climbing	10	
	15/09/25	Simulated Annealing.		
6	22/9/25	Propositional logic.	10	
7	13/10/25	First Order logic [Unification].		
8	13/10/25	Forward chaining/Reasoning.		
9	27/10/25	Proving query using Resolution	10	
10	27/10/25.	Adversarial Search.		

Completed