VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ROHAN VATS (1BM23CS273)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **ROHAN VATS (1BM23CS273),** 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.

Dr K. R Mamatha Assistant Professor Department of CSE, BMSCE	Dr. Kavita Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/rvats15/Artificial-Intelligence_1BM23CS273

Program 1

Implement Tic –Tac –Toe Game Algorithm:

25/8/25

Tic-tac-toe implementation (25-08-25)

steps

1. Initialize Board: create a 3x3 board with empty spaces.

2. Define Functions: is winner (board, player) → checks if player ('x' or 'o') has won.

is_draw (board) → checks if all positions are filled with no winner.

available_moves(board) → returns list of empty positions.

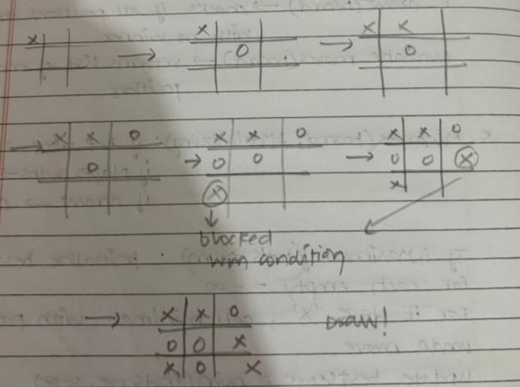
3. Minimax (board, isMaximizing): if bot wins → return
if player wins → return
if draw → 0

If isMaximizing (bot's turn) : initialize bestScore = -∞
For each empty
For it place 'x', call minimax with False
undo move
update bestscore = min(bestscore, score)
Return bestscore.

Else (player's score) : initialize bestScore = ∞,
bestmove = None
For each empty position:
Place 'x', calculate score = minimax (board, False)
undo move

if score > bestscore, update bestscore and bestMove
Place 'x' at bestMove

5. Game Loop:
while no win and no draw:
Player move → Input position, place 'O' if empty
check win/draw
Bot move → find best move using minimax, place 'x'
check win/draw.

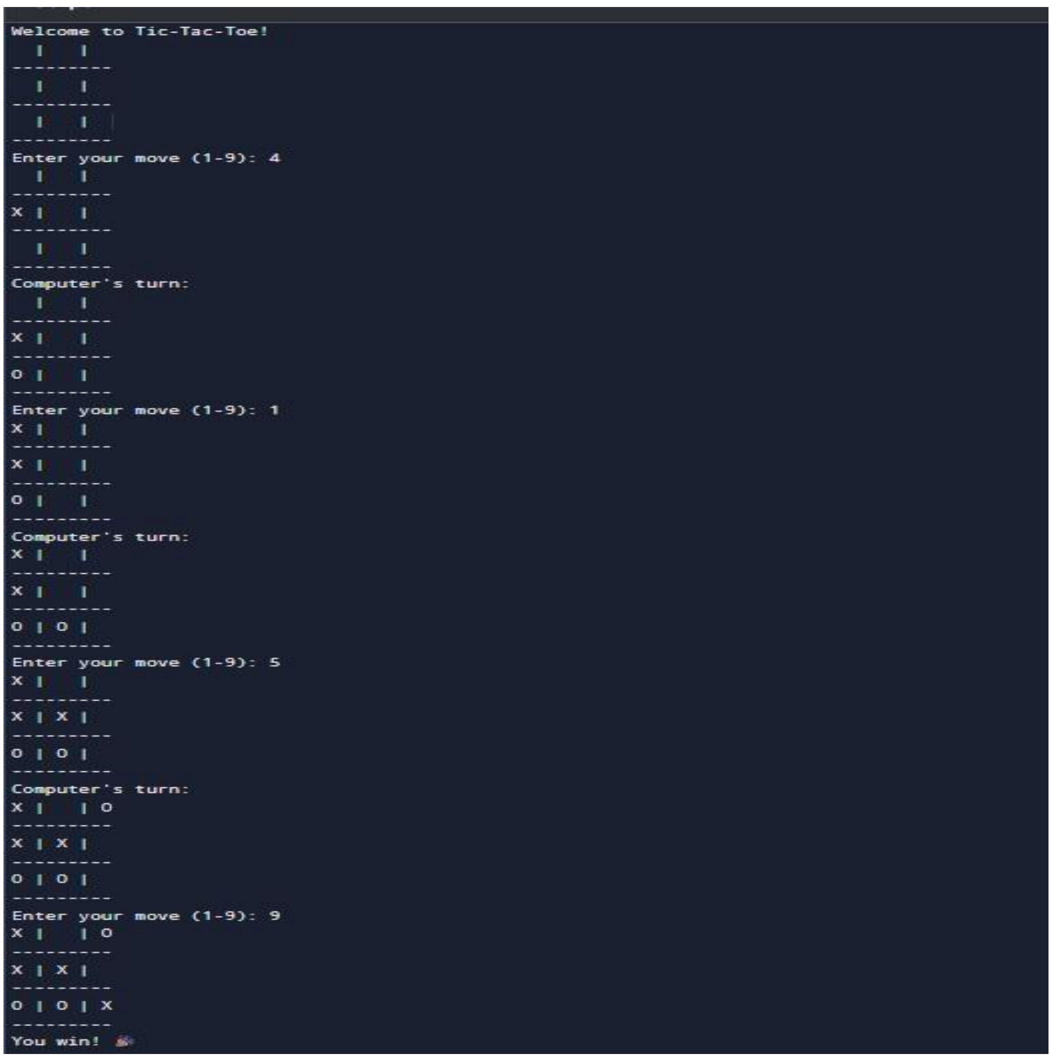


Code:

```
import random
board = [' ' for _ in range(9)]

def print_board():
print()     for i in
range(3):
       print(" " + " | ".join(board[i*3:(i+1)*3]))
if i < 2:
          print("---+---+---")
print() def
check_winner(player):
win_conditions = [
       [0,1,2], [3,4,5], [6,7,8],
       [0,3,6], [1,4,7], [2,5,8],
       [0,4,8], [2,4,6]
   ]
   for cond in win_conditions:         if
all(board[i] == player for i in cond):
          return True
return False def
is_full():
   return all(cell != ' ' for cell in board)
def player_move():     while True:
try:
          move = int(input("Enter your move (1-9): ")) - 1
if move < 0 or move >= 9:
             print("Invalid move. Choose between 1-9.")
elif board[move] != ' ':
             print("That spot is taken.")
else:
             board[move] = 'X'
break        except
ValueError:
          print("Please enter a valid number.") def
ai_move():
   empty_spots = [i for i, val in enumerate(board) if val == ' ']
   move = random.choice(empty_spots)
   board[move] = 'O'     print(f"System placed 'O' in
position {move+1}") def play_game():
print("Welcome to Tic Tac Toe!")
   print_board()     while
True:        player_move()
```

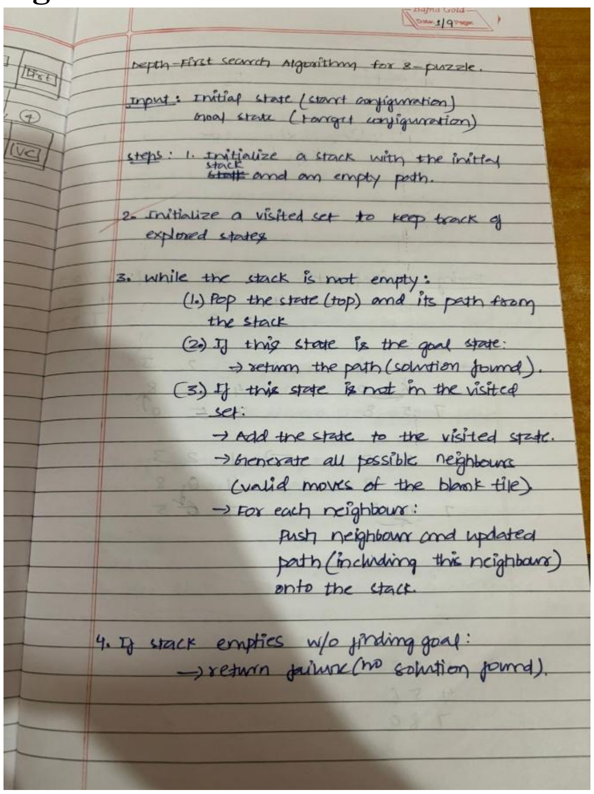
```
print_board()         if
check_winner('X'):
            print("Congratulations! You win!")
break         if is_full():             print("It's a
tie!")             break         ai_move()
print_board()         if check_winner('O'):
            print("System wins. Better luck next time!")
break         if is_full():             print("It's a tie!")
            break if __name__
== "__main__":
    play_game()
```

ScreenShots:



Program 2:

Solve 8 puzzle problems.
Algorithm:


Depth-First Search Algorithm for 8-puzzle.

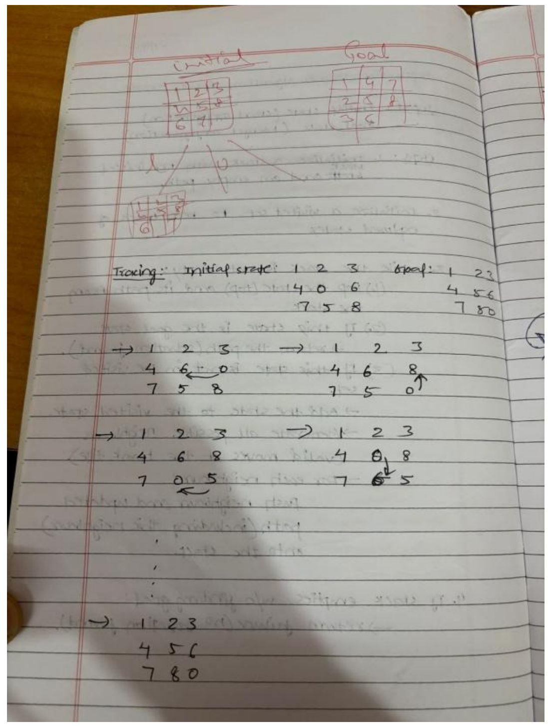
Input: Initial state (start configuration)
Goal state (target configuration)

Steps: 1. Initialize a stack with the initial state and an empty path.

2. Initialize a visited set to keep track of explored states.

3. While the stack is not empty:
(1.) Pop the state (top) and its path from the stack
(2) If this state is the goal state:
→ return the path (solution found).
(3.) If this state is not in the visited set:
→ Add the state to the visited state.
→ Generate all possible neighbours (valid moves of the blank tile)
→ For each neighbour:
Push neighbour and updated path (including this neighbour) onto the stack.

