**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Saahya K S (1BM23CS368)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

****

**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Aug 2025 to Dec 2025**

**B.M.S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

****

**CERTIFICATE**

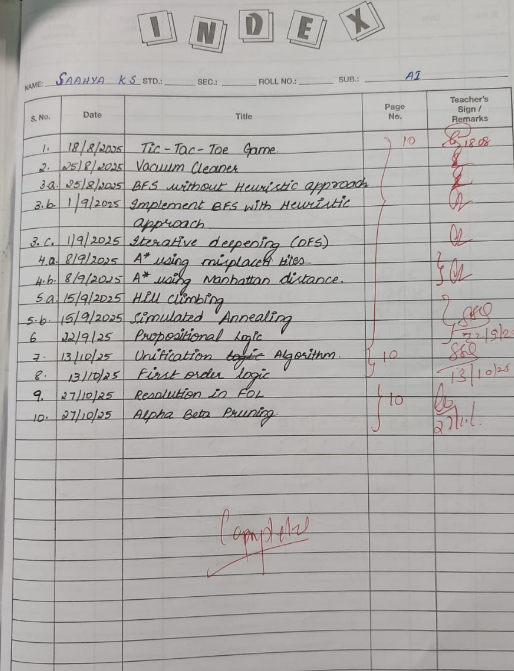
This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Saahya K S (1BM23CS368),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 18-8-2025 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent | 5-10 |
| 2 | 25-8-2025 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 11-19 |
| 3 | 8-9-2025 | Implement A\* search algorithm | 20-23 |
| 4 | 15-9-2025 | Implement Hill Climbing search algorithm to solve N-Queens problem | 24-26 |
| 5 | 15-9-2025 | Simulated Annealing to Solve 8-Queens problem | 27-29 |
| 6 | 22-9-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 30-33 |
| 7 | 13-10-2025 | Implement unification in first order logic | 34-40 |
| 8 | 13-10-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 41-47 |
| 9 | 27-10-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 50-55 |
| 10 | 27-10-2025 | Implement Alpha-Beta Pruning. | 56-59 |

Index:



Github Link:

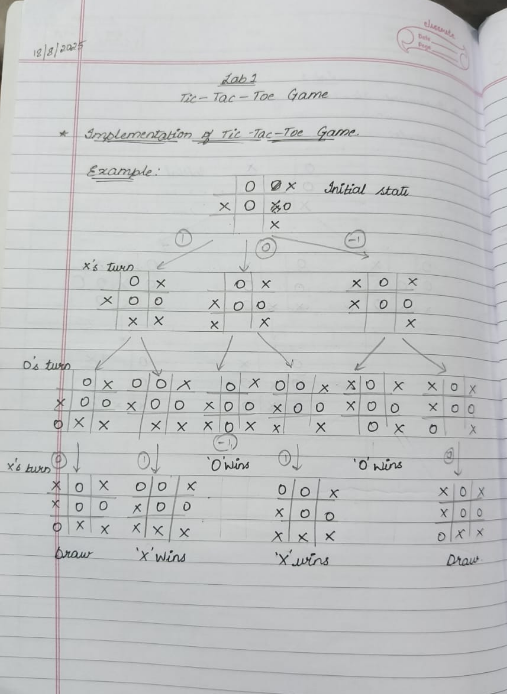
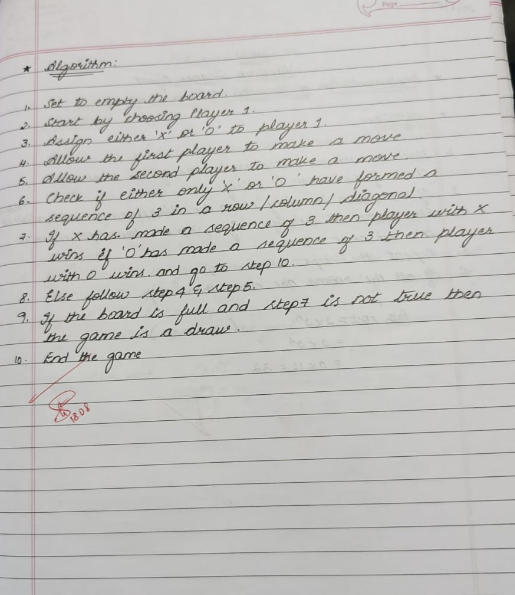
<https://github.com/Saahya-KS17/AI-lab>

