Artificial Intelligence Lab Report

Submitted by

Yash Kumar Sinha
(1BM22CS334) Batch: C3

Course:Artificial Intelligence
Course Code: 23CS5PCAIN
Sem & Section: 5F

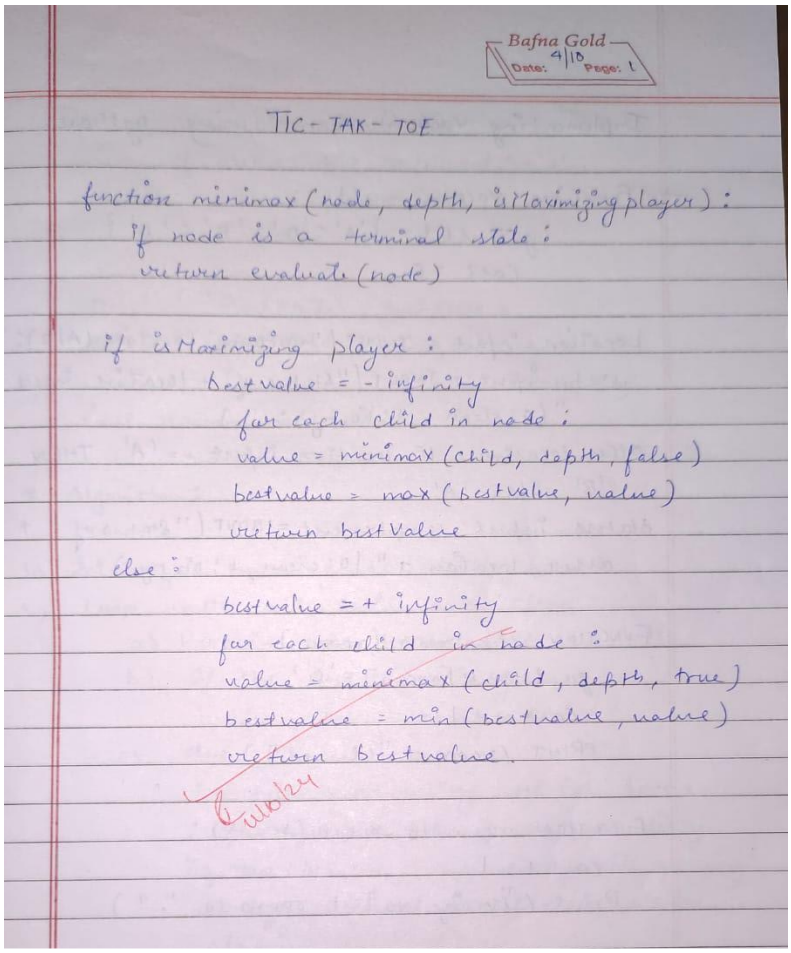
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCEAND ENGINEERING

B. M. S. COLLEGE OFENGINEERING
(Autonomous Institution underVTU)
BENGALURU-560019
2022-2023

Tableofcontents

Program 1-Tic Tac toe

Algorithm



Code

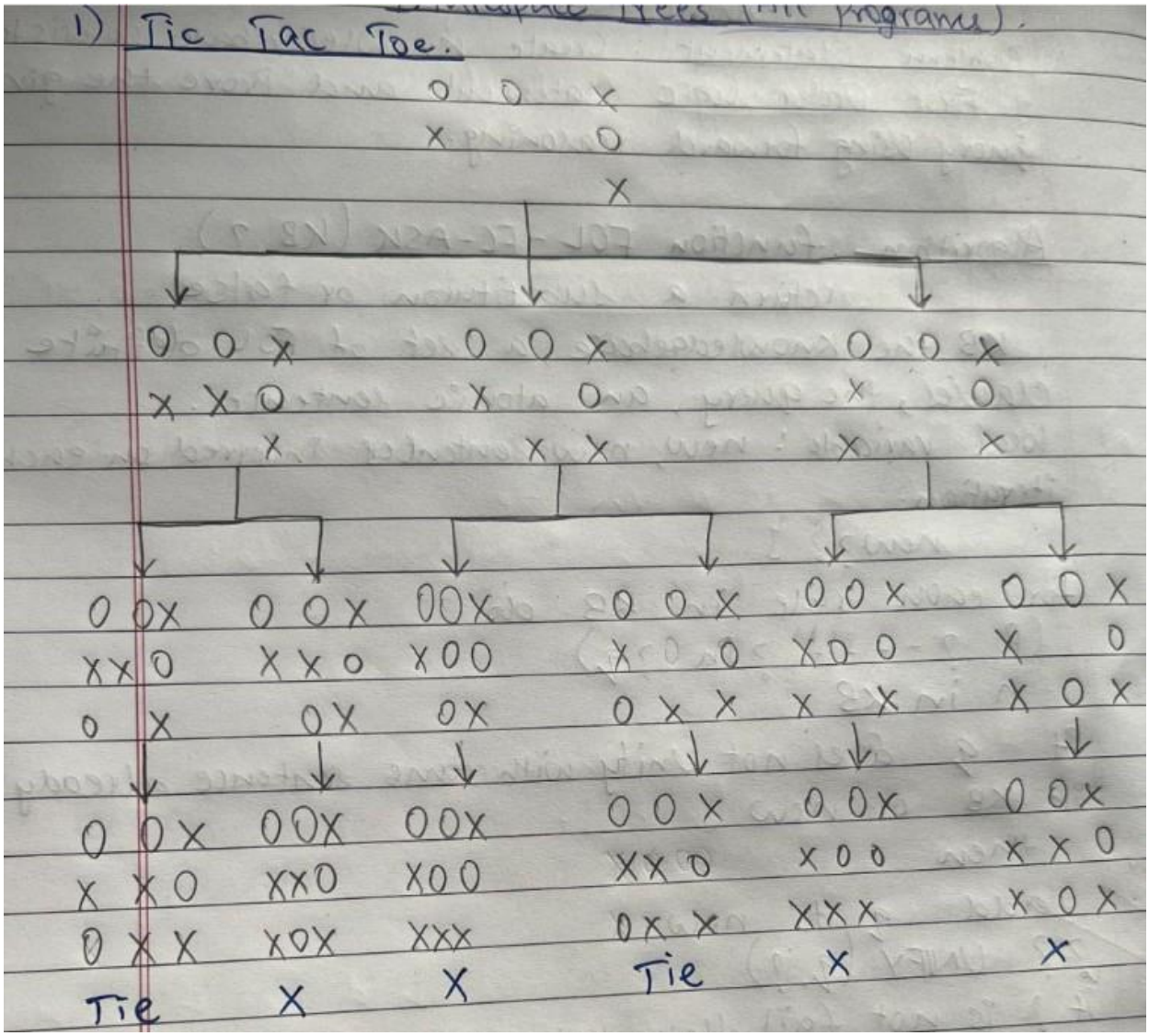
```python
def print_board(board):
    """Prints the current state of the board."""
    for row in board:
        print("|".join(row))
        print("-" * 5)


def check_winner(board):
    """Checks for a winner or a draw."""
    # Check rows and columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] and board[i][0] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] and board[0][i] != " ":
            return board[0][i]

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return board[0][0]
```

```
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return board[0][2]

    # Check for draw
    for row in board:
        if " " in row:
            return None  # Game is still ongoing

    return "Draw"  # All cells are filled and no winner


def tic_tac_toe():
    """Main function to play the Tic Tac Toe game."""
    # Initialize an empty board
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    current_player = "X"
    while True:
        print(f"Player {current_player}'s turn.")
        try:
            # Ask for the row and column
            row = int(input("Enter the row (0, 1, 2): "))
            col = int(input("Enter the column (0, 1, 2): "))

            # Check if the cell is empty
            if row < 0 or row > 2 or col < 0 or col > 2:
                print("Invalid input! Row and column must be between 0 and 2.")
                continue
            if board[row][col] != " ":
                print("Cell already occupied! Choose another cell.")
                continue

            # Make the move
            board[row][col] = current_player
            print_board(board)

            # Check for a winner
            result = check_winner(board)
            if result:
                if result == "Draw":
                    print("It's a draw!")
                else:
                    print(f"Player {result} wins!")
                break

            # Switch player
            current_player = "O" if current_player == "X" else "X"
        except ValueError:
            print("Invalid input! Please enter numbers between 0 and 2.")
        except Exception as e:
            print(f"An error occurred: {e}")


# Run the game
tic_tac_toe()
```

Output Snapshot

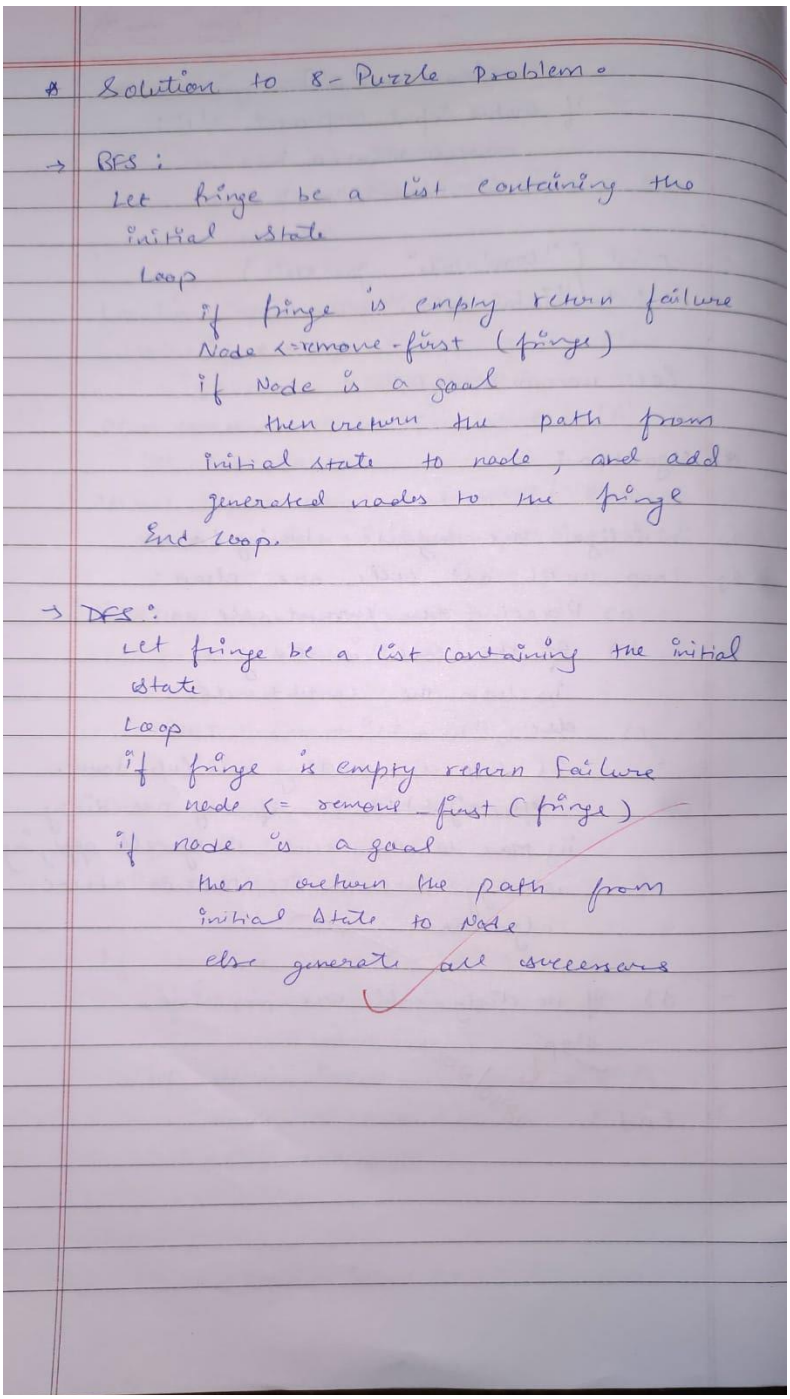
```
YashKumarSinha_1BM22CS334
Welcome to Tic Tac Toe!
 | |
-----
 | |
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X| |
-----
 | |
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 1
X| |
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 2
X| |X
-----
 |O|
-----
 | |
-----
Player O's turn.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 | |
-----
Player X's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 2
X|O|X
-----
 |O|
-----
 | |X
-----
Player O's turn.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X|O|X
-----
 |O|
-----
 |O|X
-----
Player O wins!
```

State Space Tree



Program 2/3-8 Puzzle (DFS BFS)

Algorithm



* Solution to 8-Puzzle Problem.