4. If stack empties w/o finding goal:
→ return failure (no solution found).



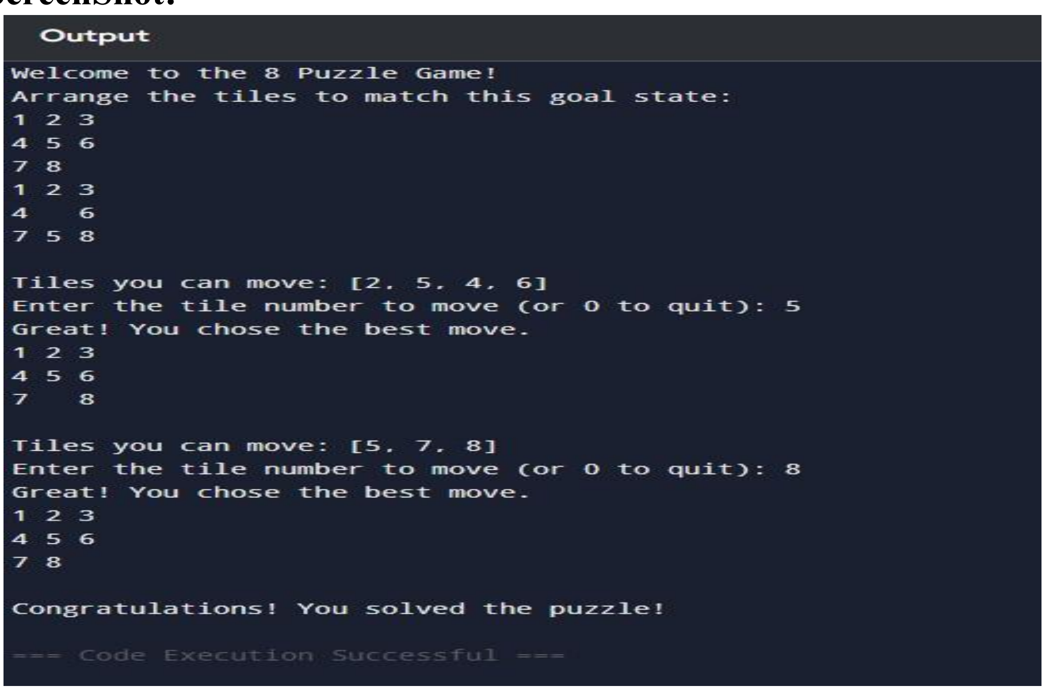
Code:

```
import copy def
print_board(board):
for row in board:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
print() def find_zero(board):      for i in range(3):
for j in range(3):               if board[i][j] == 0:
                return i, j def
is_solved(board):
    solved = [1,2,3,4,5,6,7,8,0]
    flat = [num for row in board for num in row]
    return flat == solved def
valid_moves(zero_pos):      i, j =
zero_pos      moves = []      if i > 0:
moves.append((i-1, j))      if i < 2:
```

```
moves.append((i+1, j))     if j > 0:
moves.append((i, j-1))     if j < 2:
moves.append((i, j+1))     return
moves def
correct_tiles_count(board):
    """Count how many tiles are in their correct position."""
count = 0     goal = [1,2,3,4,5,6,7,8,0]     flat = [num for
row in board for num in row]     for i in range(9):         if
flat[i] != 0 and flat[i] == goal[i]:
            count += 1     return count def
get_user_move(board):     zero_pos =
find_zero(board)     moves =
valid_moves(zero_pos)     movable_tiles =
[board[i][j] for (i,j) in moves]     print(f"Tiles
you can move: {movable_tiles}")     while True:
try:
            move = int(input("Enter the tile number to move (or 0 to quit): "))
if move == 0:                return None            if move in movable_tiles:
                return move
else:
                print("Invalid tile. Please choose a tile adjacent to the empty space.")
except ValueError:
            print("Please enter a valid number.") def
evaluate_move(board, tile):
    """Compare user move to all possible moves and tell if it's best/worst."""
zero_pos = find_zero(board)     moves = valid_moves(zero_pos)
movable_tiles = [board[i][j] for (i,j) in moves]     scores = {}     for t in
movable_tiles:
        temp_board = copy.deepcopy(board)
make_move(temp_board, t)         scores[t] =
correct_tiles_count(temp_board)     user_score =
scores[tile]     best_score = max(scores.values())
worst_score = min(scores.values())
    if user_score == best_score and user_score ==
worst_score:
        return "Your move is the only possible move."
elif user_score == best_score:
        return "Great! You chose the best move."
elif user_score == worst_score:
        return "Oops! You chose the worst move."
else:
        return "Your move is neither the best nor the worst." def
make_move(board, tile):
```

```
    zero_i, zero_j = find_zero(board)      for
i, j in valid_moves((zero_i, zero_j)):
if board[i][j] == tile:
            board[zero_i][zero_j], board[i][j] = board[i][j], board[zero_i][zero_j]
return def main():     board = [         [1, 2, 3],
        [4, 0, 6],
        [7, 5, 8]
    ]
    print("Welcome to the 8 Puzzle Game!")
print("Arrange the tiles to match this goal state:")
print("1 2 3\n4 5 6\n7 8  ")     while True:
        print_board(board)
if is_solved(board):
            print("Congratulations! You solved the puzzle!")
break
        move = get_user_move(board)
if move is None:
            print("Game exited. Goodbye!")
break
        feedback = evaluate_move(board, move)
       print(feedback)
make_move(board, move) if
__name__ == "__main__":
main()
```

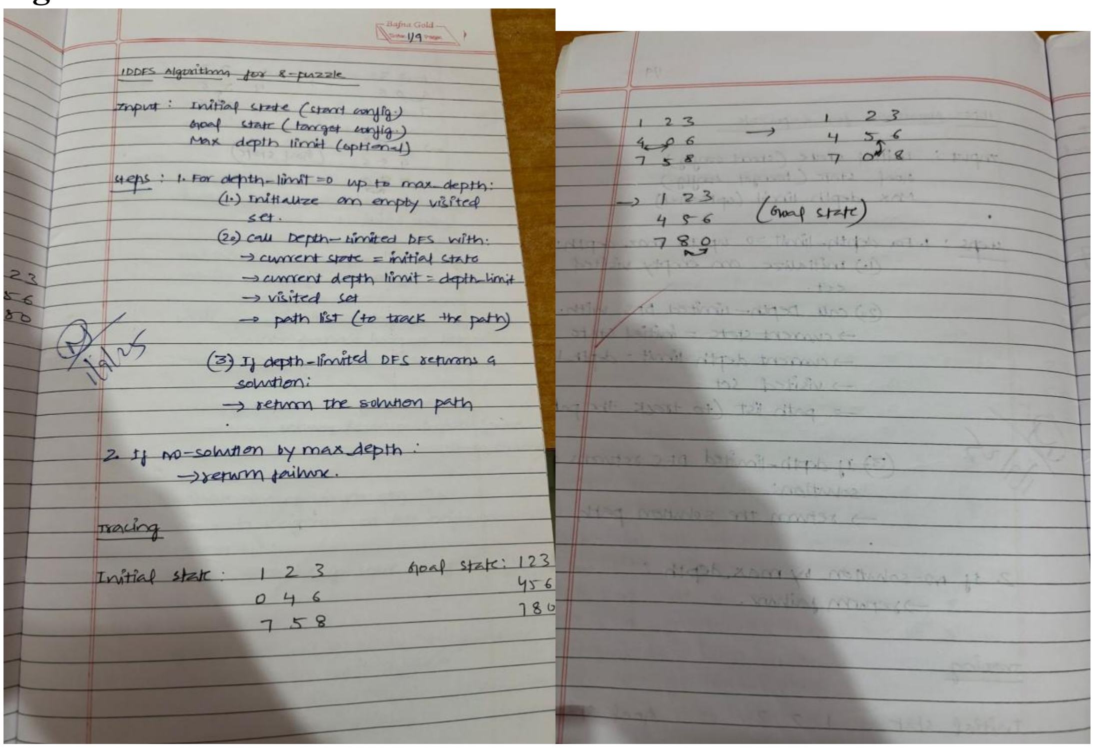
ScreenShot:



```
Output
Welcome to the 8 Puzzle Game!
Arrange the tiles to match this goal state:
1 2 3
4 5 6
7 8
1 2 3
4   6
7 5 8

Tiles you can move: [2, 5, 4, 6]
Enter the tile number to move (or 0 to quit): 5
Great! You chose the best move.
1 2 3
4 5 6
7   8

Tiles you can move: [5, 7, 8]
Enter the tile number to move (or 0 to quit): 8
Great! You chose the best move.
1 2 3
4 5 6
7 8

Congratulations! You solved the puzzle!

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

Program 3:

Implement Iterative deepening search algorithm.
Algorithm:


IDDFS Algorithm for 8-puzzle

Input: Initial state (start config)
Goal state (target config)
Max depth limit (optional)

Steps: 1. For depth-limit = 0 up to max-depth:
(1.) Initialize an empty visited set.
(2.) Call Depth-Limited DFS with:
→ current state = initial state
→ current depth limit = depth-limit
→ visited set
→ path list (to track the path)
(3) If depth-limited DFS returns a solution:
→ return the solution path

2. If no-solution by max depth:
→ return failure.

Tracing

Initial state: 1 2 3 / 0 4 6 / 7 5 8
Goal state: 1 2 3 / 4 5 6 / 7 8 0

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

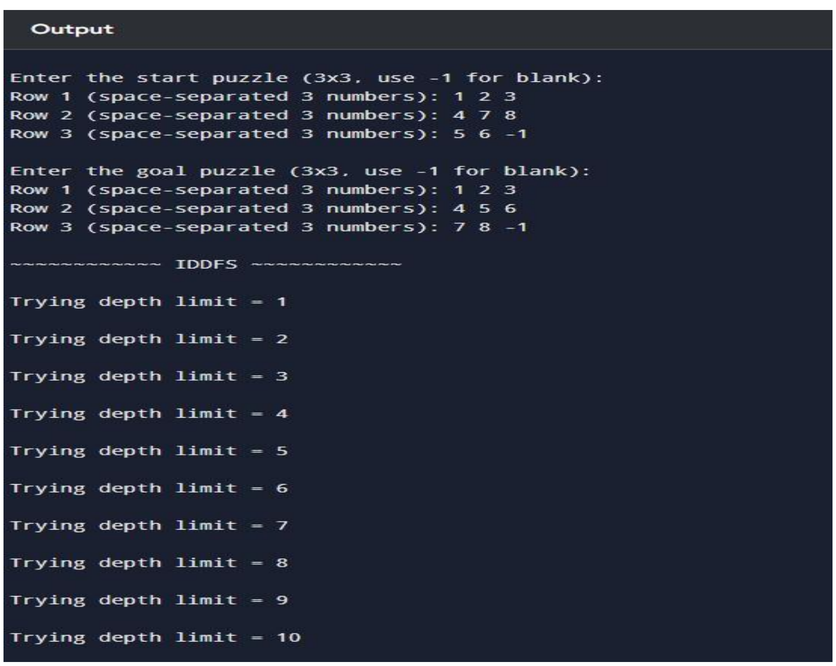
→ 1 2 3 / 4 5 6 / 7 8 0 (Goal state)

Code:

```
import copy def
get_puzzle(name):
   print(f"\nEnter the {name} puzzle (3x3, use -1 for blank):")
puzzle = []     for i in range(3):
        row = list(map(int, input(f"Row {i+1} (space-separated 3 numbers): ").split()))
puzzle.append(row)     return puzzle def move(temp, movement):     for i in
range(3):         for j in range(3):             if temp[i][j] == -1:
                if movement == "up" and i > 0:
                    temp[i][j], temp[i-1][j] = temp[i-1][j], temp[i][j]
elif movement == "down" and i < 2:
                    temp[i][j], temp[i+1][j] = temp[i+1][j], temp[i][j]
elif movement == "left" and j > 0:
                    temp[i][j], temp[i][j-1] = temp[i][j-1], temp[i][j]
elif movement == "right" and j < 2:
                    temp[i][j], temp[i][j+1] = temp[i][j+1], temp[i][j]
return temp     return temp def dls(puzzle, depth, limit,
last_move, goal):     if puzzle == goal:
        return True, [puzzle], []     if depth >= limit:         return False, [], []     for move_dir,
opposite in [("up","down"), ("left","right"), ("down","up"), ("right","left")]:
```