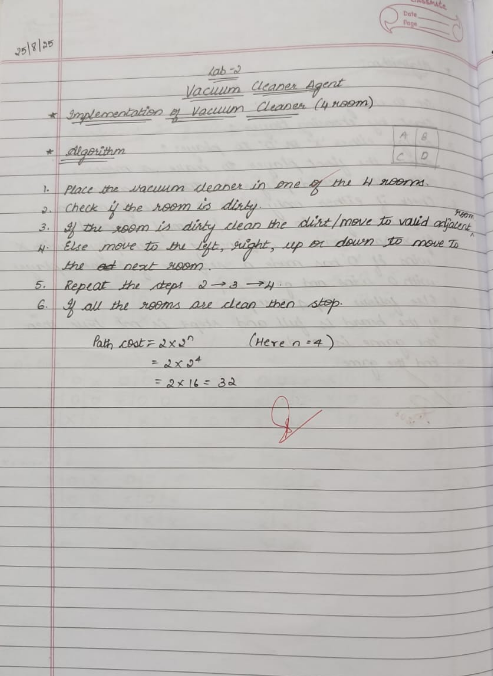
**Program 1**

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Code:

**TIC TAC TOE Game**

def print\_board(board):

print()

print("-------------")

for i in range(3):

print("|", board[i][0], "|", board[i][1], "|", board[i][2], "|")

print("-------------")

print()

def is\_win(board,symbol):

*#same row*

for i in range(3):

if board[i][0] == symbol and board[i][1] == symbol and board[i][2] == symbol:

return True

*#same Column*

for j in range(3):

if board[0][j] == symbol and board[1][j] == symbol and board[2][j] == symbol:

return True

*#same diagonal*

if board[0][0] == symbol and board[1][1] == symbol and board[2][2] == symbol:

return True

if board[0][2] == symbol and board[1][1] == symbol and board[2][0] == symbol:

return True

*#if all case fails*

return False

def is\_draw(board):

for i in range(3):

for j in range(3):

if board[i][j] == " ":

return False

return True

def get\_input(mark,id):

while True:

try:

move = input(f"Player{id}, Enter the position to place {mark}-> ")

ip = move.strip().split()

if len(ip) != 2:

print("Enter exactly 2 coordinates!")

continue

row,col = int(ip[0]),int(ip[1])

if (row>=1 and row<= 3) and (col>=1 and col<=3):

return [row-1,col-1]

else:

print("Enter values between 1 and 3 only")

except ValueError:

print("Invalid inputs, enter numbers only")

board = []

for i in range(3):

board.append([" "," "," "])

mark1 = "X"

mark2 = "O"

p1 = 1

p2 = 2

curr\_p = 1

curr\_m = "X"

while True:

print\_board(board)

row,col = get\_input(curr\_m,curr\_p)

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

print("Move already taken!,Try again")

continue

board[row][col] = curr\_m

if is\_win(board,curr\_m):

print\_board(board)

print(f"Congrats🎉, Player{curr\_p} wins!!!")

break

if is\_draw(board):

print\_board(board)

print("Its a draw,🫡!")

break

if curr\_p == p1:

curr\_p = p2

curr\_m = "O"

else:

curr\_p = p1

curr\_m = "X"

-------------

| | | |

-------------

| | | |

-------------

| | | |

-------------

Player1, Enter the position to place X-> 1 1

-------------

| X | | |

-------------

| | | |

-------------

| | | |

-------------

Player2, Enter the position to place O-> 3 1

-------------

| X | | |

-------------

| | | |

-------------

| O | | |

-------------

Player1, Enter the position to place X-> 2 2

-------------

| X | | |

-------------

| | X | |

-------------

| O | | |

-------------

Player2, Enter the position to place O-> 3 2

-------------

| X | | |

-------------

| | X | |

-------------

| O | O | |

-------------

Player1, Enter the position to place X-> 3 3

-------------

| X | | |

-------------

| | X | |

-------------

| O | O | X |

-------------

Congrats🎉, Player1 wins!!!