→ BFS:
Let fringe be a list containing the initial state
Loop
if fringe is empty return failure
Node <=remove-first (fringe)
if Node is a goal
then return the path from initial state to node, and add generated nodes to the fringe
End Loop.

→ ~~DFS~~:
Let fringe be a list containing the initial state
Loop
if fringe is empty return Failure
node <= remove-first (fringe)
if node is a goal
then return the path from initial state to node
else generate all successors

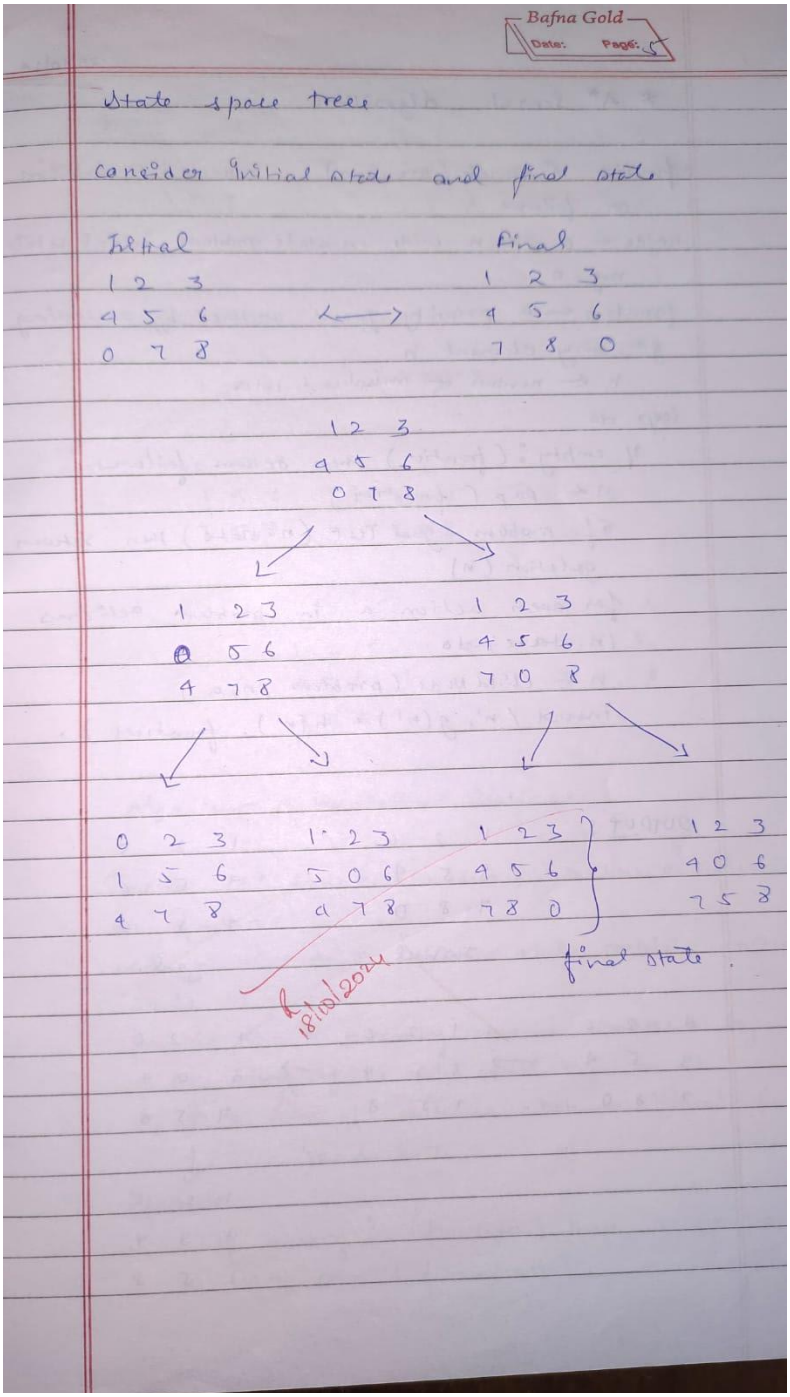
Code

```python
#8 Puzzle program Solution

def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()

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

def is_goal_state(board):
    return board == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def get_neighbors(board):
    x, y = find_empty_tile(board)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_board = [row[:] for row in board]
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
            neighbors.append(new_board)
    return neighbors

def bfs(initial_board):
    queue = [(initial_board, 0, [])]
    visited = set()

    while queue:
        current_board, moves, path = queue.pop(0)
        if is_goal_state(current_board):
            return moves, path + [current_board]
        visited.add(tuple(map(tuple, current_board)))

        for neighbor in get_neighbors(current_board):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, moves + 1, path + [current_board]))

    return -1, []

# Function to get user input
def get_user_input():
    print("Enter the 8-puzzle configuration (use 0 for the empty space):")
    user_input = input("Example: 1 2 3 4 0 6 7 5 8\n")
    tiles = list(map(int, user_input.split()))

    if len(tiles) != 9:
        raise ValueError("Please enter exactly 9 numbers.")

    return [tiles[i:i + 3] for i in range(0, 9, 3)]

# Example usage
if __name__ == "__main__":
    try:
        initial_board = get_user_input()
        print("Initial Board:")
        print_board(initial_board)
        moves, solution_path = bfs(initial_board)

        if moves != -1:
            print(f"Solved in {moves} moves.\nSolution path:")
            for step in solution_path:
                print_board(step)
        else:
            print("No solution found.")
    except ValueError as e:
        print(e)
```

Output Snapshot

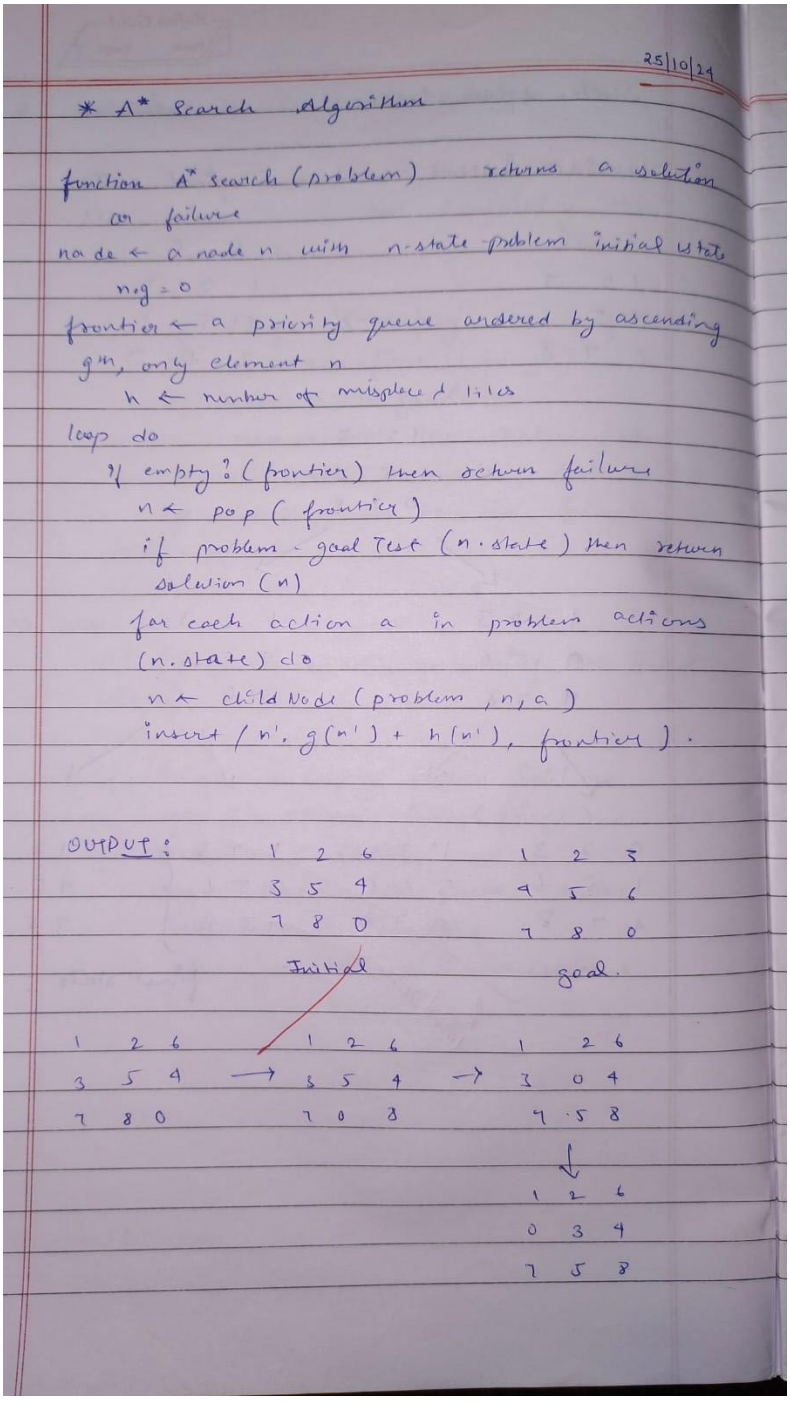
```
YashKumarSinha_1BM22CS334
Enter the 8-puzzle configuration (use 0 for the empty space):
Example: 1 2 3 4 0 6 7 5 8
1 2 3 4 0 6 7 5 8
Initial Board:
1 2 3
4 0 6
7 5 8

Solved in 2 moves.
Solution path:
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
```

State Space Tree



Program 04 -8 Puzzle Using A*

Algorithm



25/10/24

* A* Search Algorithm

function A* search (problem) returns a solution or failure

node ← a node n with n.state = problem initial state
n.g = 0
frontier ← a priority queue ordered by ascending g(n), only element n
h ← number of misplaced tiles

loop do
if empty? (frontier) then return failure
n ← pop (frontier)
if problem.goalTest (n.state) then return solution (n)
for each action a in problem.actions (n.state) do
n' ← childNode (problem, n, a)
insert (n', g(n') + h(n'), frontier).

OUTPUT :

1 2 6
3 5 4
7 8 0
Initial

1 2 3
4 5 6
7 8 0
goal.