```
        if last_move == opposite:  # avoid direct backtracking
continue
        temp = copy.deepcopy(puzzle)         new_state = move(temp,
move_dir)         if new_state != puzzle:  # valid move              found, path,
moves = dls(new_state, depth+1, limit, move_dir, goal)              if found:
                return True, [puzzle] + path, [move_dir] + moves
return False, [], [] def ids(start, goal):
    for limit in range(1, 50):  # reasonable max depth
print(f"\nTrying depth limit = {limit}")         found,
path, moves = dls(start, 0, limit, None, goal)         if
found:
            print("Solution found!")
for step in path:                 print(step)
print("Moves:", moves)
print("Path cost =", len(path)-1)
return
    print(" Solution not found within depth limit.")
start_puzzle = get_puzzle("start") goal_puzzle =
get_puzzle("goal") print("\n~~~~~~~~~~~~
IDDFS ~~~~~~~~~~~~") ids(start_puzzle,
goal_puzzle)
```

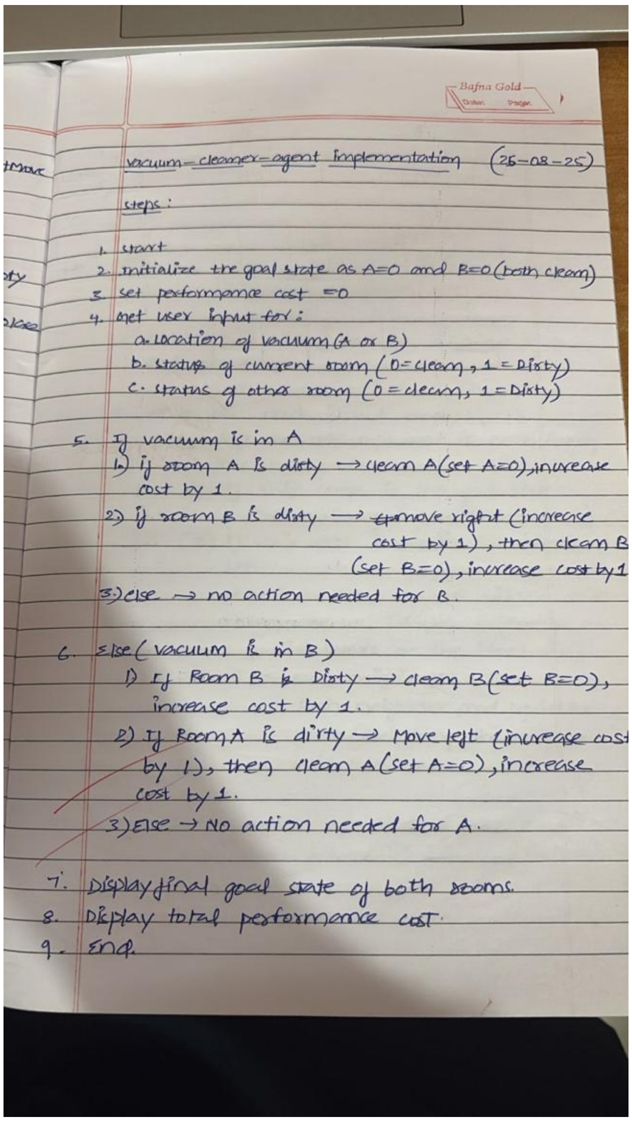
ScreenShot:



```
Output

Enter the start puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 7 8
Row 3 (space-separated 3 numbers): 5 6 -1

Enter the goal puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 5 6
Row 3 (space-separated 3 numbers): 7 8 -1

~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~

Trying depth limit = 1

Trying depth limit = 2

Trying depth limit = 3

Trying depth limit = 4

Trying depth limit = 5

Trying depth limit = 6

Trying depth limit = 7

Trying depth limit = 8

Trying depth limit = 9

Trying depth limit = 10
```

Program 4:

Implement a vacuum cleaner agent.
Algorithm:



Vacuum-cleaner-agent implementation (25-08-25)

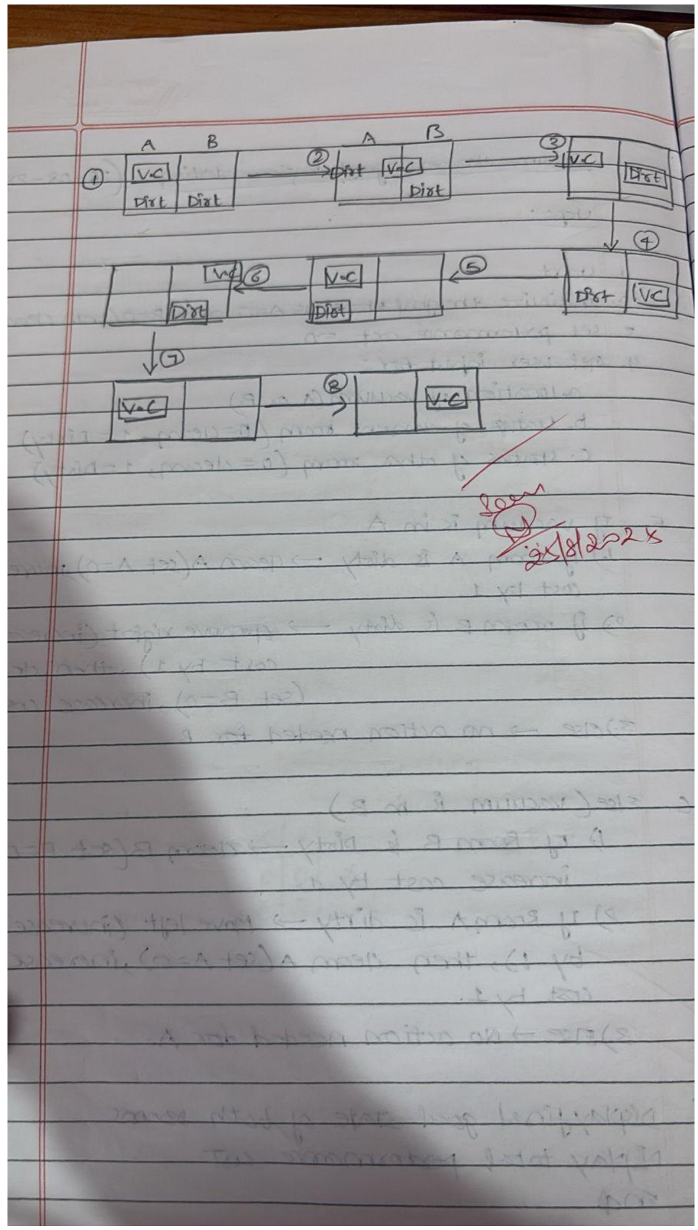
steps:

1. start
2. Initialize the goal state as A=0 and B=0 (both clean)
3. set performance cost =0
4. Get user input for:
   a. Location of vacuum (A or B)
   b. status of current room (0=clean, 1=Dirty)
   c. status of other room (0=clean, 1=Dirty)

5. If vacuum is in A
   1.) if room A is dirty → clean A (set A=0), increase cost by 1.
   2) if room B is dirty → ~~move~~ move right (increase cost by 1), then clean B (set B=0), increase cost by 1
   3.) else → no action needed for B.

6. Else (vacuum is in B)
   1) If Room B is Dirty → clean B (set B=0), increase cost by 1.
   2) If Room A is dirty → Move left (increase cost by 1), then clean A (set A=0), increase cost by 1.
   3) Else → No action needed for A.

7. Display final goal state of both rooms.
8. Display total performance cost.
9. End.



Code:

```
def show_rooms_status(rooms):      for
room_number, status in rooms.items():
        print(f"Room {room_number}: {'Clean' if status else 'Dirty'}")
def clean_room(rooms, room_number):      if rooms[room_number]:
```

```
        print(f"Room {room_number} is already clean.")
else:
        print(f"Cleaning room {room_number}...")
rooms[room_number] = True          print(f"Room
{room_number} is now clean!") def
clean_all_rooms(rooms):      print("Initial room
statuses:")      show_rooms_status(rooms)
    print("\nStarting cleaning process...\n")
for room_number in rooms:
        clean_room(rooms, room_number)
        print()
    print("Final room statuses:")
show_rooms_status(rooms) if
__name__ == "__main__":
    rooms = {
1: False,
        2: True,
        3: False,
        4: False
    }
    clean_all_rooms(rooms)
```

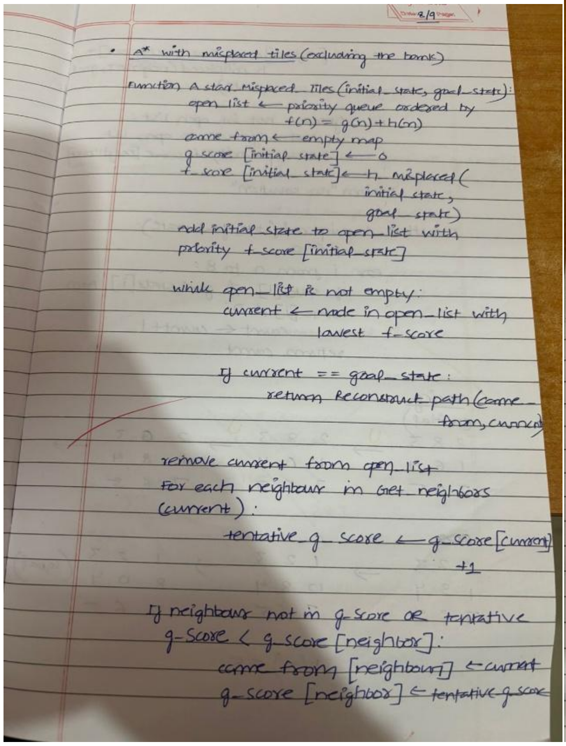
ScreenShot:

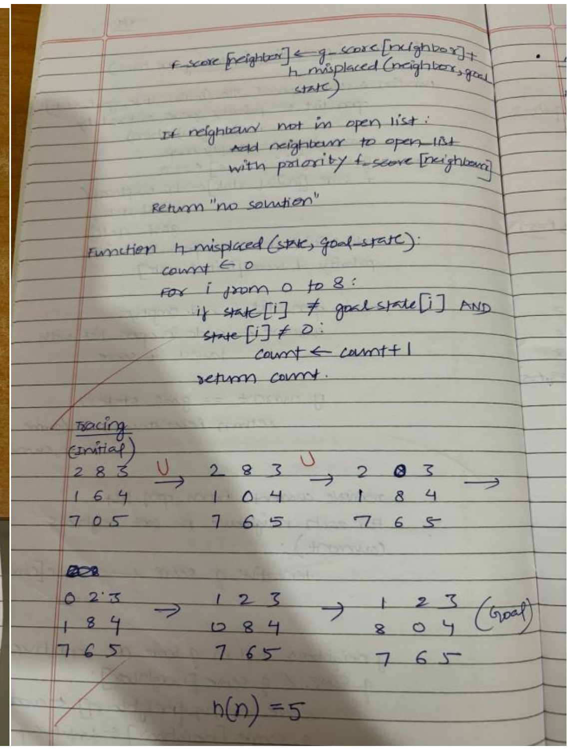
```
Output
Initial room statuses:
Room 1: Dirty
Room 2: Clean
Room 3: Dirty
Room 4: Dirty

Starting cleaning process...