**Vacuum Cleaner Agent**

def is\_clean(status):

return status[room\_a] and status[room\_b]

def simulate(state, choice, status, cost, do\_clean=True):

if is\_clean(status):

print("All rooms are clean")

return cost

if choice != 1 and choice != -1:

print("Invalid choice")

return cost

*# Vacuum in room A*

if state[0][0]:

if choice == -1:

if do\_clean and not state[0][1]:

state[0][1] = True

status[room\_a] = True

print("Cleaned room A")

cost += 1 *# Cost of cleaning*

else:

print("No cleaning in room A")

elif choice == 1:

state[0][0] = False

state[1][0] = True

print("Moved vacuum from A to B")

else:

print("Cannot move from A to B")

*# Vacuum in room B*

elif state[1][0]:

if choice == 1:

if do\_clean and not state[1][1]:

state[1][1] = True

status[room\_b] = True

print("Cleaned room B")

cost += 1 *# Cost of cleaning*

else:

print("No cleaning in room B")

elif choice == -1:

state[1][0] = False

state[0][0] = True

print("Moved vacuum from B to A")

else:

print("Cannot move from B to A")

else:

print("Vacuum is not in any room!")

return cost

if \_\_name\_\_ == "\_\_main\_\_":

room\_a = 'A'

room\_b = 'B'

state = [[True, False], [False, False]]

status = {room\_a:False, room\_b:False}

total\_cost = 0 *# Initialize total cost*

while True:

if is\_clean(status):

print("All rooms are clean. Exiting.")

print(f"Total cost: {total\_cost}") *# Display total cost*

break

choice = int(input("Enter -1 to act in Room A, 1 to act in Room B: "))

action = input("Enter 'c' to clean, 'm' to move without cleaning: ").lower()

if action == 'c':

total\_cost = simulate(state, choice, status, total\_cost)

elif action == 'm':

total\_cost = simulate(state, choice, status, total\_cost, False)

else:

print("Invalid action choice")

Enter -1 to act in Room A, 1 to act in Room B: -1

Enter 'c' to clean, 'm' to move without cleaning: c

Cleaned room A

Enter -1 to act in Room A, 1 to act in Room B: 1

Enter 'c' to clean, 'm' to move without cleaning: m

Moved vacuum from A to B

Enter -1 to act in Room A, 1 to act in Room B: 1

Enter 'c' to clean, 'm' to move without cleaning: c

Cleaned room B

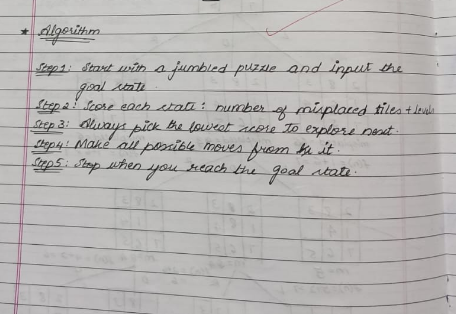
All rooms are clean. Exiting.