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

1 2 6
0 3 4
7 5 8

Code

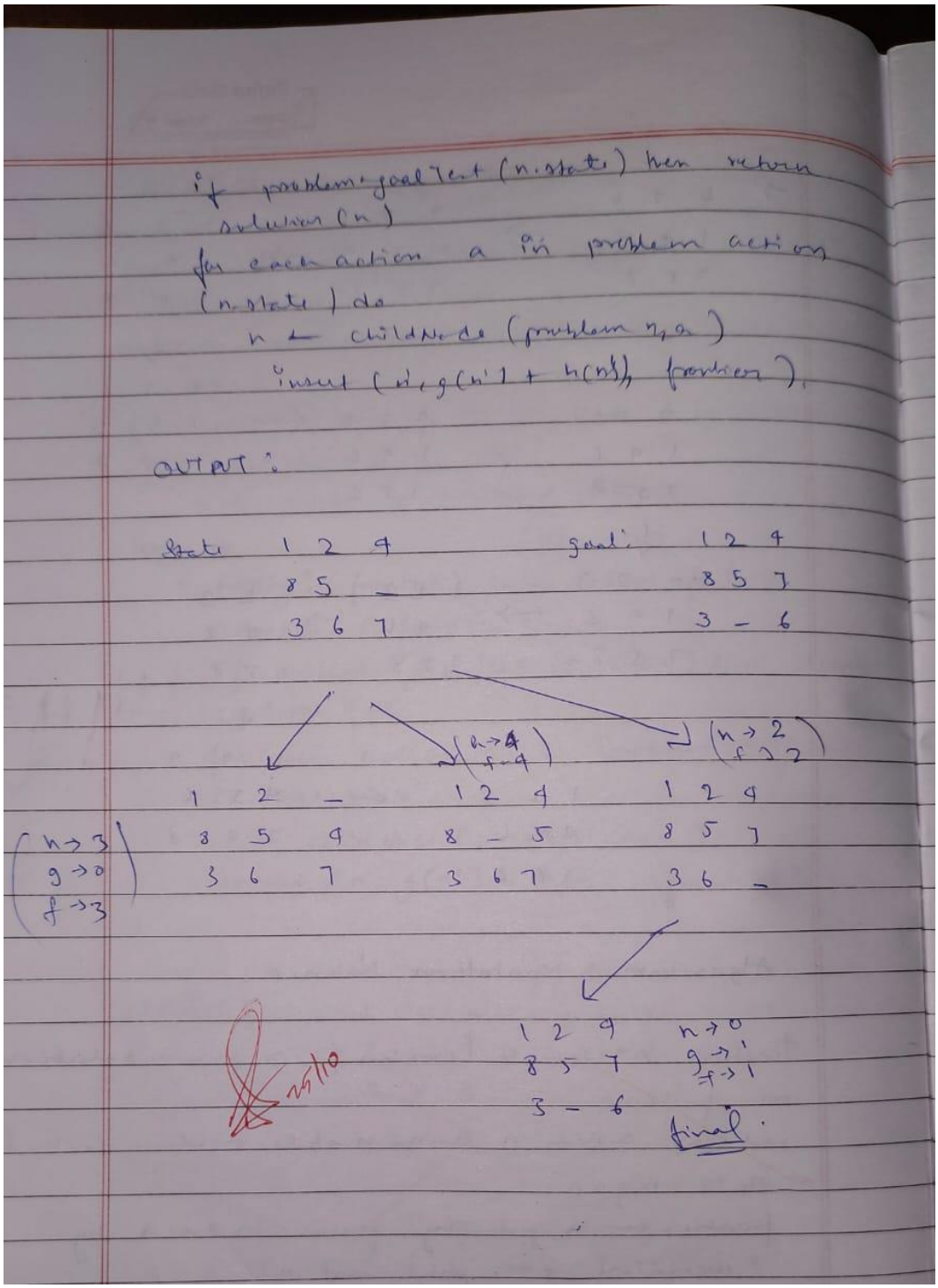
```
import heapq

class PuzzleState:
    def __init__(self, board, empty_tile, moves=0, previous=None):
        self.board = board
        self.empty_tile = empty_tile  # (row, col) of the empty tile
        self.moves = moves
        self.previous = previous  # to trace the path back

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

    def f(self):
        # Total cost function (g + h)
        return self.moves + self.heuristic()

    def heuristic(self):
        # Using Manhattan distance as the heuristic
        total_distance = 0
        for i in range(3):
            for j in range(3):
                if self.board[i][j] != 0:  # Skip empty tile
                    # Calculate the target position of the tile
                    target_x = (self.board[i][j] - 1) // 3
                    target_y = (self.board[i][j] - 1) % 3
                    # Calculate Manhattan distance for each tile
                    distance = abs(target_x - i) + abs(target_y - j)
                    total_distance += distance
        return total_distance

    def get_neighbors(self):
        neighbors = []
        row, col = self.empty_tile
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Down, Up, Right, Left

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:  # Within bounds
                new_board = [list(row) for row in self.board]
                # Swap the empty tile with the adjacent tile
                new_board[row][col], new_board[new_row][new_col] = new_board[new_row][new_col], new_board[row][col]
                # Create a new PuzzleState for the neighbor
                neighbors.append(PuzzleState(new_board, (new_row, new_col), self.moves + 1, self))

        return neighbors

def a_star(start_board):
    # Find the position of the empty tile
    start_tile = next((i, j) for i in range(3) for j in range(3) if start_board[i][j] == 0)
    start_state = PuzzleState(start_board, start_tile)
    # Define the goal state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    open_set = []
    closed_set = set()
    heapq.heappush(open_set, start_state)

    while open_set:
        current_state = heapq.heappop(open_set)

        if current_state.board == goal_state:
            path = []
            while current_state:
                path.append(current_state.board)
                current_state = current_state.previous
            return path[::-1]  # Return reversed path

        closed_set.add(tuple(map(tuple, current_state.board)))

        for neighbor in current_state.get_neighbors():
            if tuple(map(tuple, neighbor.board)) in closed_set:
                continue
```

```
            heapq.heappush(open_set, neighbor)

    return None  # No solution found

def get_user_input():
    print("Enter the 3x3 puzzle board (use 0 for the empty tile):")
    board = []
    for i in range(3):
        row = input(f"Row {i + 1} (space-separated): ").strip().split()
        if len(row) != 3 or any(not num.isdigit() or int(num) < 0 or int(num) > 8 for num in row):
            print("Invalid input. Please enter numbers between 0 and 8.")
            return None
        board.append(list(map(int, row)))

    if set(num for row in board for num in row) != set(range(9)):
        print("Invalid input. The board must contain numbers 0 through 8 exactly once.")
        return None

    return board

# Example usage:
start_board = get_user_input()
if start_board is not None:
    solution = a_star(start_board)
    if solution:
        for step in solution:
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")
```

Output Snapshot

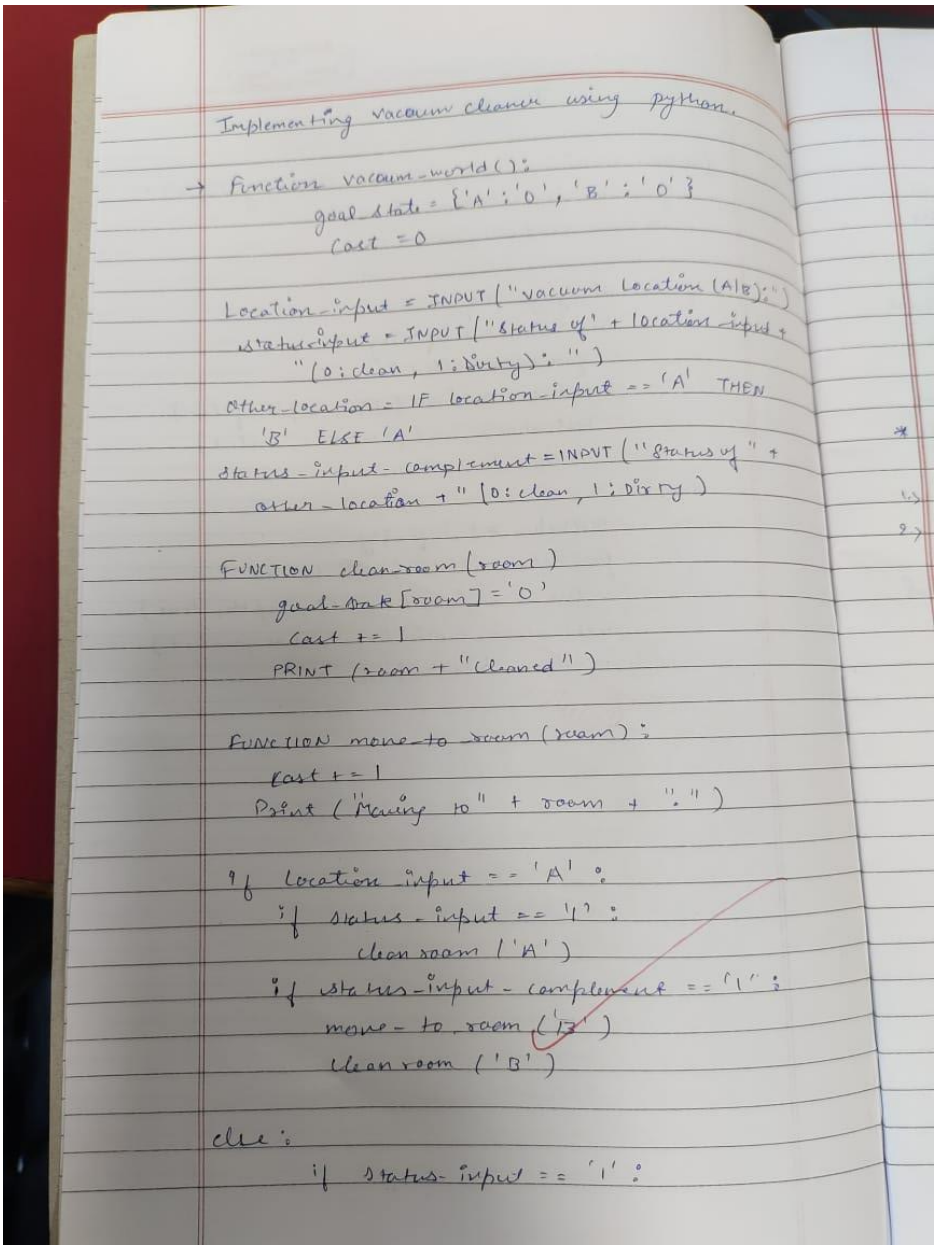
```
YashKumarSinha_1BM22CS334
Enter the 3x3 puzzle board (use 0 for the empty tile):
Row 1 (space-separated): 1 2 3
Row 2 (space-separated): 4 0 6
Row 3 (space-separated): 7 5 8
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

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

State Space Tree



Program 5- Vacuum Cleaner

Algorithm



Implementing vacuum cleaner using python.

→ Function vacuum-world():
goal_state = {'A':'0', 'B':'0'}
cost = 0

Location_input = INPUT("vacuum Location (A|B):")
status_input = INPUT("Status of" + location_input + "(0:clean, 1:Dirty):")
Other_location = IF location_input == 'A' THEN 'B' ELSE 'A'
status_input_complement = INPUT("Status of" + other_location + "(0:clean, 1:Dirty)

FUNCTION clean_room(room)
goal_state[room] = '0'
cost += 1
PRINT(room + "cleaned")

FUNCTION move_to_room(room):
cost += 1
Print("Moving to" + room + ".")

if location_input == 'A':
if status_input == '1':
clean_room('A')
if status_input_complement == '1':
move_to_room('B')
clean_room('B')

else:
if status_input == '1':

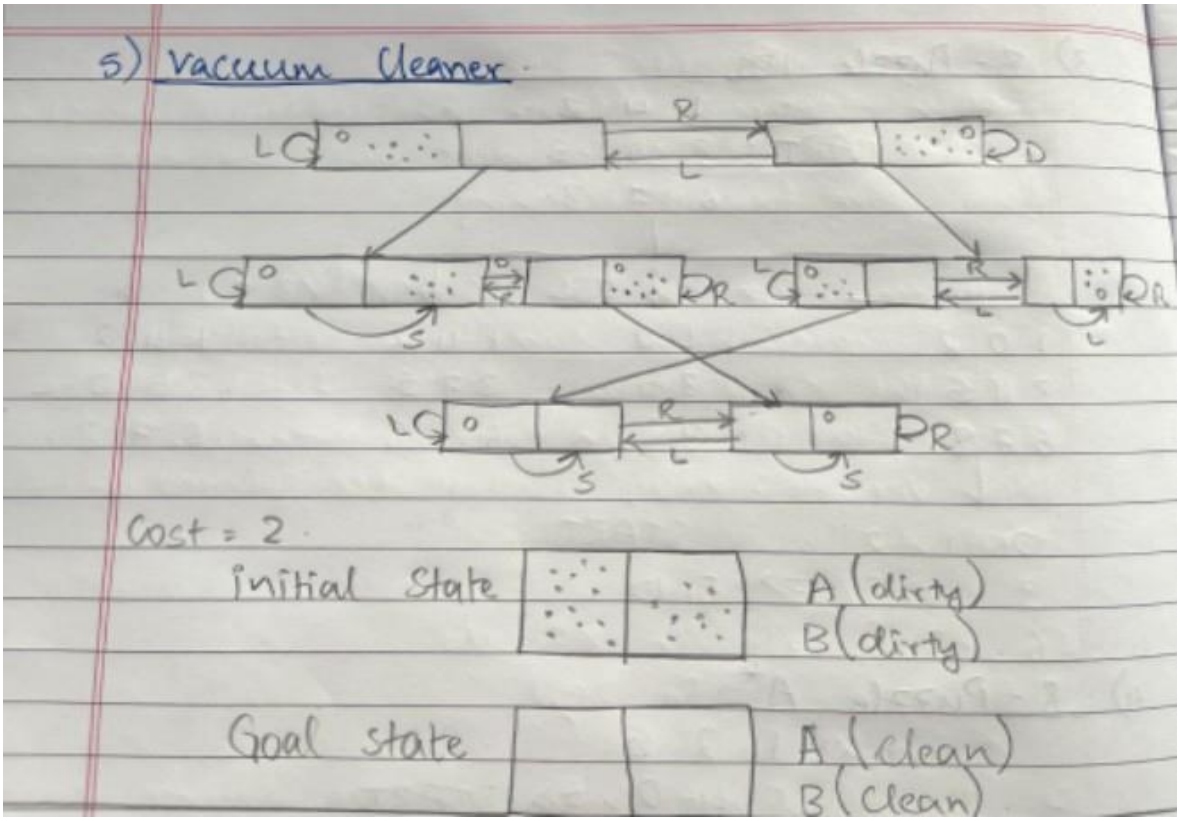
Code