Cleaning room 1...
Room 1 is now clean!

Room 2 is already clean.

Cleaning room 3...
Room 3 is now clean!

Cleaning room 4...
Room 4 is now clean!

Final room statuses:
Room 1: Clean
Room 2: Clean
Room 3: Clean
Room 4: Clean

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

Program 5:

Implement A* search algorithm.

Algorithm:





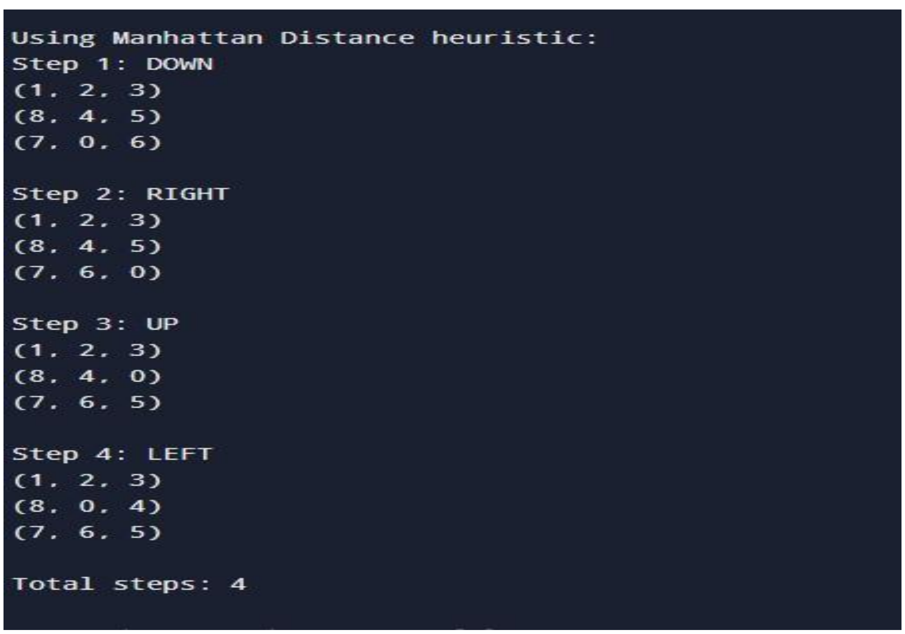
Code:

```
from heapq import heappush, heappop
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
direction_names = ["UP", "DOWN", "LEFT", "RIGHT"] 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_state[i][j]:
                count += 1     return
count def
manhattan_distance(state):
```

```
   distance = 0     for i
in range(3):        for j
in range(3):
tile = state[i][j]
if tile != 0:
             goal_x, goal_y = divmod(tile - 1, 3)
distance += abs(i - goal_x) + abs(j - goal_y)     return
distance def get_neighbors_with_actions(state):
   neighbors = []     for i
in range(3):        for j in
range(3):           if
state[i][j] == 0:
             x, y = i, j              break     for (dx, dy), action
in zip(directions, direction_names):
       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((new_state, action))     return neighbors def
state_to_tuple(state):
   return tuple(tuple(row) for row in state) def
reconstruct_path(came_from, current):
   actions = []     states = []     while current in came_from:
prev_state, action = came_from[current]
actions.append(action)        states.append(current)
current = prev_state     states.append(current)
actions.reverse()     states.reverse()     return actions, states
def a_star_search_with_steps(initial_state, heuristic_func):
   open_list = []
closed_set = set()
   g_score = {state_to_tuple(initial_state): 0}     f_score =
{state_to_tuple(initial_state): heuristic_func(initial_state)}
came_from = {}
   heappush(open_list, (f_score[state_to_tuple(initial_state)], initial_state))
   while open_list:
       _, current_state = heappop(open_list)
current_t = state_to_tuple(current_state)        if
current_state == goal_state:
          return reconstruct_path(came_from, current_t)
closed_set.add(current_t)        for neighbor, action in
get_neighbors_with_actions(current_state):
          neighbor_t = state_to_tuple(neighbor)
if neighbor_t in closed_set:
             continue           tentative_g = g_score[current_t] + 1           if
neighbor_t not in g_score or tentative_g < g_score[neighbor_t]:
```

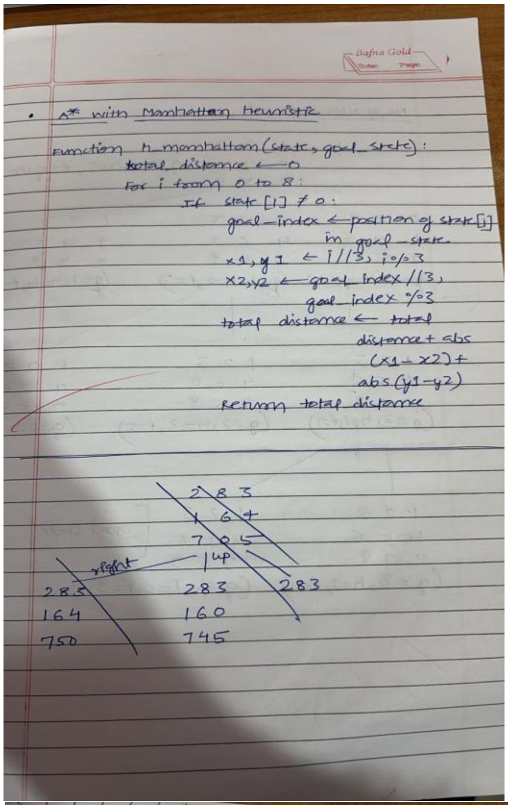
```
came_from[neighbor_t] = (current_t, action)
g_score[neighbor_t] = tentative_g                f_score[neighbor_t] =
tentative_g + heuristic_func(neighbor)              heappush(open_list,
(f_score[neighbor_t], neighbor))     return None, None def
print_path(actions, states):     for i, (action, state) in
enumerate(zip(actions, states[1:]), 1):
    print(f"Step {i}: {action}")
for row in state:
print(row)        print()
initial_state = [     [1, 2, 3],
   [8, 0, 5],
   [7, 4, 6]
]
print("Using Misplaced Tiles heuristic:") actions, states =
a_star_search_with_steps(initial_state, misplaced_tiles) if actions:
   print_path(actions, states)
print("Total steps:", len(actions)) else:
   print("No solution found.") print("\nUsing Manhattan Distance
heuristic:") actions, states = a_star_search_with_steps(initial_state,
manhattan_distance) if actions:
   print_path(actions, states)
print("Total steps:", len(actions))
else:     print("No solution found.")
```

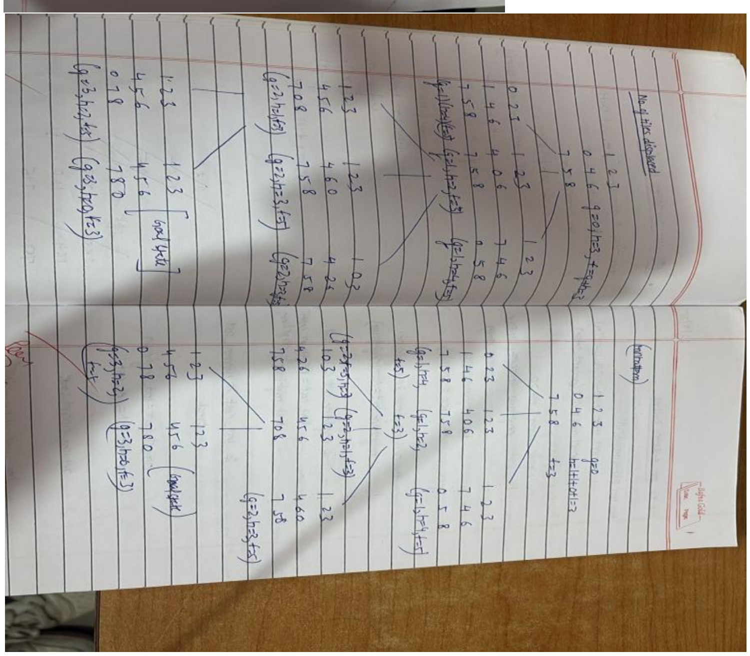
ScreenShot:



```
Using Manhattan Distance heuristic:
Step 1: DOWN
(1, 2, 3)
(8, 4, 5)
(7, 0, 6)

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

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

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

Total steps: 4
```

b. Implement Hill Climbing Algorithm Algorithm:





Code:

```
import random import time def
generate_initial_state(n=4):
   return [random.randint(0, n - 1) for _ in range(n)] def
calculate_conflicts(state):
   conflicts = 0     n =
len(state)     for i in
range(n):         for j in
range(i + 1, n):             if
state[i] == state[j]:
             conflicts += 1            if
abs(state[i] - state[j]) == abs(i - j):
             conflicts += 1
return conflicts def
get_neighbors(state):
   neighbors = []     n = len(state)
for col in range(n):         for row in
range(n):             if state[col] != row:
neighbor = state.copy()
neighbor[col] = row
neighbors.append(neighbor)     return
neighbors def print_board(state):
   n = len(state)     for row
in range(n):         line = ""
for col in range(n):
if state[col] == row:
             line += "Q "            else:                line += ". "         print(line)     print() def
hill_climbing_with_steps(n=4, max_restarts=100):     for restart in range(max_restarts):
current = generate_initial_state(n)         step = 0         print(f"Restart #{restart+1}: Initial
state (Conflicts = {calculate_conflicts(current)})")         print_board(current)         while
True:
          current_conflicts = calculate_conflicts(current)
if current_conflicts == 0:
             print(f"Solution found in {step} steps!")
return current
          neighbors = get_neighbors(current)             neighbor_conflicts =
[calculate_conflicts(nbr) for nbr in neighbors]             min_conflict =
min(neighbor_conflicts)             if min_conflict >= current_conflicts:
             print("Reached local minimum, restarting...\n")
break
          best_neighbor = neighbors[neighbor_conflicts.index(min_conflict)]
step += 1             print(f"Step {step}: Conflicts = {min_conflict}")
print_board(best_neighbor)
```

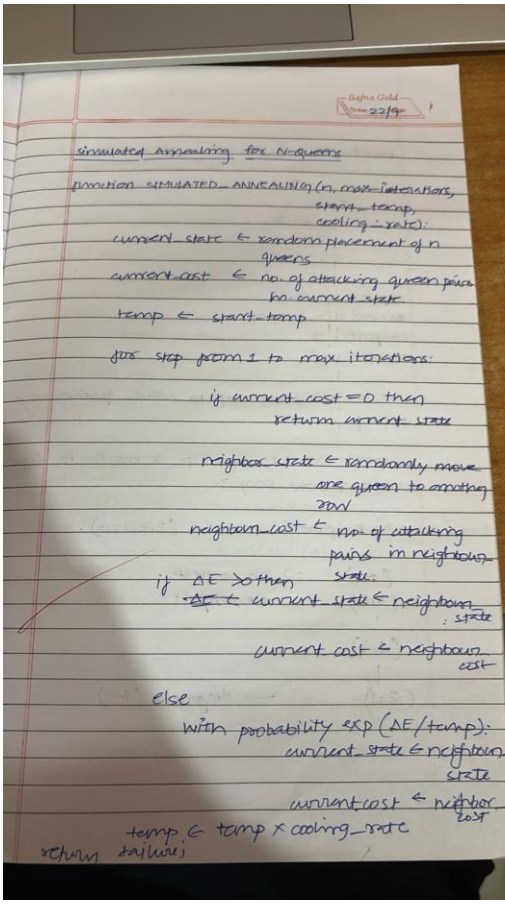
current = best_neighbor
return None solution =
hill_climbing_with_steps() if
solution:
print("Final Solution:")
print_board(solution) else:
print("No solution found.")