Total cost: 2

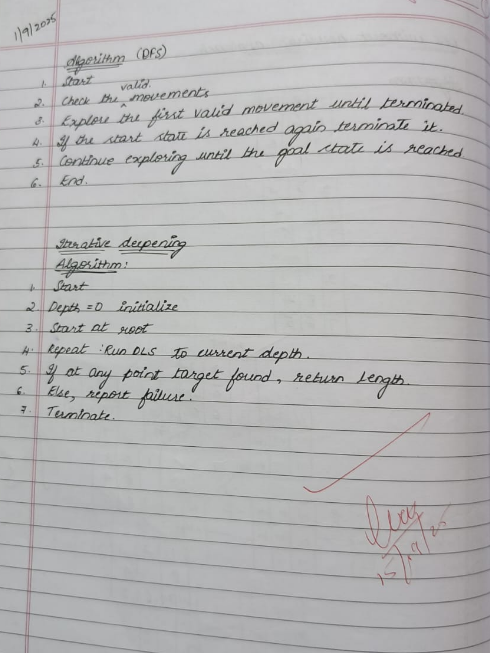
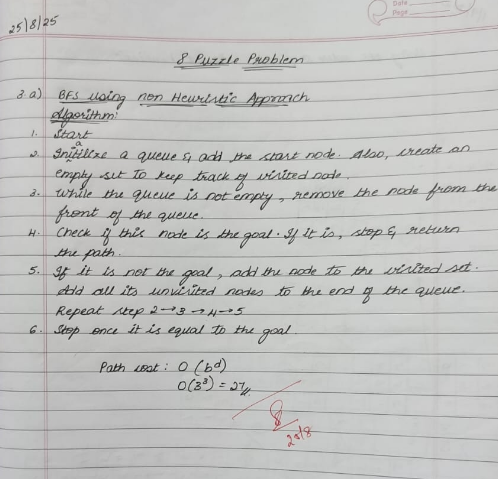
**Program 2**

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm



Algorithm:

Code:

**8 Puzzle using DFS**

from collections import deque

*# Helper to print board in 3x3 format*

def print\_board(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print()

*# Generate possible moves*

def get\_neighbors(state):

neighbors = []

idx = state.index(0) *# blank space*

row, col = divmod(idx, 3)

moves = []

if row > 0: moves.append((-1, 0, 'Up'))

if row < 2: moves.append((1, 0, 'Down'))

if col > 0: moves.append((0, -1, 'Left'))

if col < 2: moves.append((0, 1, 'Right'))

for dr, dc, action in moves:

new\_row, new\_col = row + dr, col + dc

new\_idx = new\_row \* 3 + new\_col

new\_state = list(state)

new\_state[idx], new\_state[new\_idx] = new\_state[new\_idx], new\_state[idx]

neighbors.append((tuple(new\_state), action))

return neighbors

*# DFS Search*

def solve\_puzzle(start, goal, max\_depth=50):

stack = [(start, [])]

explored = set()

while stack:

state, path = stack.pop()

if state in explored:

continue

explored.add(state)

if state == goal:

return path

if len(path) < max\_depth:

for neighbor, action in get\_neighbors(state):

if neighbor not in explored:

stack.append((neighbor, path + [(action, neighbor)]))

return None

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter the initial state (0 for blank, space-separated, 9 numbers):")

start = tuple(map(int, input().split()))

print("Enter the goal state (0 for blank, space-separated, 9 numbers):")

goal = tuple(map(int, input().split()))

print("\nSolving puzzle with DFS...\n")

solution = solve\_puzzle(start, goal)

if solution:

print("Solution found using DFS! (may not be optimal)\n")

print("Total steps:", len(solution))

current = start

print("Initial State:")

print\_board(current)

for step, state in solution:

print("Move:", step)

print\_board(state)

else:

print("No solution found within depth limit.")

Enter the initial state (0 for blank, space-separated, 9 numbers):

2 8 3 1 6 4 7 0 5

Enter the goal state (0 for blank, space-separated, 9 numbers):

1 2 3 8 0 4 7 6 5

Solving puzzle with DFS...