```
# Vaccume cleaner Code
def vacuum_world():
    # Initialize goal state
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    # User inputs for the vacuum's location and status
    location_input = input("Enter Location of Vacuum (A/B): ").strip().upper()
    status_input = input(f"Enter status of {location_input} (0: Clean, 1: Dirty): ").strip()
    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input(f"Enter status of {other_location} (0: Clean, 1: Dirty): ").strip()

    print("Initial Location Condition:", goal_state)

    def clean_room(room):
        """Cleans the specified room and updates the cost."""
        nonlocal cost
        goal_state[room] = '0'
        cost += 1  # Cost for sucking dirt
        print(f"Location {room} has been cleaned.")

    def move_to_room(room):
        """Moves to the specified room and updates the cost."""
        nonlocal cost
        cost += 1  # Cost for moving
        print(f"Moving to Location {room}.")

    # Start cleaning based on the location input
    if location_input == 'A':
        if status_input == '1':
            print("Location A is Dirty.")
            clean_room('A')

        if status_input_complement == '1':
            print("Location B is Dirty.")
            move_to_room('B')
            clean_room('B')
        elif status_input == '0' and status_input_complement == '0':
            print("Both locations are clean.")

    elif location_input == 'B':
        if status_input == '1':
            print("Location B is Dirty.")
            clean_room('B')

        if status_input_complement == '1':
            print("Location A is Dirty.")
            move_to_room('A')
            clean_room('A')
        elif status_input == '0' and status_input_complement == '0':
            print("Both locations are clean.")

    # Output final state and cost
    print("GOAL STATE:", goal_state)
    print("Performance Measurement:", cost)

vacuum_world()
```

Output Snapshot

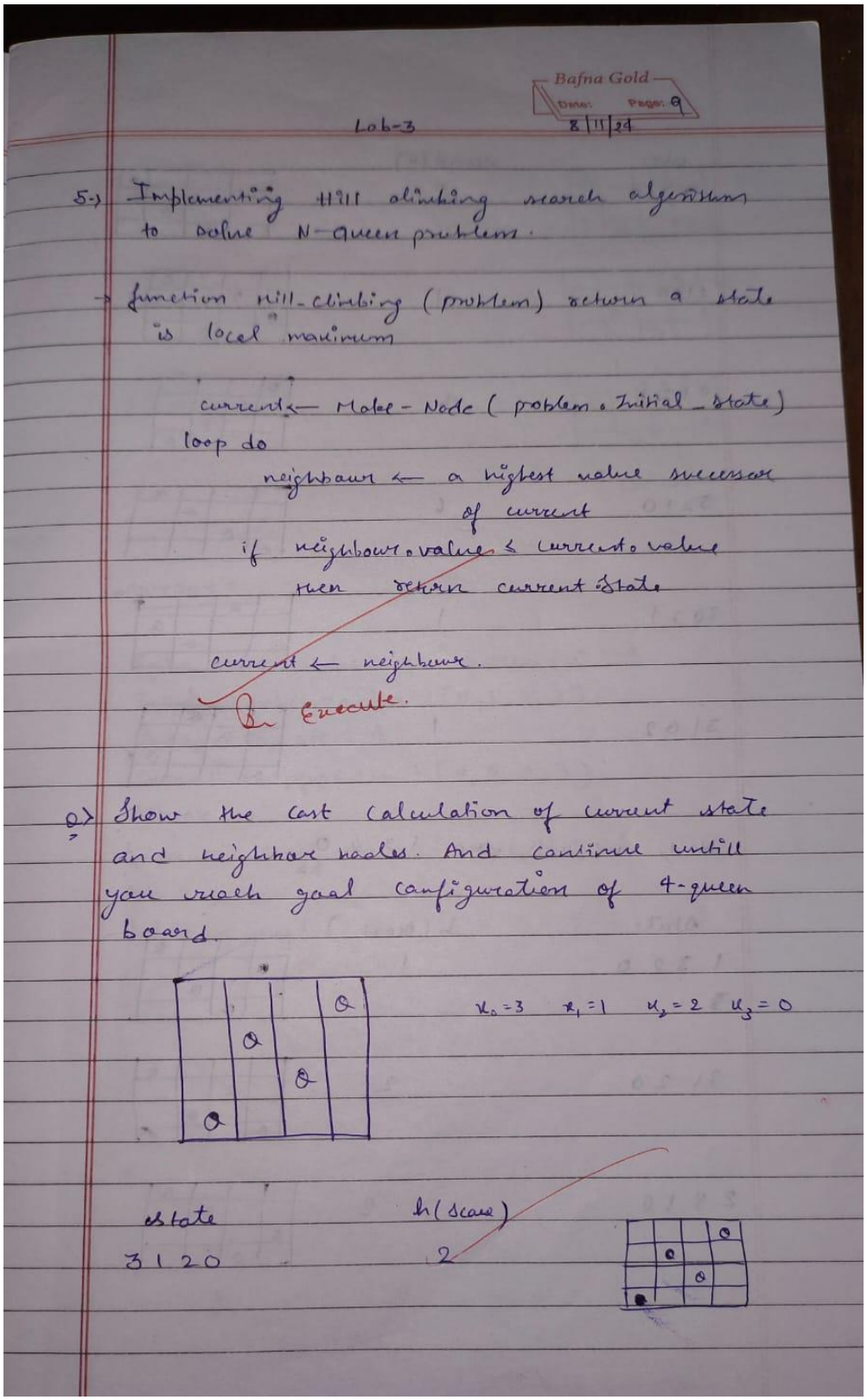
```
YashKumarSinha_1BM22CS334
Enter Location of Vacuum (A/B): A
Enter status of A (0: Clean, 1: Dirty): 0
Enter status of B (0: Clean, 1: Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Location B is Dirty.
Moving to Location B.
Location B has been cleaned.
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement: 2
```

State Space Tree



Program-06 Hill Climbling

Algorithm



Lab-3 8/11/24

5.) Implementing Hill climbing search algorithm to solve N-queen problem.

→ function Hill-climbing (problem) return a state is local maximum

current ← Make-Node (problem.Initial_State)
loop do
neighbour ← a highest value successor of current
if neighbour.value ≤ current.value
then return current.State
current ← neighbour.
Execute.

Q) Show the last calculation of current state and neighbor nodes. And continue until you reach goal configuration of 4-queen board.

$x_0 = 3 \quad x_1 = 1 \quad x_2 = 2 \quad x_3 = 0$

State h(Score)
3 1 2 0 2

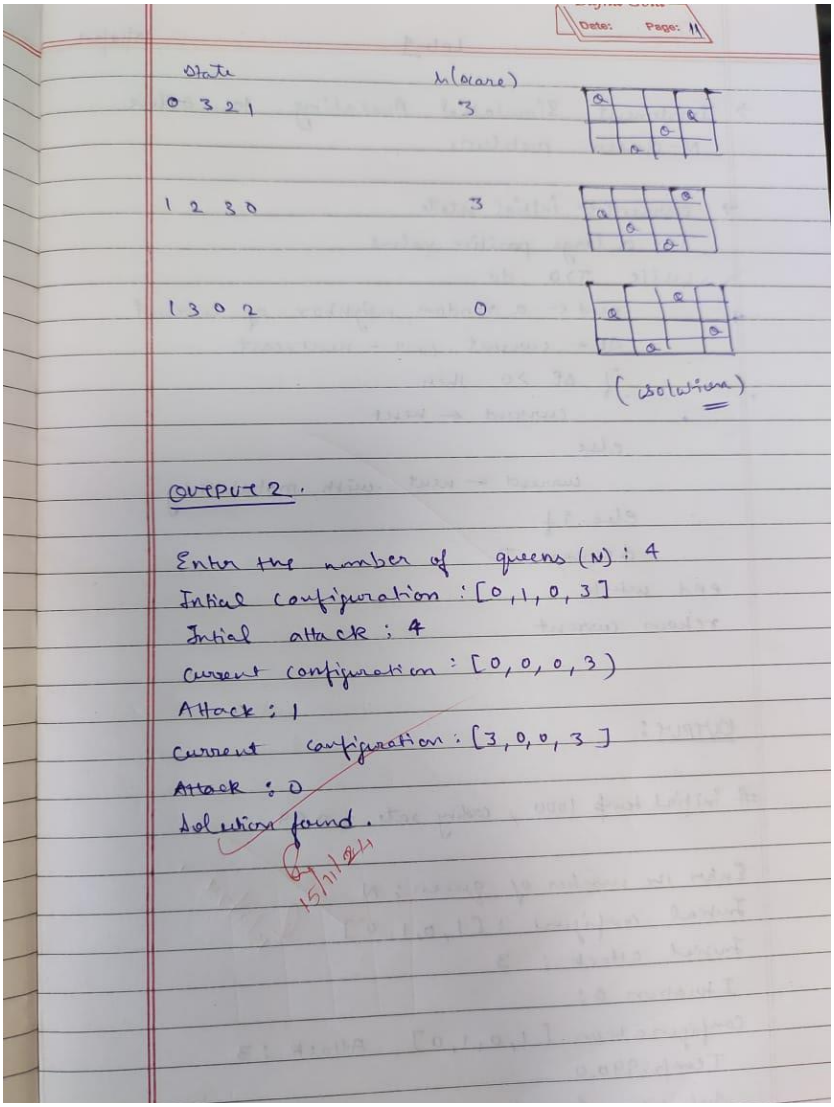
Code