ScreenShot:

```
Output
Step 2: Temp=95.000, Cost=5
Step 3: Temp=90.250, Cost=2
Step 4: Temp=85.737, Cost=2
Step 5: Temp=81.451, Cost=3
Step 6: Temp=77.378, Cost=4
Step 7: Temp=73.509, Cost=4
Step 8: Temp=69.834, Cost=4
Step 9: Temp=66.342, Cost=4
Step 10: Temp=63.025, Cost=3
Step 11: Temp=59.874, Cost=5
Step 12: Temp=56.880, Cost=4
Step 13: Temp=54.036, Cost=4
Step 14: Temp=51.334, Cost=4
Step 15: Temp=48.767, Cost=4
Step 16: Temp=46.329, Cost=4
Step 17: Temp=44.013, Cost=3
Step 18: Temp=41.812, Cost=2
Step 19: Temp=39.721, Cost=3
Step 20: Temp=37.735, Cost=3
Step 21: Temp=35.849, Cost=3
Step 22: Temp=34.056, Cost=3
Step 23: Temp=32.353, Cost=0

Final Board:
. Q . .
. . . Q
Q . . .
. . Q .

Final Cost: 0
Goal State Reached!
```

Program 6:

Write a program to implement Simulated Annealing Algorithm



Bafna Gold 22/9

simulated Annealing for N-queens

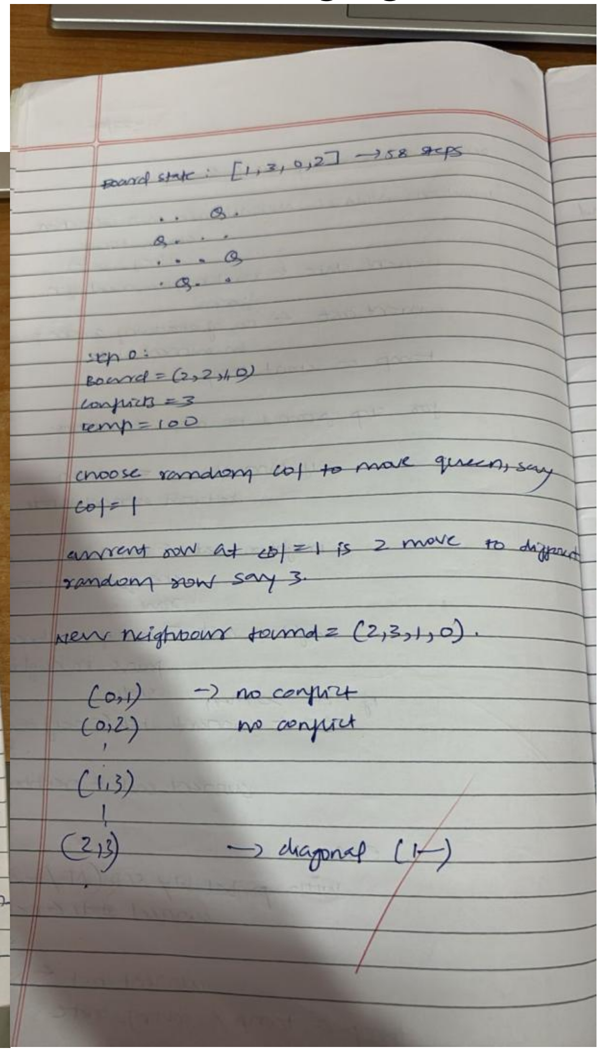
function SIMULATED_ANNEALING(n, max_iterations,
start_temp,
cooling_rate):
current_state ← random placement of n queens
current_cost ← no. of attacking queen pairs in current state
temp ← start_temp

for step from 1 to max iterations:

if current_cost = 0 then
return current state

neighbor_state ← randomly move one queen to another row
neighbour_cost ← no. of attacking pairs in neighbour state.
if ΔE > 0 then
~~ΔE~~ ← current_state ← neighbour_state
current cost ← neighbour_cost

else
with probability exp(ΔE/temp):
current_state ← neighbour_state
current cost ← neighbor_cost
temp ← temp × cooling_rate
return failure;



Board state: [1, 3, 0, 2] → 58 steps

. . Q .
Q . . .
. . . Q
. Q . .

step 0:
Board = (2, 2, 1, 0)
conflicts = 3
temp = 100

choose random col to move queen, say col = 1

current row at col = 1 is 2 move to different random row say 3.

new neighbour found = (2, 3, 1, 0).

(0,1) → no conflict
(0,2) no conflict
⋮
(1,3)
⋮
(2,3) → diagonal (1 ~~–~~)

Code:

```
import random import
math def
print_board(board):
   n = len(board)
for i in range(n):
row = ["Q" if
board[i] == j else
"." for j in range(n)]
print(" ".join(row))
print() def
calculate_cost(boar
d):
```

```
   """Heuristic: number of pairs of queens attacking each other"""
n = len(board)     cost = 0     for i in range(n):         for j in range(i +
1, n):             if board[i] == board[j] or abs(board[i] - board[j]) ==
abs(i - j):
             cost += 1     return
cost def
random_neighbor(board):
   """Generate a random neighboring board by moving one queen"""     n =
len(board)     neighbor = list(board)     row = random.randint(0, n - 1)     col =
random.randint(0, n - 1)     neighbor[row] = col     return neighbor def
simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
   current_board = [random.randint(0, n - 1) for _ in range(n)]
current_cost = calculate_cost(current_board)     temperature =
initial_temp     step = 1     print("Initial Board:")
print_board(current_board)     print(f"Initial Cost:
{current_cost}\n")     while temperature > stopping_temp and
current_cost > 0:
      neighbor = random_neighbor(current_board)        neighbor_cost
= calculate_cost(neighbor)        delta = neighbor_cost - current_cost
if delta < 0 or random.random() < math.exp(-delta / temperature):
         current_board = neighbor            current_cost = neighbor_cost
print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1        temperature *= cooling_rate     print("\nFinal Board:")
print_board(current_board)     print(f"Final Cost: {current_cost}")     if
current_cost == 0:
      print("Goal State Reached!")
else:
      print("Terminated before reaching goal.")
simulated_annealing(4)
```

ScreenShot:

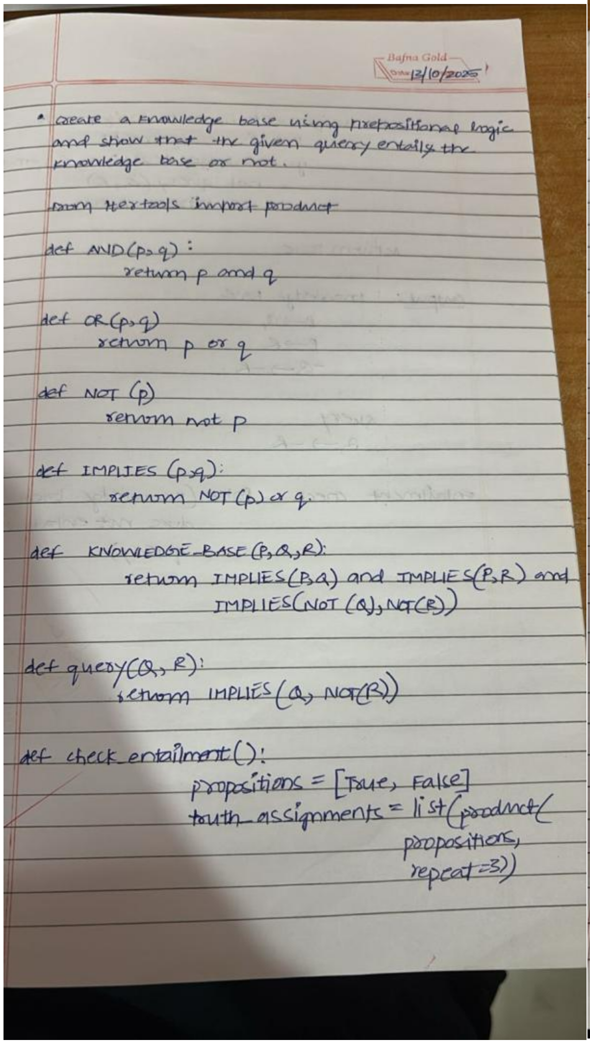
Output

```
Restart #1: Initial state (Conflicts = 2)
. Q Q .
. . . Q
. . . .
Q . . .

Step 1: Conflicts = 1
. Q . .
. . . Q
. . Q .
Q . . .

Reached local minimum, restarting...

Restart #2: Initial state (Conflicts = 2)
. . Q .
Q Q . .
. . . Q
. . . .

Step 1: Conflicts = 0
. . Q .
Q . . .
. . . Q
. Q . .

Solution found in 1 steps!
Final Solution:
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 7:

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

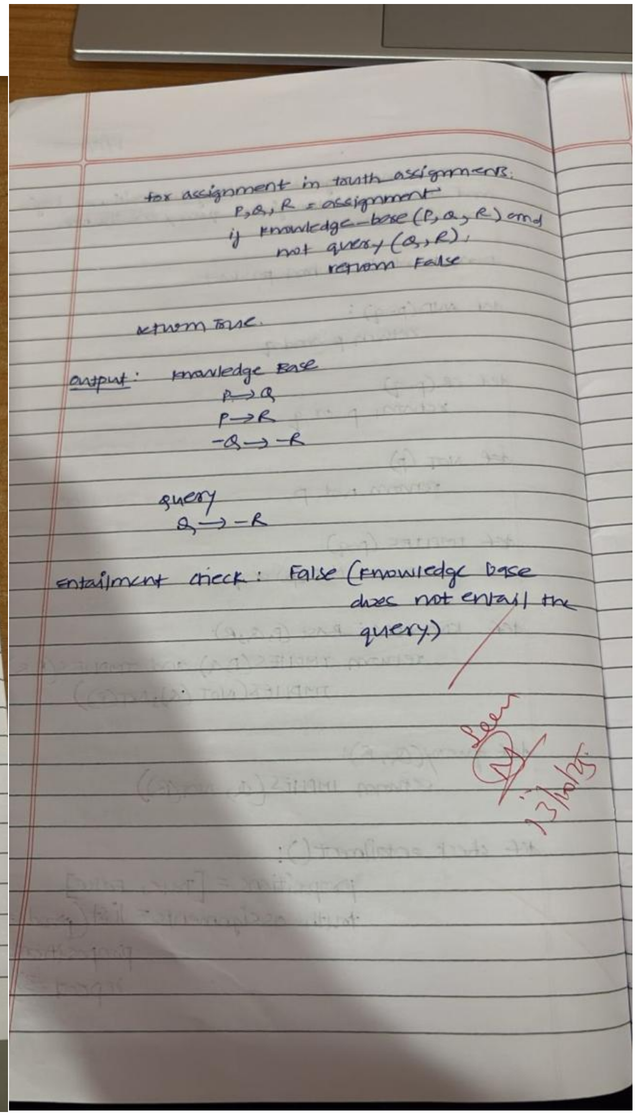
Algorithm:



Bafna Gold
Date: 13/10/2025

• create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