Solution found using DFS! (may not be optimal)

Total steps: 49

Initial State:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Move: Right

(2, 8, 3)

(1, 6, 4)

(7, 5, 0)

Move: Up

(2, 8, 3)

(1, 6, 0)

(7, 5, 4)

Move: Left

(2, 8, 3)

(1, 0, 6)

(7, 5, 4)

Move: Left

(2, 8, 3)

(0, 1, 6)

(7, 5, 4)

Move: Down

(2, 8, 3)

(7, 1, 6)

(0, 5, 4)

Move: Right

(2, 8, 3)

(7, 1, 6)

(5, 0, 4)

Move: Right

(2, 8, 3)

(7, 1, 6)

(5, 4, 0)

Move: Up

(2, 8, 3)

(7, 1, 0)

(5, 4, 6)

Move: Left

(2, 8, 3)

(7, 0, 1)

(5, 4, 6)

Move: Left

(2, 8, 3)

(0, 7, 1)

(5, 4, 6)

Move: Down

(2, 8, 3)

(5, 7, 1)

(0, 4, 6)

Move: Right

(2, 8, 3)

(5, 7, 1)

(4, 0, 6)

Move: Right

(2, 8, 3)

(5, 7, 1)

(4, 6, 0)

Move: Up

(2, 8, 3)

(5, 7, 0)

(4, 6, 1)

Move: Left

(2, 8, 3)

(5, 0, 7)

(4, 6, 1)

Move: Left

(2, 8, 3)

(0, 5, 7)

(4, 6, 1)

Move: Down

(2, 8, 3)

(4, 5, 7)

(0, 6, 1)

Move: Right

(2, 8, 3)

(4, 5, 7)

(6, 0, 1)

Move: Right

(2, 8, 3)

(4, 5, 7)

(6, 1, 0)

Move: Up

(2, 8, 3)

(4, 5, 0)

(6, 1, 7)

Move: Left

(2, 8, 3)

(4, 0, 5)

(6, 1, 7)

Move: Left

(2, 8, 3)

(0, 4, 5)

(6, 1, 7)

Move: Down

(2, 8, 3)

(6, 4, 5)

(0, 1, 7)

Move: Right

(2, 8, 3)

(6, 4, 5)

(1, 0, 7)

Move: Right

(2, 8, 3)

(6, 4, 5)

(1, 7, 0)

Move: Up

(2, 8, 3)

(6, 4, 0)

(1, 7, 5)

Move: Left

(2, 8, 3)

(6, 0, 4)

(1, 7, 5)

Move: Down

(2, 8, 3)

(6, 7, 4)

(1, 0, 5)

Move: Left

(2, 8, 3)

(6, 7, 4)

(0, 1, 5)

Move: Up

(2, 8, 3)

(0, 7, 4)

(6, 1, 5)

Move: Right

(2, 8, 3)

(7, 0, 4)

(6, 1, 5)

Move: Right

(2, 8, 3)

(7, 4, 0)

(6, 1, 5)

Move: Down

(2, 8, 3)

(7, 4, 5)

(6, 1, 0)

Move: Left

(2, 8, 3)

(7, 4, 5)

(6, 0, 1)

Move: Up

(2, 8, 3)

(7, 0, 5)

(6, 4, 1)

Move: Right

(2, 8, 3)

(7, 5, 0)

(6, 4, 1)

Move: Down

(2, 8, 3)

(7, 5, 1)

(6, 4, 0)

Move: Left

(2, 8, 3)

(7, 5, 1)

(6, 0, 4)

Move: Up

(2, 8, 3)

(7, 0, 1)

(6, 5, 4)

Move: Right

(2, 8, 3)

(7, 1, 0)

(6, 5, 4)

Move: Down

(2, 8, 3)

(7, 1, 4)

(6, 5, 0)

Move: Left

(2, 8, 3)

(7, 1, 4)

(6, 0, 5)

Move: Left

(2, 8, 3)

(7, 1, 4)

(0, 6, 5)

Move: Up

(2, 8, 3)

(0, 1, 4)

(7, 6, 5)

Move: Right

(2, 8, 3)

(1, 0, 4)

(7, 6, 5)

Move: Up

(2, 0, 3)

(1, 8, 4)

(7, 6, 5)

Move: Left

(0, 2, 3)

(1, 8, 4)

(7, 6, 5)

Move: Down

(1, 2, 3)

(0, 8, 4)

(7, 6, 5)