```
mport random

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

def generate_random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]

def get_neighbors(board):
    neighbors = []
    n = len(board)
    for row in range(n):
        for col in range(n):
            if board[row] != col:
                new_board = board[:]
                new_board[row] = col
                neighbors.append(new_board)
    return neighbors

def hill_climbing(n):
    current = generate_random_state(n)
    while True:
        neighbors = get_neighbors(current)
        next_state = min(neighbors, key=lambda board: fitness(board))

        if fitness(next_state) == 0:
            return next_state

        if fitness(next_state) >= fitness(current):
            return None

        current = next_state

# Example usage:
n = 5  # Size of the board (8 queens problem)
solution = hill_climbing(n)
if solution:
    print("Solution found:", solution)
else:
    print("No solution found")
```

Output Snapshot

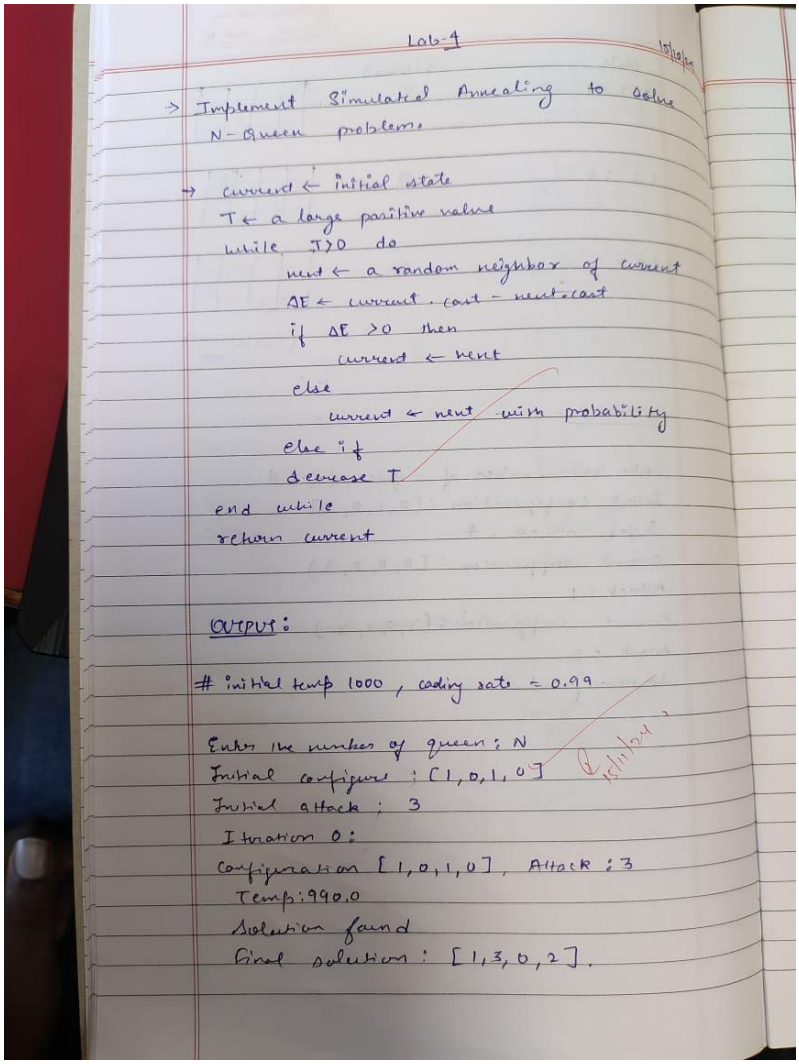
```
YashKumarSinha_1BM22CS334
Solution found: [4, 2, 0, 3, 1]
```

StateSpaceTree



Program 07 Simulated Annealing

Algorithm



Lab-7

→ Implement Simulated Annealing to solve N-Queen problems.

→ current ← initial state
T ← a large positive value
while T>0 do
next ← a random neighbor of current
ΔE ← current.cost − next.cost
if ΔE >0 then
current ← next
else
current ← next with probability
else if
decrease T
end while
return current

OUTPUT:

initial temp 1000, cooling rate = 0.99

Enter the number of queen: N
Initial configure : [1,0,1,0]
Initial attack : 3
Iteration 0:
Configuration [1,0,1,0], Attack :3
Temp:990.0
Solution found
Final solution : [1,3,0,2].

Code

```
#STIMULATED ANNEALING
import random
import math

def initialize_state():
    """Initialize the state with a random configuration of the problem."""
    # Replace with specific initialization for your problem, e.g., N-Queens board
    return [random.randint(0, n - 1) for _ in range(n)]

def cost_function(state):
    """Calculate the cost (or conflicts) of a given state."""
    # Replace with specific cost calculation, e.g., number of conflicts for N-Queens
    n = len(state)
    row_conflicts = [0] * n
    main_diag_conflicts = [0] * (2 * n - 1)
    anti_diag_conflicts = [0] * (2 * n - 1)

    conflicts = 0
    for row in range(n):
```

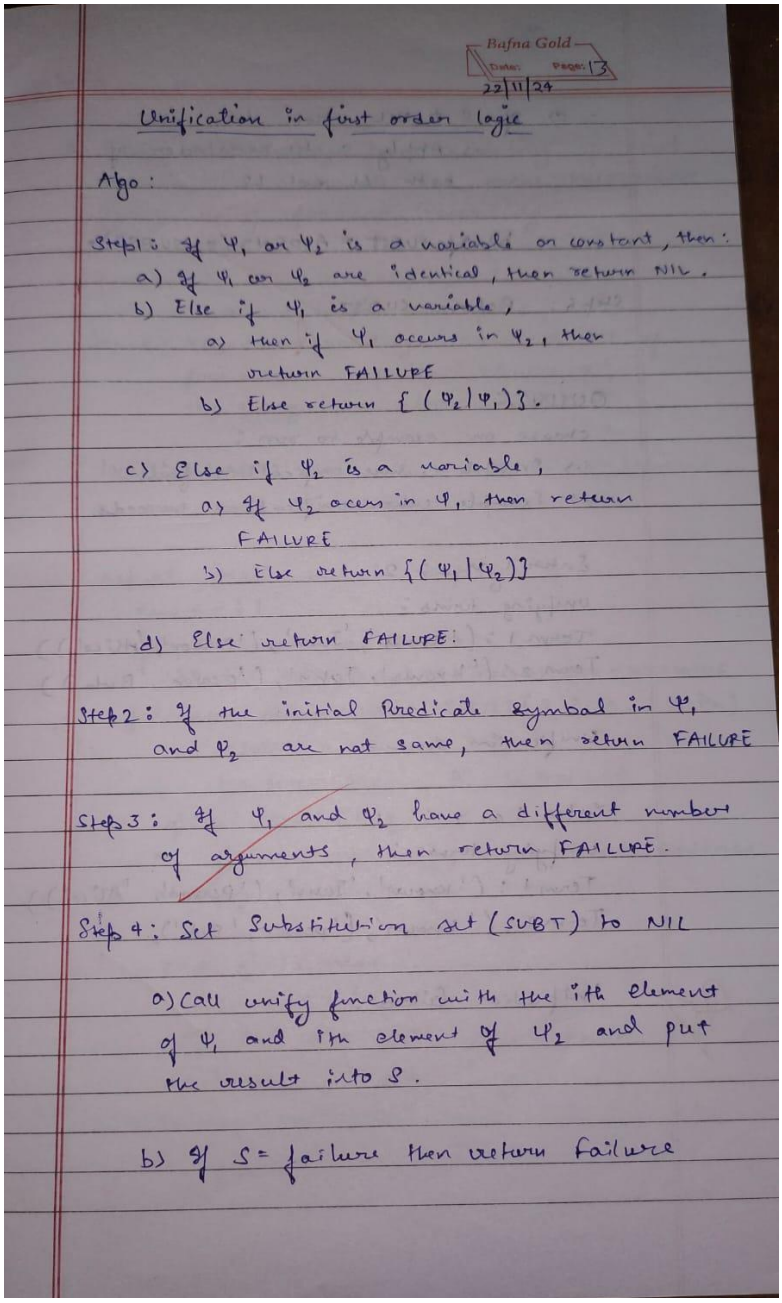
```
        col = state[row]
        row_conflicts[col] += 1
        main_diag_conflicts[row - col + n - 1] += 1
        anti_diag_conflicts[row + col] += 1

    for row in range(n):
        col = state[row]
        if row_conflicts[col] > 1:
            conflicts += row_conflicts[col] - 1
        if main_diag_conflicts[row - col + n - 1] > 1:
            conflicts += main_diag_conflicts[row - col + n - 1] - 1
        if anti_diag_conflicts[row + col] > 1:
            conflicts += anti_diag_conflicts[row + col] - 1

    return conflicts

def get_neighbor(state):
    """Get a random neighboring state by modifying the current state slightly."""
    new_state = state[:]
    row = random.randint(0, len(state) - 1)
    new_col = random.randint(0, len(state) - 1)
    while new_col == new_state[row]:
        new_col = random.randint(0, len(state) - 1)
    new_state[row] = new_col
    return new_state

def simulated_annealing(initial_temp=1000, cooling_rate=0.99, max_iterations=10000):
    """Perform Simulated Annealing to find a solution."""
    current = initialize_state()
    T = initial_temp

    for i in range(max_iterations):
        if T <= 0:
            break

        next_state = get_neighbor(current)
        delta_E = cost_function(current) - cost_function(next_state)

        if delta_E > 0:
            current = next_state
        else:
            # Accept the worse state with a certain probability
            if random.random() < math.exp(delta_E / T):
                current = next_state

        # Decrease the temperature
        T *= cooling_rate

    return current

# Example usage
n = 4  # For 4-Queens problem
solution = simulated_annealing()
print("Final solution:", solution)
print("Number of conflicts:", cost_function(solution))
```

OUTPUT