```
from itertools import product

def AND(p, q):
        return p and q

def OR(p, q)
        return p or q

def NOT (p)
        return not p

def IMPLIES (p, q):
        return NOT(p) or q.

def KNOWLEDGE_BASE(P, Q, R):
        return IMPLIES(P,Q) and IMPLIES(P,R) and
               IMPLIES(NOT (Q), NOT(R))

def query(Q, R):
        return IMPLIES(Q, NOT(R))

def check_entailment():
        propositions = [True, False]
        truth_assignments = list(product(
                                  propositions,
                                  repeat=3))
```



```
for assignment in truth_assignments:
        P, Q, R = assignment
        if knowledge_base(P, Q, R) and
              not query(Q, R):
                  return False

return True.
```

output: Knowledge Base
P → Q
P → R
¬Q → ¬R

query
Q → ¬R

Entailment check: False (Knowledge base does not entail the query)

Seen
13/10/25

Code:

import itertools
import pandas as pd variables = ['P', 'Q', 'R'] combinations =
list(itertools.product([False, True], repeat=3)) rows = [] for
(P, Q, R) in combinations:
    s1 = (not Q) or P   s2 = (not
P) or (not Q)   # P → ¬Q   s3 = Q

```
or R              # Q ∨ R     KB = s1
and s2 and s3     entail_R = R
    entail_R_imp_P = (not R) or P
entail_Q_imp_R = (not Q) or R     rows.append({
        'P': P, 'Q': Q, 'R': R,
        'Q → P': s1,
        'P → ¬Q': s2,
        'Q ∨ R': s3,
        'KB True?': KB,
        'R': entail_R,
        'R → P': entail_R_imp_P,
        'Q → R': entail_Q_imp_R
    })
df = pd.DataFrame(rows) print("Truth
Table for Knowledge Base:\n")
print(df.to_string(index=False))
models_true = df[df['KB True?'] == True]
print("\nModels where KB is True:\n")
print(models_true[['P', 'Q', 'R']]) def
entails(column):
    """Check if KB entails the given statement."""
return all(models_true[column]) print("\nEntailment
Results:")
print(f"KB ⊨ R ? {'Yes' if entails('R') else 'No'}")
print(f"KB ⊨ R → P ? {'Yes' if entails('R → P') else 'No'}")
print(f"KB ⊨ Q → R ? {'Yes' if entails('Q → R') else
'No'}")
```

ScreenShot:

```
Output
Truth Table for Knowledge Base:

    P     Q     R  Q → P  P → ¬Q  Q ∨ R  KB True?  R → P  Q → R
False False False   True    True  False     False   True   True
False False  True   True    True   True      True  False   True
False  True False  False    True   True     False   True  False
False  True  True  False    True   True     False  False   True
 True False False   True    True  False     False   True   True
 True False  True   True    True   True      True   True   True
 True  True False   True   False   True     False   True  False
 True  True  True   True   False   True     False   True   True

Models where KB is True:

      P      Q     R
1  False  False  True
5   True  False  True

Entailment Results:
KB ⊨ R ? Yes
KB ⊨ R → P ? No
KB ⊨ Q → R ? Yes

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

Program 8:

Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:

Bafna Gold
Date 17/11

FOL- create a knowledge base consisting of first order logic statements and prove the given query using resolution

```
RESOLUTION (KB, S):
    clauses = CNF(KB) ∪ CNF(¬S)

    loop:
        new = {}

        for each pair of clauses (Ci, Cj) in clauses:
            resolvents = RESOLVE(Ci, Cj)
            if {} in resolvents:
                return PROVED
            new = new ∪ resolvents

        if new ⊆ clauses:
            return NOT PROVED

        clauses = clauses ∪ new
```

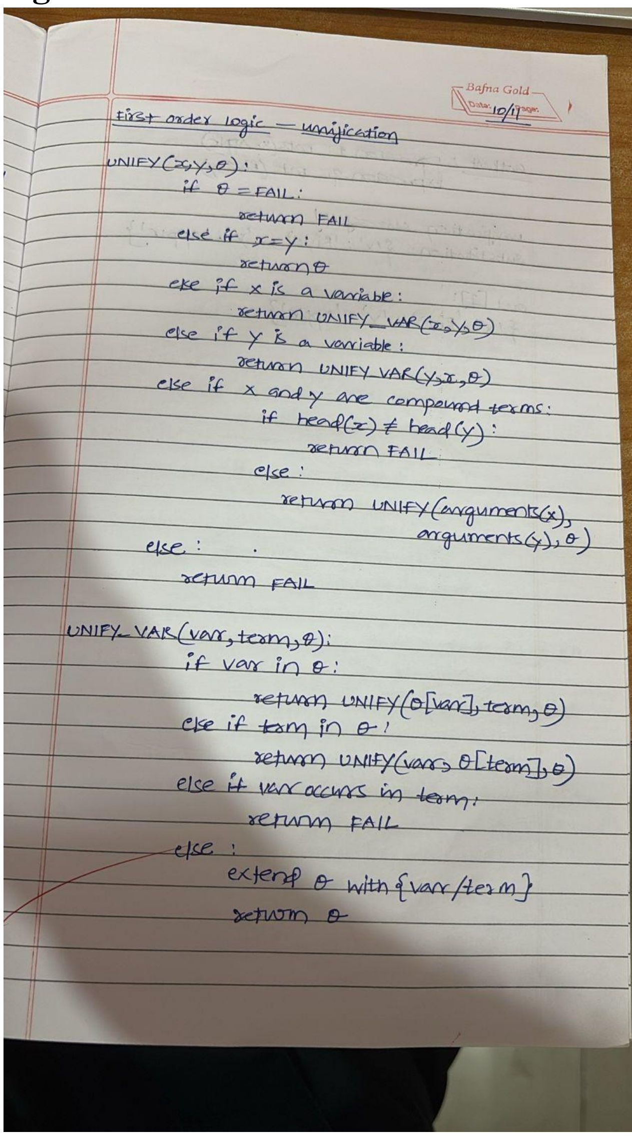
output sample: Initial clauses loaded.
Resolving...

Resolve: ¬Food (x) ∨ Likes (John, x)
AND Food (Peanuts)
=> Likes (John, Peanuts)

Resolve: Likes (John, Peanuts) AND ¬Likes (JOHN, Peanuts)
Empty clause Derived - Proved! => ϕ

Program 9:

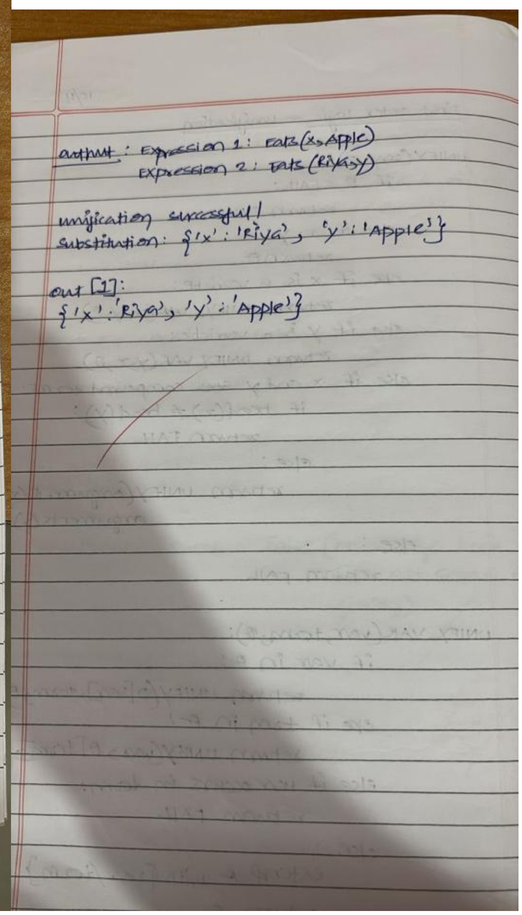
Implement unification in first order logic.
Algorithm:



Bafna Gold
Date: 10/11

first order logic — unification

```
UNIFY(x,y,θ):
    if θ = FAIL:
        return FAIL
    else if x=y:
        return θ
    else if x is a variable:
        return UNIFY_VAR(x,y,θ)
    else if y is a variable:
        return UNIFY VAR(y,x,θ)
    else if x and y are compound terms:
        if head(x) ≠ head(y):
            return FAIL
        else:
            return UNIFY(arguments(x),
                         arguments(y), θ)
    else:
        return FAIL

UNIFY_VAR(var,term,θ):
    if var in θ:
        return UNIFY(θ[var], term, θ)
    else if term in θ:
        return UNIFY(var, θ[term], θ)
    else if var occurs in term:
        return FAIL
    else:
        extend θ with {var/term}
        return θ
```



output: Expression 1: Eats(x, Apple)
Expression 2: Eats(Riya, y)

unification successful!
substitution: {'x': 'Riya', 'y': 'Apple'}

out [1]:
{'x': 'Riya', 'y': 'Apple'}

Code :

```
In [1]:
def is_variable(x):
    return x[0].islower() and x.isalpha()

def get_args(expr):
    name, args = expr.split('(')
    args = args[:-1]   # remove ')'
    return name.strip(), [a.strip() for a in args.split(',')]

def unify(expr1, expr2):
    name1, args1 = get_args(expr1)
    name2, args2 = get_args(expr2)

    print(f"\nExpression 1: {expr1}")
    print(f"Expression 2: {expr2}")


    if name1 != name2 or len(args1) != len(args2):
        print("Unification failed: predicate mismatch")
        return None

    substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1 == a2:
            continue
        elif is_variable(a1):
            substitution[a1] = a2
        elif is_variable(a2):
            substitution[a2] = a1
        else:
            print(f"Unification failed for {a1} and {a2}")
            return None

    print("\nUnification Successful!")
    print("Substitution:", substitution)
    return substitution


expr1 = input("Enter first expression (e.g., Eats(x, Apple)): ")
expr2 = input("Enter second expression (e.g., Eats(Riya, y)): ")

unify(expr1, expr2)
```

```
Expression 1: Eats(x,Apple)
Expression 2: Eats(Riya,y)

Unification Successful!
Substitution: {'x': 'Riya', 'y': 'Apple'}
Out[1]:  {'x': 'Riya', 'y': 'Apple'}
```

Program 10:

Convert a given first order logic statement into Conjunctive Normal Form (CNF).
Algorithm:

Bafna Gold
Date 17/11

FOL- create a knowledge base consisting of first order logic statements and prove the given query using resolution

```
RESOLUTION (KB, S):
    clauses = CNF(KB) ∪ CNF(¬S)

    loop:
      new = { }

      for each pair of clauses (Ci, Cj) in clauses:
          resolvents = RESOLVE(Ci, Cj)
          if {} in resolvents:
              return PROVED
          new = new ∪ resolvents

      if new ⊆ clauses:
          return NOT PROVED

      clauses = clauses ∪ new
```

<u>output sample:</u> Initial clauses loaded.
Resolving..