```
YashKumarSinha_1BM22CS334
Final solution: [2, 0, 3, 1]
Number of conflicts: 0
```

Program-08 Unification in FOL

Algorithm



Bafna Gold
Date: Page: 3
22/11/24

Unification in first order logic

Algo:

Step1: If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if Ψ_1 is a variable,
a) then if Ψ_1 occurs in Ψ_2, then return FAILURE
b) Else return $\{(\Psi_2 / \Psi_1)\}$.

c) Else if Ψ_2 is a variable,
a) If Ψ_2 occurs in Ψ_1, then return FAILURE
b) Else return $\{(\Psi_1 / \Psi_2)\}$

d) Else return FAILURE.

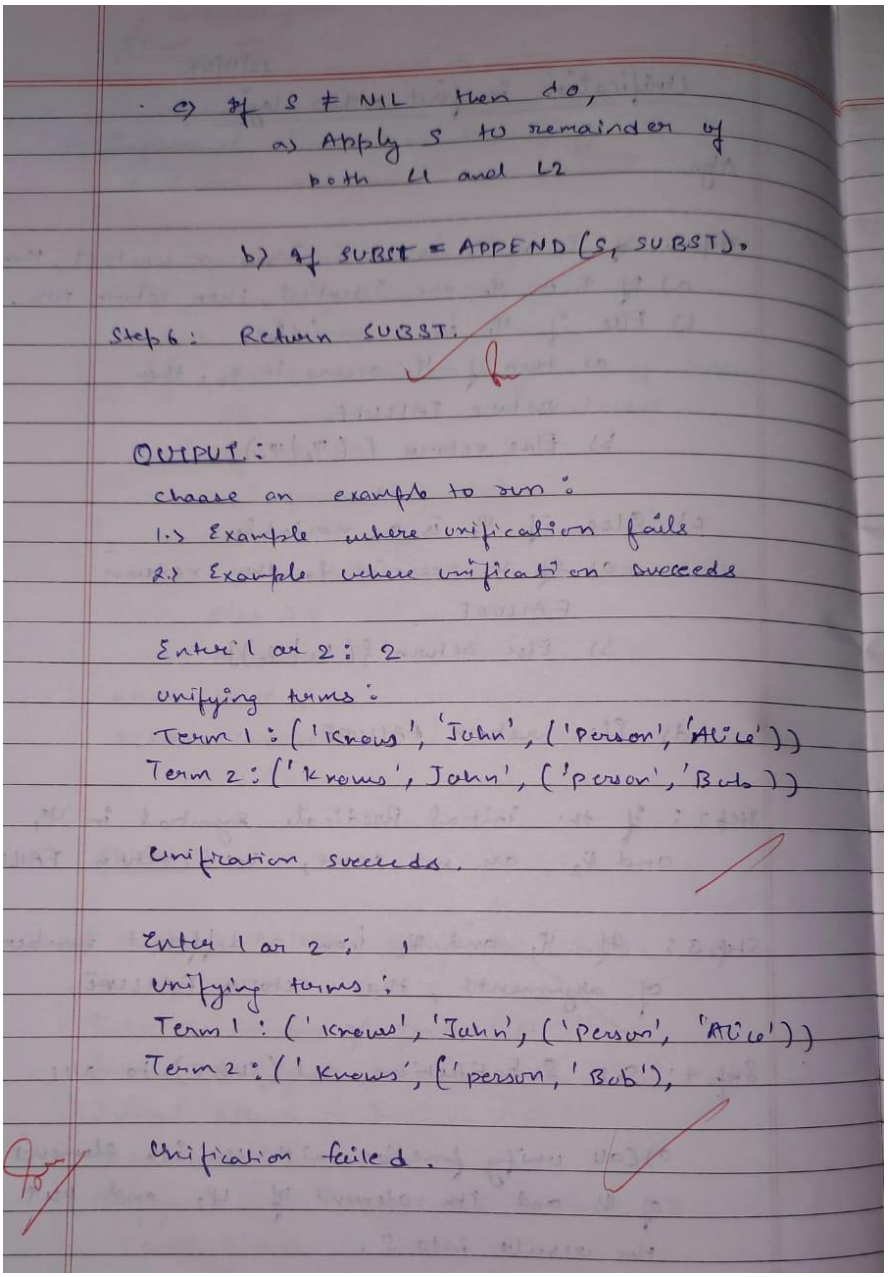
Step2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE

Step3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step4: Set Substitution set (SUBT) to NIL

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

b) If S = failure then return failure



c) If S ≠ NIL then do,
a) Apply S to remainder of both L1 and L2

b) If SUBST = APPEND (S, SUBST).

Step 6: Return SUBST.

OUTPUT:

Choose an example to run:
1.) Example where unification fails
2.) Example where unification succeeds

Enter 1 or 2: 2
Unifying terms:
Term 1: ('Knows', 'John', ('Person', 'Alice'))
Term 2: ('Knows', John', ('person', 'Bob'))

Unification succeeds.

Enter 1 or 2: 1
Unifying terms:
Term 1: ('Knows', 'John', ('Person', 'Alice'))
Term 2: ('Knows', ('person', 'Bob'),

Unification failed.

Code

```
def unify(s1, s2, theta={}):

    if theta is None:
        return None

    if s1 == s2:
        return theta

    if isinstance(s1, str) and s1.islower():
        return unify_var(s1, s2, theta)

    if isinstance(s2, str) and s2.islower():
        return unify_var(s2, s1, theta)
```

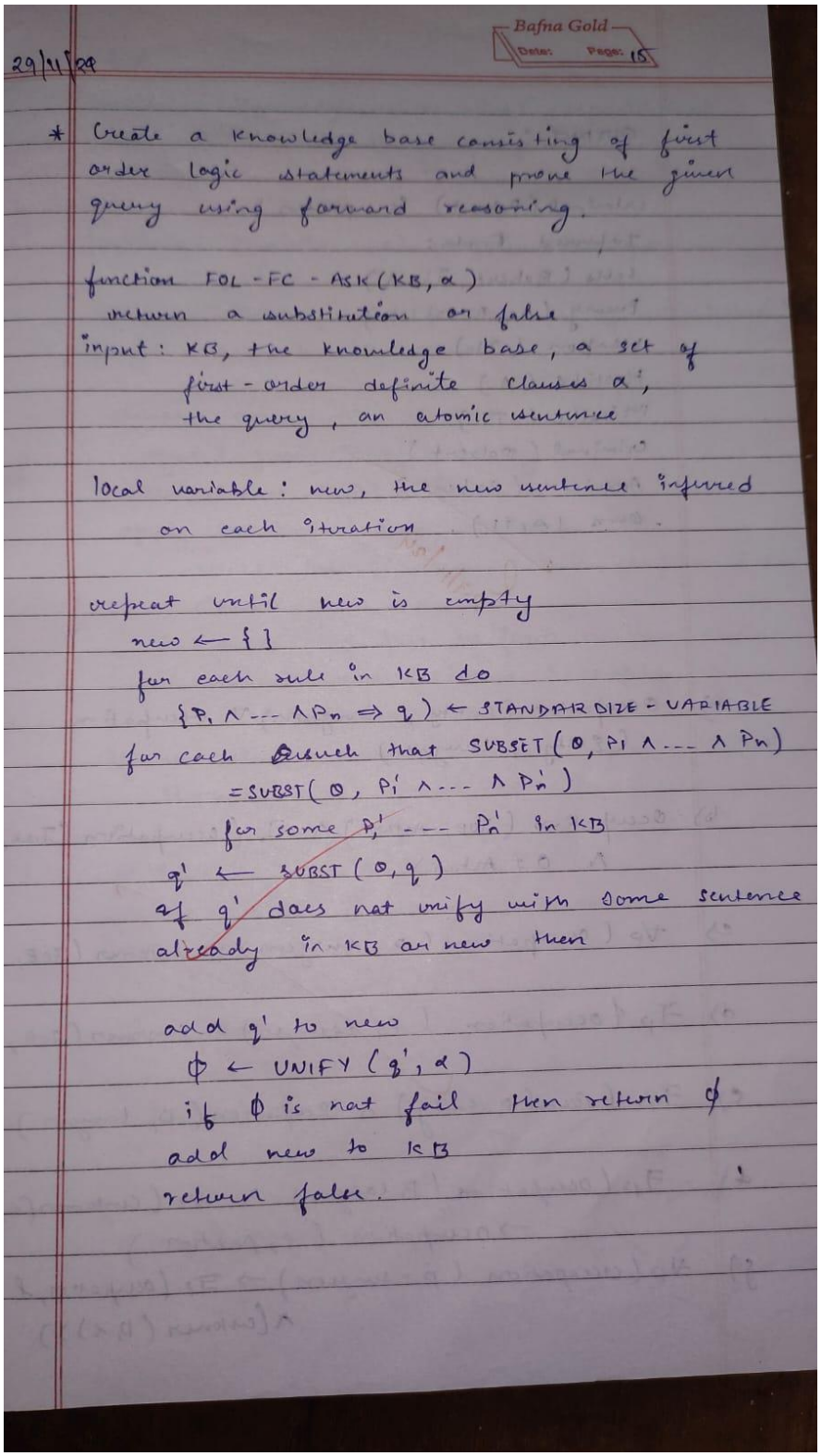
```
    if isinstance(s1, tuple) and isinstance(s2, tuple) and len(s1) == len(s2):
        return unify(s1[1:], s2[1:], unify(s1[0], s2[0], theta))

    return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    elif occurs_check(var, x, theta):
        return None
    else:
        theta[var] = x
        return theta

def occurs_check(var, x, theta):
    if var == x:
        return True
    elif isinstance(x, str) and x.islower() and x in theta:
        return occurs_check(var, theta[x], theta)
    elif isinstance(x, tuple):
        for arg in x:
            if occurs_check(var, arg, theta):
                return True
    return False


s1 = ('p', 'x', ('f', 'x'), ('y'))
s2 = ('p', 'a', 'y', ('f', 'x'))

substitution = unify(s1, s2)

if substitution:
    print("Unification successful:")
    print(f"Substitution: {substitution}")
else:
    print("Unification failed.")
```

Output Snapshot

\

Program-09 Forward Channing

Algorithm



29/11/24

Bafna Gold
Date: Page: 15

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

function FOL-FC-ASK(KB, α)
return a substitution or false
input: KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence

local variable: new, the new sentence inferred on each iteration

repeat until new is empty
new ← { }
for each rule in KB do
$(P_1 \wedge \dots \wedge P_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLE
for each θ such that SUBST(θ, $P_1 \wedge \dots \wedge P_n$) = SUBST(θ, $P_1' \wedge \dots \wedge P_n'$)
for some $P_1' \dots P_n'$ in KB
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then

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

Code

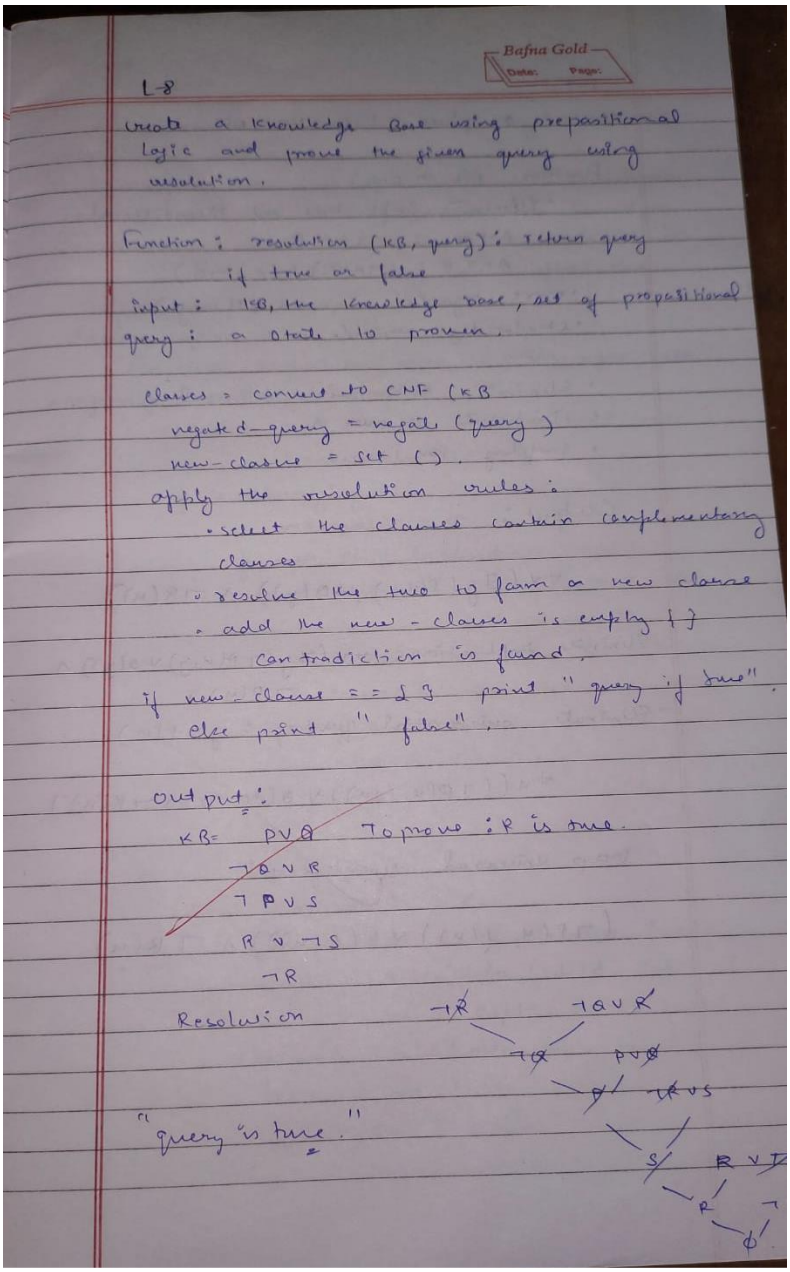
```
def forward_reasoning_algorithm():
    print("=== Forward Reasoning Algorithm ===")
    print("Enter the knowledge base (rules and facts), one per line.")
    print("Rules should be in the format: premise1 AND premise2 => conclusion")
    print("Facts should be entered as standalone atomic sentences.")
    print("Enter 'END' to finish entering the knowledge base.\n")

    # Initialize the knowledge base and facts
    knowledge_base = []
    facts = set()

    # Input: Knowledge base (rules and facts)
    while True:
        line = input("Enter rule or fact: ").strip()
        if line.upper() == "END":
            break
        if "=>" in line:  # Rule with premises and conclusion
            premises, conclusion = line.split(" => ")
            knowledge_base.append((premises.split(" AND "), conclusion.strip()))
        else:  # Fact
            facts.add(line.strip())

    print("\n=== Knowledge Base and Initial Facts ===")
    print("Rules:")
    for premises, conclusion in knowledge_base:
        print(f"  {' AND '.join(premises)} => {conclusion}")
    print("Facts:")
    for fact in facts:
        print(f"  {fact}")
    print()

    # Input: Query
    query = input("Enter the query (atomic sentence): ").strip()
    print("\n=== Forward Reasoning Process ===")

    # Forward-chaining algorithm
    inferred = set()  # Store all inferred facts
    new_inferences = True

    while new_inferences:
        new_inferences = False
        for premises, conclusion in knowledge_base:
            # Check if all premises are satisfied in the current set of facts
            if all(p in facts for p in premises) and conclusion not in facts:
                # Infer the conclusion
                facts.add(conclusion)
                inferred.add(conclusion)
                print(f"Inferred: {conclusion}")
                new_inferences = True

    # Check if the query can be inferred
    print("\n=== Query Result ===")
    if query in facts:
        print(f"The query '{query}' is satisfied: YES")
    else:
        print(f"The query '{query}' is not satisfied: NO")
    print("=== End of Process ===")


# Run the algorithm
forward_reasoning_algorithm()
```

Output Snapshot

```
PS C:\Users\YASH KUMAR SINHA\OneDrive\Desktop\AI lab\lab 7> python -u "c:\Users\YASH KUMAR SINHA\OneDrive\Desktop\AI lab\lab 7\FOL_ForwardChanning.py"
=== Forward Reasoning Algorithm ===
Enter the knowledge base (rules and facts), one per line.
Rules should be in the format: premise1 AND premise2 => conclusion
Facts should be entered as standalone atomic sentences.
Enter 'END' to finish entering the knowledge base.

Enter rule or fact: A
Enter rule or fact: B
Enter rule or fact: A AND B => C
Enter rule or fact: C AND D => E
Enter rule or fact: END

=== Knowledge Base and Initial Facts ===
Rules:
  A AND B => C
  C AND D => E
Facts:
  A
  B

Enter the query (atomic sentence): C

=== Forward Reasoning Process ===
Inferred: C

=== Query Result ===
The query 'C' is satisfied: YES
=== End of Process ===
PS C:\Users\YASH KUMAR SINHA\OneDrive\Desktop\AI lab\lab 7>
```

Program-10 Preposition Logic Using Resolution

Algorithm



Code

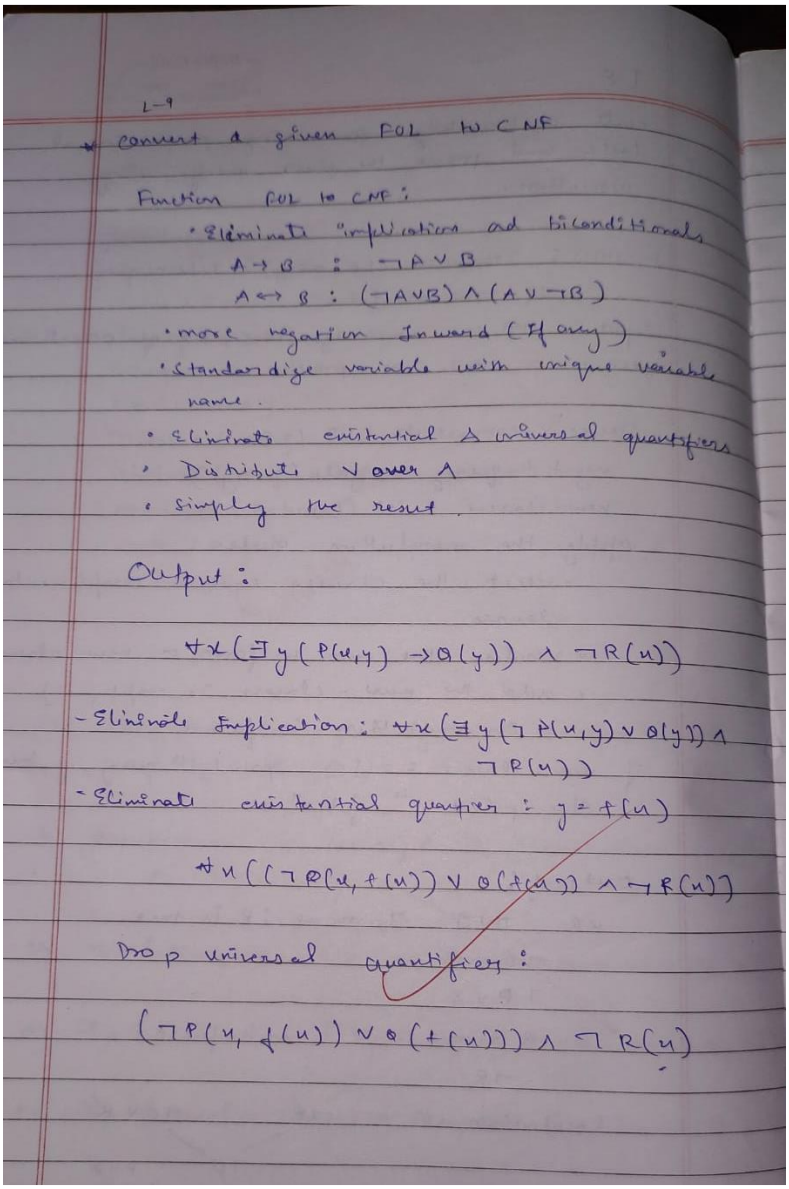
```python
from sympy.logic.boolalg import Or, And, Not, Implies
from sympy import symbols

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)
    else:
        print("The query is proved (contradiction found).")

# Example usage
if __name__ == "__main__":
    knowledge_base_resolution()
```

Output Snapshot

```
Knowledge Base in CNF: False
The query is proved (contradiction found).
```

Program-11 FOL To CNF

Algorithm



L-9

* Convert a given FOL to CNF

Function FOL to CNF:

- Eliminate implication and biconditionals
  - $A \rightarrow B$: $\neg A \vee B$
  - $A \leftrightarrow B$: $(\neg A \vee B) \wedge (A \vee \neg B)$
- move negation inward (If any)
- Standardize variable with unique variable name.
- Eliminate existential & universal quantifiers
- Distribute $\vee$ over $\wedge$
- Simplify the result.

Output:

$$\forall x (\exists y (P(x,y) \rightarrow Q(y)) \wedge \neg R(x))$$

- Eliminate Implication: $\forall x (\exists y (\neg P(x,y) \vee Q(y)) \wedge \neg P(x))$
- Eliminate existential quantifier: $y = f(x)$

$$\forall x ((\neg P(x, f(x)) \vee Q(f(x))) \wedge \neg R(x))$$

Drop universal quantifier:

$$(\neg P(x, f(x)) \vee Q(f(x))) \wedge \neg R(x)$$

Code

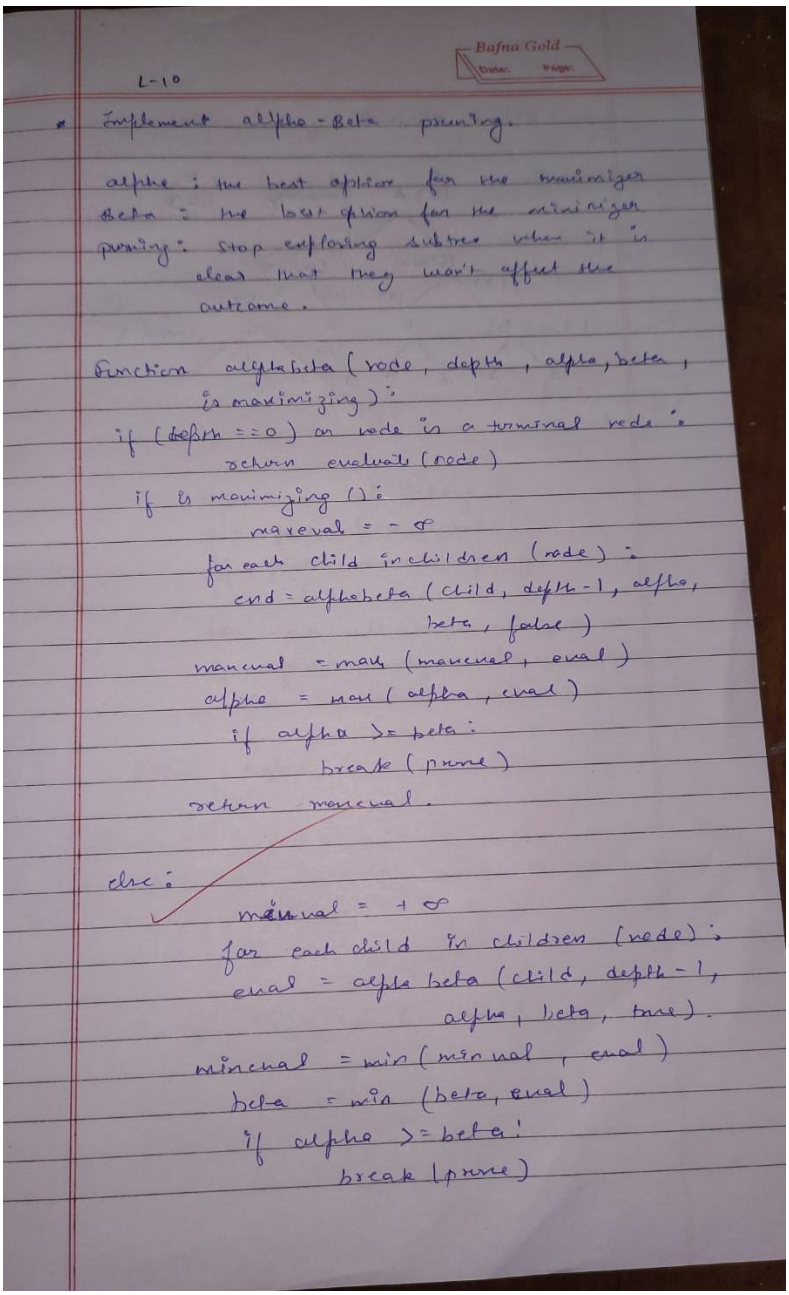
```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