Resolve: ¬Food (x) ∨ Likes (John, x)
AND Food (Peanuts)
=> Likes (John, Peanuts)

Resolve: Likes (John, Peanuts) AND ¬Likes (JOHN, Peanuts)
Empty clause Derived - Proved! => ϕ

Code :

In [1]:

```python
def negate(literal):
    pred, sign, args = literal
    return (pred, not sign, args)

def apply_subst_literal(literal, subst):
    pred, sign, args = literal
    return (pred, sign, [subst.get(a, a) for a in args])

def unify(x, y, subst):
    if subst is None:
        return None
    if x == y:
        return subst
    if isinstance(x, str) and x.islower():  # variable
        return unify_var(x, y, subst)
    if isinstance(y, str) and y.islower():
        return unify_var(y, x, subst)
    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for a, b in zip(x, y):
            subst = unify(a, b, subst)
        return subst
    return None

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    if x in subst:
        return unify(var, subst[x], subst)
    new = subst.copy()
    new[var] = x
    return new

def resolve(c1, c2):
    resolvents = []
    for l1 in c1:
        for l2 in c2:
            if l1[0] == l2[0] and l1[1] != l2[1]:
                subst = unify(l1[2], l2[2], {})
                if subst is not None:
                    new_clause = (
                        [apply_subst_literal(x, subst) for x in c1 if x != l1] +
                        [apply_subst_literal(x, subst) for x in c2 if x != l2]
                    )
                    # remove duplicates
                    cleaned = []
                    for c in new_clause:
                        if c not in cleaned:
                            cleaned.append(c)
                    resolvents.append(cleaned)
    return resolvents
```

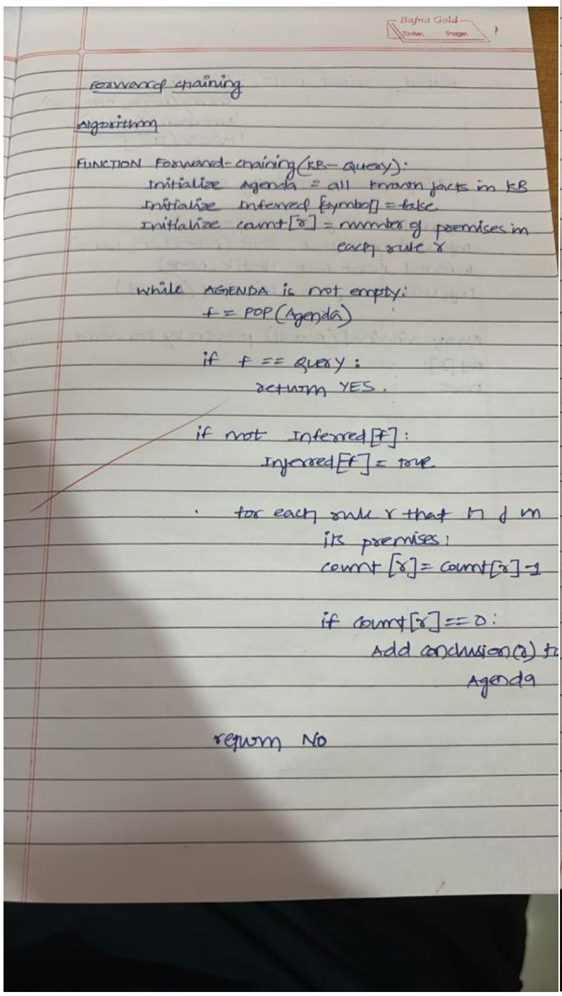
```python
KB = [
    # a. John likes all food:  Food(x) → Likes(John,x)
    [( "Food", False, ["x"] ), ("Likes", True, ["John", "x"])],

    # b. Apple & vegetables are food
    [("Food", True, ["Apple"])],
    [("Food", True, ["Vegetables"])],

    # c. Eats(x,y) ∧ ¬Killed(x) → Food(y)
    [("Eats", False, ["x","y"]), ("Killed", True, ["x"]), ("Food", True, ["y"])],

    # d. Anil eats peanuts; Anil alive
    [("Eats", True, ["Anil","Peanuts"])],
    [("Alive", True, ["Anil"])],

    # e. Harry eats everything Anil eats
    [("Eats", False, ["Anil","z"]), ("Eats", True, ["Harry","z"])],

    # f. Alive(x) → ¬Killed(x)
    [("Alive", False, ["x"]), ("Killed", False, ["x"])],

    # g. ¬Killed(x) → Alive(x)
    [("Killed", True, ["x"]), ("Alive", True, ["x"])]
]

# Query: h. John likes peanuts
query = [("Likes", False, ["John", "Peanuts"])]

def resolution(KB, query):
    clauses = KB + [query]  # add negated query
    print("\n--- Resolution Steps ---\n")

    while True:
        new = []
        for i in range(len(clauses)):
            for j in range(i+1, len(clauses)):
                resolvents = resolve(clauses[i], clauses[j])
                for r in resolvents:
                    print("Resolvent:", r)
                if [] in resolvents:
                    print("\nDerived empty clause → PROVED!\n")
                    return True
                new.extend(resolvents)

        if all(c in clauses for c in new):
            return False

        clauses.extend(new)


# ---------- Run ----------
result = resolution(KB, query)
print("Final Result:", result)
```

```
Resolvent: [('Alive', True, ['x']), ('Killed', True, ['x'])]
Resolvent: [('Killed', True, ['x']), ('Alive', True, ['x'])]
Resolvent: [('Killed', True, ['y']), ('Likes', True, ['John', 'y']), ('Eats', False, ['y', 'y'])]
Resolvent: [('Killed', True, ['x']), ('Eats', False, ['x', 'y']), ('Food', True, ['y'])]
Resolvent: [('Killed', True, ['Anil']), ('Food', True, ['Peanuts'])]
Resolvent: [('Alive', True, ['Anil'])]
Resolvent: [('Killed', True, ['Harry']), ('Eats', False, ['Anil', 'y']), ('Food', True, ['y'])]
Resolvent: [('Alive', True, ['x']), ('Killed', True, ['x'])]
Resolvent: [('Killed', True, ['x']), ('Alive', True, ['x'])]
Resolvent: [('Killed', True, ['y']), ('Likes', True, ['John', 'y']), ('Eats', False, ['y', 'y'])]
Resolvent: [('Killed', True, ['Anil']), ('Food', True, ['Peanuts'])]
Resolvent: [('Killed', True, ['Harry']), ('Food', True, ['z']), ('Eats', False, ['Anil', 'z'])]
Resolvent: [('Alive', True, ['x']), ('Alive', False, ['x'])]
Resolvent: [('Killed', True, ['x']), ('Killed', False, ['x'])]
Resolvent: [('Alive', True, ['x']), ('Alive', False, ['x'])]
Resolvent: [('Killed', True, ['x']), ('Killed', False, ['x'])]
Resolvent: [('Killed', True, ['x']), ('Eats', False, ['x', 'Peanuts'])]
Resolvent: [('Killed', True, ['Harry']), ('Food', True, ['Peanuts'])]
Resolvent: [('Killed', True, ['x']), ('Eats', False, ['x', 'y']), ('Food', True, ['y'])]
Resolvent: [('Alive', True, ['Anil'])]
Resolvent: [('Eats', False, ['Peanuts', 'Peanuts']), ('Killed', True, ['Peanuts'])]
Resolvent: [('Eats', False, ['Peanuts', 'Peanuts']), ('Killed', True, ['Peanuts'])]
Resolvent: [('Killed', True, ['Anil'])]
Resolvent: [('Killed', True, ['Harry']), ('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Eats', False, ['Peanuts', 'Peanuts']), ('Alive', False, ['Peanuts'])]
Resolvent: [('Alive', False, ['Peanuts']), ('Eats', False, ['Peanuts', 'Peanuts'])]
Resolvent: [('Eats', False, ['Peanuts', 'Peanuts']), ('Killed', True, ['Peanuts'])]
Resolvent: [('Likes', True, ['John', 'Anil']), ('Eats', False, ['Anil', 'Anil'])]
Resolvent: [('Likes', True, ['John', 'x']), ('Eats', False, ['x', 'x']), ('Killed', True, ['x'])]
Resolvent: [('Likes', True, ['John', 'Anil']), ('Eats', False, ['Anil', 'Anil'])]
Resolvent: [('Likes', True, ['John', 'x']), ('Eats', False, ['x', 'x']), ('Killed', True, ['x'])]
Resolvent: [('Likes', True, ['John', 'x']), ('Eats', False, ['x', 'x']), ('Alive', False, ['x'])]
Resolvent: [('Likes', True, ['John', 'x']), ('Eats', False, ['x', 'x']), ('Alive', False, ['x'])]
Resolvent: [('Likes', True, ['John', 'Anil']), ('Eats', False, ['Anil', 'Anil'])]
Resolvent: [('Killed', True, ['Anil'])]
Resolvent: [('Killed', True, ['Harry']), ('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Eats', False, ['x', 'Peanuts']), ('Alive', False, ['x'])]
Resolvent: [('Killed', True, ['Anil'])]
Resolvent: [('Killed', True, ['Harry']), ('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Eats', False, ['x', 'Peanuts']), ('Alive', False, ['x'])]
Resolvent: [('Killed', True, ['Harry'])]
Resolvent: [('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Eats', False, ['x', 'Peanuts']), ('Killed', True, ['x'])]
Resolvent: [('Killed', True, ['Harry'])]
Resolvent: [('Alive', False, ['Anil'])]
Resolvent: [('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Eats', False, ['Anil', 'Peanuts']), ('Alive', False, ['Harry'])]
Resolvent: [('Alive', False, ['Anil'])]
Resolvent: [('Alive', False, ['Harry']), ('Eats', False, ['Anil', 'Peanuts'])]
Resolvent: [('Killed', True, ['x']), ('Eats', False, ['x', 'Peanuts'])]
Resolvent: []

Derived empty clause → PROVED!

Final Result: True
```

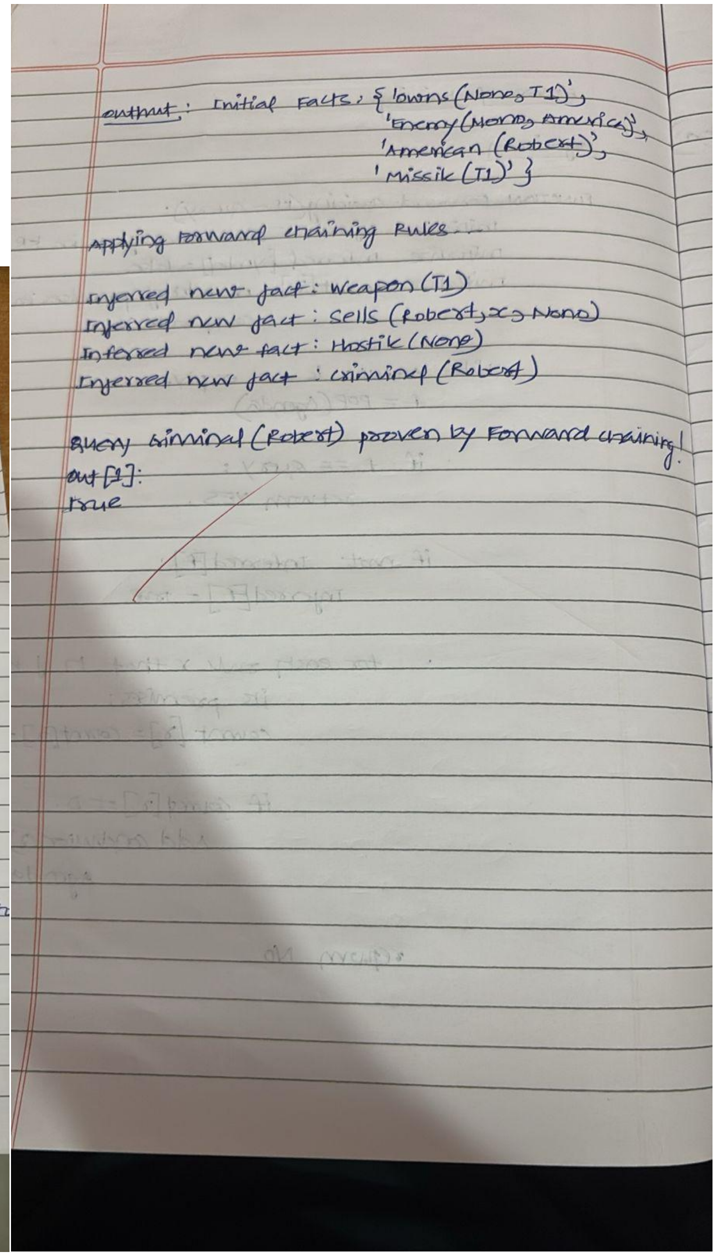
Program 11:

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

Algorithm:



```
Forward chaining

Algorithm

FUNCTION Forward-chaining(KB, query):
    Initialize Agenda = all known facts in KB
    Initialize Inferred[symbol] = false.
    Initialize count[r] = number of premises in
                          each rule r

    while AGENDA is not empty:
        f = POP(Agenda)

        if f == query:
            return YES.

        if not Inferred[f]:
            Inferred[f] = true

            for each rule r that has f in
                its premises:
                count[r] = count[r] - 1

                if count[r] == 0:
                    Add conclusion(r) to
                        Agenda

    return No
```



```
output: Initial Facts: {'owns(Nono, T1)',
                        'Enemy(Nono, America)',
                        'American(Robert)',
                        'Missile(T1)'}

Applying forward chaining Rules...

Inferred new fact: Weapon(T1)
Inferred new fact: Sells(Robert, x, Nono)
Inferred new fact: Hostile(Nono)
Inferred new fact: criminal(Robert)

Query criminal(Robert) proven by Forward chaining!
out[1]:
True
```

Code:

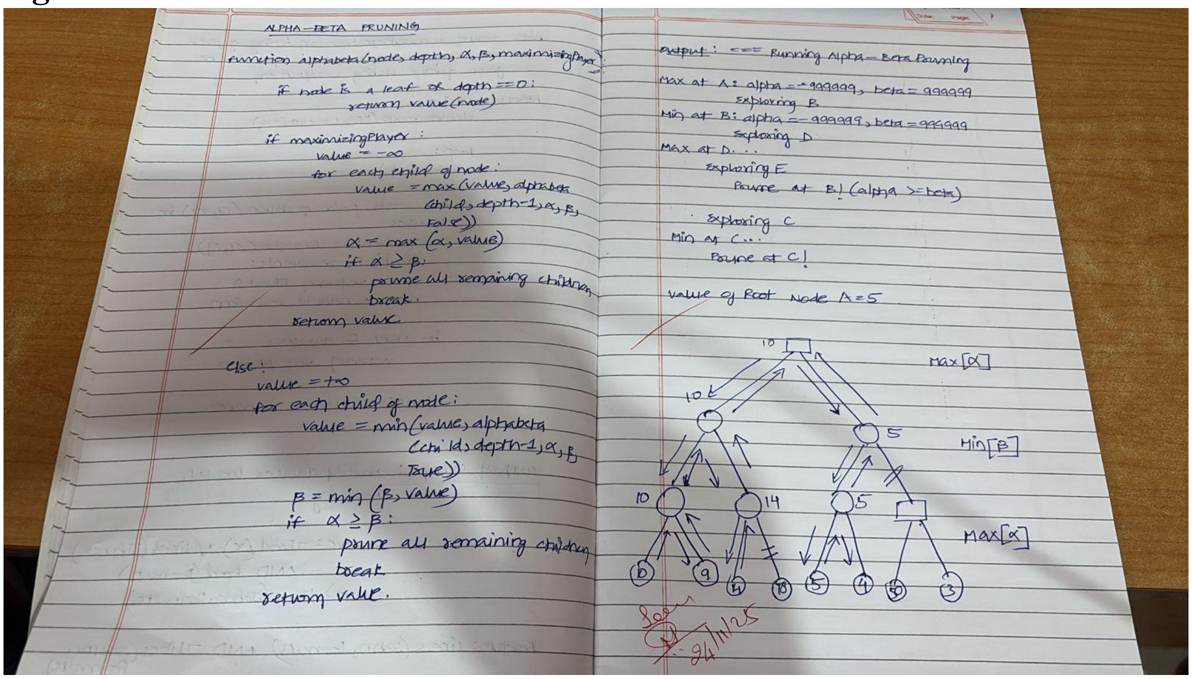
In [1]:
```python
from collections import defaultdict

facts = {
    "American(Robert)",
    "Enemy(Nono, America)",
    "Owns(Nono, T1)",
    "Missile(T1)"
}

rules = [
    ("Missile(x) => Weapon(x)"),
    ("Owns(Nono, x) ^ Missile(x) => Sells(Robert, x, Nono)"),
    ("Enemy(x, America) => Hostile(x)"),
    ("American(p) ^ Weapon(q) ^ Sells(p, q, r) ^ Hostile(r) => Criminal(p)")
]

query = "Criminal(Robert)"



def substitute(expr, var, const):
    return expr.replace(f"({var})", f"({const})").replace(f",{var}", f",{const}")

def forward_chaining(facts, rules, query):
    inferred = set()
    while True:
        new_fact_added = False
        for rule in rules:
            lhs, rhs = rule.split("=>")
            premises = [p.strip() for p in lhs.split("^")]
            conclusion = rhs.strip()


            variable_bindings = {}

            for fact in facts:
                for premise in premises:
                    if "(" in premise and "(" in fact:
                        pname = premise.split("(")[0]
                        fname = fact.split("(")[0]
                        if pname == fname:
                            p_args = premise[premise.find("(")+1:-1].split(",")
                            f_args = fact[fact.find("(")+1:-1].split(",")
                            for pa, fa in zip(p_args, f_args):
                                if pa[0].islower():  # variable
                                    variable_bindings[pa.strip()] = fa.strip()


            new_fact = conclusion
            for var, val in variable_bindings.items():
                new_fact = substitute(new_fact, var, val)


            if all(any(pname.split("(")[0] == f.split("(")[0] for f in facts) for pname in premises):
                if new_fact not in facts:
                    facts.add(new_fact)
                    inferred.add(new_fact)
                    new_fact_added = True
                    print(f"Inferred new fact: {new_fact}")

                    if new_fact == query:
                        print(f"\n✅ Query {query} proven by Forward Chaining!")
                        return True

        if not new_fact_added:
            break

    print(f"\n❌ Query {query} could not be proven.")
    return False



print("Initial Facts:", facts)
print("\nApplying Forward Chaining Rules...\n")
forward_chaining(facts, rules, query)
```

```
Initial Facts: {'Owns(Nono, T1)', 'Enemy(Nono, America)', 'American(Robert)', 'Missile(T1)'}

Applying Forward Chaining Rules...

Inferred new fact: Weapon(T1)
Inferred new fact: Sells(Robert, x, Nono)
Inferred new fact: Hostile(Nono)
Inferred new fact: Criminal(Robert)

✅ Query Criminal(Robert) proven by Forward Chaining!
```

Out[1]: True

Program 12:

Implement Alpha-Beta Pruning.
Algorithm:



ALPHA-BETA PRUNING

function alphabeta(node, depth, α, β, maximizingPlayer):
if node is a leaf or depth == 0:
return value(node)

if maximizingPlayer:
value = -∞
for each child of node:
value = max(value, alphabeta(child, depth-1, α, β, False))
α = max(α, value)
if α ≥ β:
prune all remaining children
break.
return value.

else:
value = +∞
for each child of node:
value = min(value, alphabeta(child, depth-1, α, β, True))
β = min(β, value)
if α ≥ β:
prune all remaining children
break.
return value.

output: === Running Alpha-Beta Pruning
Max at A: alpha = -999999, beta = 999999
Exploring B
Min at B: alpha = -999999, beta = 999999
Exploring D
Max at D...
Exploring E
Prune at B! (alpha >= beta)
Exploring C
Min at C...
Prune at C!
Value of Root Node A = 5

Code:

```
import math
PLAYER = "X"  # Human
AI = "O"      # Computer def
print_board(board):     for
row in board:         print(" |
".join(row))        print("-" *
9) def
available_moves(board):
    """Return list of available (row, col) moves."""
moves = []
    for i in range(3):          for j
in range(3):             if
board[i][j] == " ":
moves.append((i, j))     return
moves def
check_winner(board):
```

```
    """Return 'X' if X wins, 'O' if O wins, or None otherwise."""
for i in range(3):         if board[i][0] == board[i][1] ==
board[i][2] != " ":
            return board[i][0]         if board[0][i] ==
board[1][i] == board[2][i] != " ":
            return board[0][i]     if board[0][0] ==
board[1][1] == board[2][2] != " ":
        return board[0][0]     if board[0][2] ==
board[1][1] == board[2][0] != " ":
        return board[0][2]
return None def
is_full(board):
    return all(cell != " " for row in board for cell in row) def
minimax(board, depth, is_maximizing):
    winner = check_winner(board)
if winner == AI:
        return 1     elif winner
== PLAYER:
        return -1     elif
is_full(board):
        return 0     if
is_maximizing:
        best_score = -math.inf         for (i, j)
in available_moves(board):
            board[i][j] = AI             score =
minimax(board, depth + 1, False)
board[i][j] = " "             best_score = max(score,
best_score)         return best_score     else:
        best_score = math.inf         for (i, j)
in available_moves(board):
            board[i][j] = PLAYER             score =
minimax(board, depth + 1, True)
board[i][j] = " "             best_score = min(score,
best_score)         return best_score def
best_move(board):
    """Find the best move for the AI."""
best_score = -math.inf     move =
None
    for (i, j) in available_moves(board):
board[i][j] = AI         score =
minimax(board, 0, False)
        board[i][j] = " "
if score > best_score:
best_score = score
```

```
            move = (i, j)
return move def
play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
print("Tic Tac Toe - You are X, AI is O")
print_board(board)     while True:
        row = int(input("Enter row (0-2): "))
col = int(input("Enter col (0-2): "))         if
board[row][col] != " ":
            print("Cell taken, try again.")
continue
        board[row][col] = PLAYER         if
check_winner(board) == PLAYER:
            print_board(board)
print("You win!")             break
elif is_full(board):
print_board(board)
print("It's a draw!")             break
print("AI is making a move...")
move = best_move(board)         if
move:
            board[move[0]][move[1]] = AI
print_board(board)         if
check_winner(board) == AI:
            print("AI wins!")
break         elif
is_full(board):
print("It's a draw!")
            break if __name__
== "__main__":
    play_game()
```

ScreenShot:

Output

```
  |   |
---------
  |   |
---------
  |   |
---------
Enter row (0-2): 0
Enter col (0-2): 0
AI is making a move...
X |   |
---------
  | O |
---------
  |   |
---------
Enter row (0-2): 1
Enter col (0-2): 2
AI is making a move...
X | O |
---------
  | O | X
---------
  |   |
---------
Enter row (0-2): 0
Enter col (0-2): 2
AI is making a move...
X | O | X
---------
  | O | X
---------
  | O |
---------
AI wins!
```