 if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)

 else:
        print("The query is proved (contradiction found).")

# Example usage for converting FOL to CNF
if __name__ == "__main__":
    # Define symbols for FOL example
    A, B, C = symbols('A B C')

    # Example FOL statement: (A -> B) AND (B -> C)
    fol_statement = And(Implies(A, B), Implies(B, C))

    # Convert to CNF
    cnf_statement = convert_to_cnf(fol_statement)
    print("Original FOL Statement:", fol_statement)
    print("Converted CNF Statement:", cnf_statement)

    # Run resolution demonstration
    knowledge_base_resolution()
```

Output Snapshot

```
Original FOL Statement: (Implies(A, B)) & (Implies(B, C))
Converted CNF Statement: (B | ~A) & (C | ~B)
Knowledge Base in CNF: False
The query is proved (contradiction found).
```

Program-12 Alpha Beta Pruning

Algorithm



L-10

* Implement alpha-Beta pruning.

alpha : the best option for the maximizer
Beta : the best option for the minimizer
pruning: Stop exploring subtree when it is clear that they won't affect the outcome.

function alphabeta (node, depth, alpha, beta, is maximizing):
if (depth == 0) or node is a terminal node:
return evaluate (node)
if is maximizing ():
maxeval = - ∞
for each child in children (node):
eval = alphabeta (child, depth-1, alpha, beta, false)
maxeval = max (maxeval, eval)
alpha = max (alpha, eval)
if alpha >= beta:
break (prune)
return maxeval.

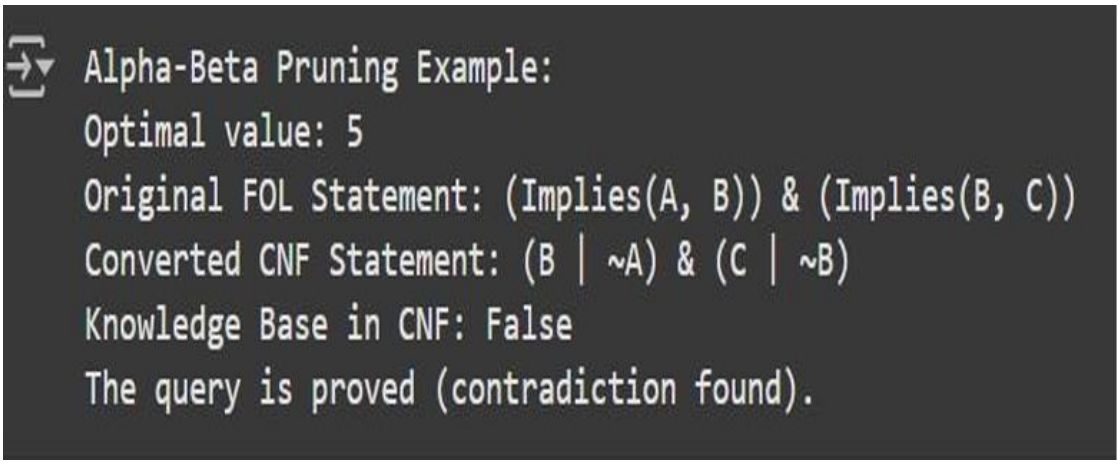
else:
minval = + ∞
for each child in children (node):
eval = alpha beta (child, depth-1, alpha, beta, true).
minval = min (minval, eval)
beta = min (beta, eval)
if alpha >= beta:
break (prune)

Code

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def alpha_beta_pruning(depth, node_index, maximizing_player, values, alpha, beta):
    """
    Implement the Alpha-Beta Pruning algorithm.

    Parameters:
        depth (int): Current depth in the game tree.
        node_index (int): Index of the current node in the game tree.
        maximizing_player (bool): True if the current player is maximizing, False otherwise.
        values (list): Terminal node values (leaf nodes).
        alpha (float): Alpha value for pruning.
        beta (float): Beta value for pruning.

    Returns:
        int: The optimal value for the current player.
    """
    if depth == 0 or node_index >= len(values):
        return values[node_index]

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # Assume binary tree
            eval = alpha_beta_pruning(depth - 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # Assume binary tree
            eval = alpha_beta_pruning(depth - 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
53
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)
```

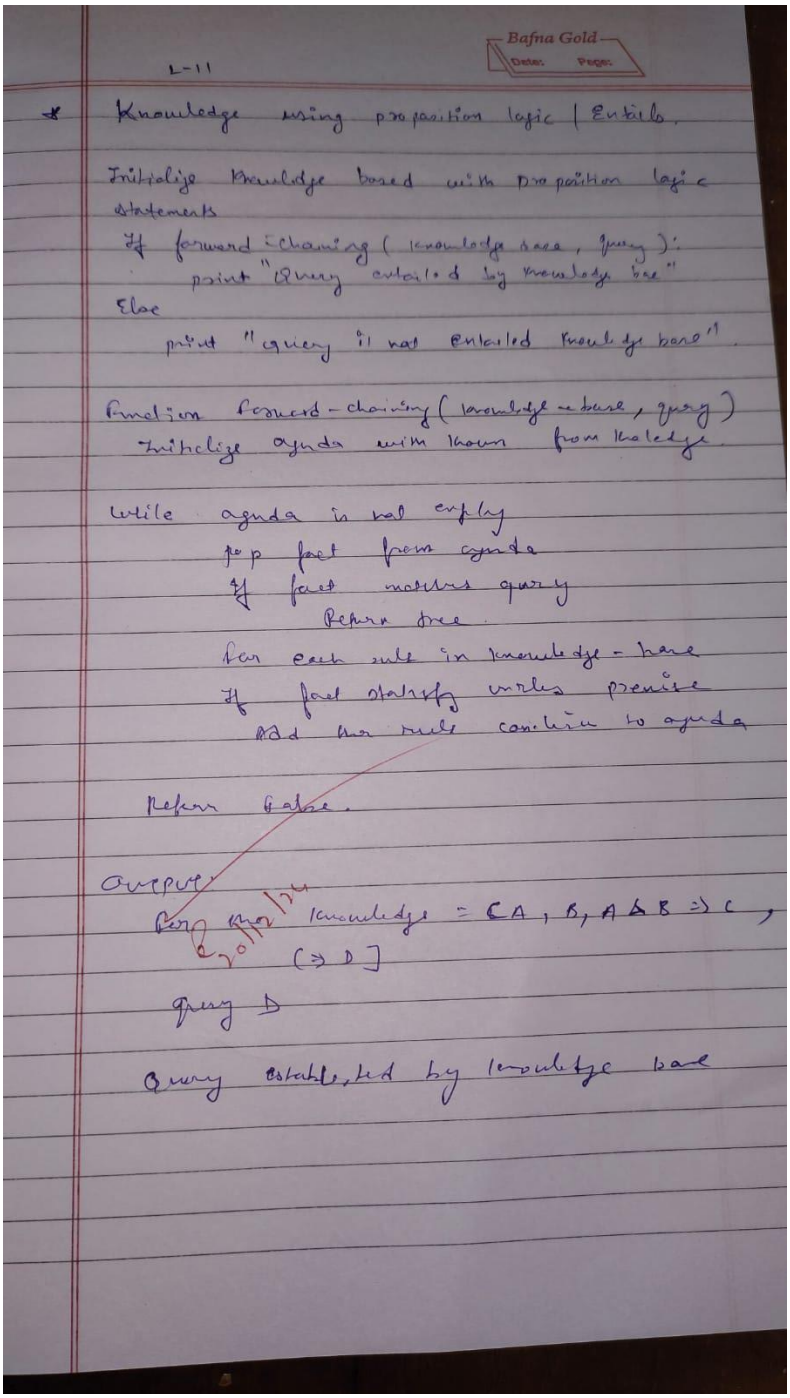
```
    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)
    else:
        print("The query is proved (contradiction found).")

# Example usage for converting FOL to CNF
if __name__ == "__main__":
    # Example usage of Alpha-Beta Pruning
    print("Alpha-Beta Pruning Example:")
    values = [3, 5, 6, 9, 1, 2, 0, -1]  # Leaf nodes of the game tree
    depth = 3  # Depth of the tree
    optimal_value = alpha_beta_pruning(depth, 0, True, values, float('-inf'), float('inf'))
    print("Optimal value:", optimal_value)

    # Define symbols for FOL example
    A, B, C = symbols('A B C')

    # Example FOL statement: (A -> B) AND (B -> C)
    fol_statement = And(Implies(A, B), Implies(B, C))
54
# Convert to CNF
cnf_statement = convert_to_cnf(fol_statement)
print("Original FOL Statement:", fol_statement)
print("Converted CNF Statement:", cnf_statement)
# Run resolution demonstration
knowledge_base_resolution()
```

Output SnapShot



Program-13 Preposition Logic Query Entails

Algorithm



Code

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def check_entailment(kb, query):
    """
    Check if the given query is entailed by the knowledge base (KB) using resolution.

    Parameters:
        kb (Expr): The knowledge base in propositional logic.
        query (Expr): The query to check for entailment.

    Returns:
        str: Result indicating whether the query is entailed or not.
    """
    # Step 1: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 2: Convert KB with negated query to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:\n", kb_cnf)

    # Step 3: Apply Resolution
    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        return "The query is NOT entailed by the knowledge base (no contradiction found)."
    else:
        return "The query is entailed by the knowledge base (contradiction found)."

if __name__ == "__main__":
    # Define symbols for the knowledge base and query
    P, Q, R = symbols('P Q R')

    # Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )
57
# Define the query
query = R
# Check entailment
print("Knowledge Base:", kb)
print("Query:", query)
result = check_entailment(kb, query)
print("Entailment Result:", result)
```

Output SnapShot