Move: Right

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

**8 Puzzle using Iterative Deepening DFS**

from collections import deque

*# Helper to print board in 3x3 format*

def print\_board(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print()

*# Generate possible moves*

def get\_neighbors(state):

neighbors = []

idx = state.index(0) *# blank space*

row, col = divmod(idx, 3)

moves = []

if row > 0: moves.append((-1, 0, 'Up'))

if row < 2: moves.append((1, 0, 'Down'))

if col > 0: moves.append((0, -1, 'Left'))

if col < 2: moves.append((0, 1, 'Right'))

for dr, dc, action in moves:

new\_row, new\_col = row + dr, col + dc

new\_idx = new\_row \* 3 + new\_col

new\_state = list(state)

new\_state[idx], new\_state[new\_idx] = new\_state[new\_idx], new\_state[idx]

neighbors.append((tuple(new\_state), action))

return neighbors

*# Depth-Limited Search (helper for IDDFS)*

def dls(state, goal, depth, path, explored):

if state == goal:

return path

if depth == 0:

return None

explored.add(state)

for neighbor, action in get\_neighbors(state):

if neighbor not in explored:

result = dls(neighbor, goal, depth-1, path + [(action, neighbor)], explored)

if result is not None:

return result

return None

*# Iterative Deepening DFS*

def iddfs(start, goal, max\_depth=50):

for depth in range(max\_depth):

explored = set()

result = dls(start, goal, depth, [], explored)

if result is not None:

return result

return None

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter the initial state (0 for blank, space-separated, 9 numbers):")

start = tuple(map(int, input().split()))

print("Enter the goal state (0 for blank, space-separated, 9 numbers):")

goal = tuple(map(int, input().split()))

print("\nSolving puzzle with Iterative Deepening DFS...\n")

solution = iddfs(start, goal)

if solution:

print("Optimal solution found using IDDFS!\n")

print("Total steps:", len(solution))

current = start

print("Initial State:")

print\_board(current)

for step, state in solution:

print("Move:", step)

print\_board(state)

else:

print("No solution found within depth limit.")

Enter the initial state (0 for blank, space-separated, 9 numbers):

2 8 3 1 6 4 7 0 5

Enter the goal state (0 for blank, space-separated, 9 numbers):

1 2 3 8 0 4 7 6 5

Solving puzzle with Iterative Deepening DFS...

Optimal solution found using IDDFS!

Total steps: 5

Initial State:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Move: Up

(2, 8, 3)

(1, 0, 4)

(7, 6, 5)

Move: Up

(2, 0, 3)

(1, 8, 4)

(7, 6, 5)

Move: Left

(0, 2, 3)

(1, 8, 4)

(7, 6, 5)

Move: Down

(1, 2, 3)

(0, 8, 4)

(7, 6, 5)

Move: Right

(1, 2, 3)

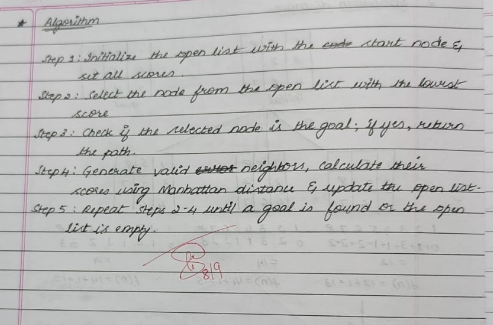
(8, 0, 4)

(7, 6, 5)

**Program 3:**

Implement A\* search algorithm

Algorithm:



Code:

**A\* using misplaced tiles for 8 puzzle**

from heapq import heappush, heappop

MOVES = {

'up': -3,

'down': 3,

'left': -1,

'right': 1

}

def is\_valid\_move(zero\_pos, move):

if move == 'left' and zero\_pos % 3 == 0:

return False

if move == 'right' and zero\_pos % 3 == 2:

return False

if move == 'up' and zero\_pos < 3:

return False

if move == 'down' and zero\_pos > 5:

return False

return True

def misplaced\_tiles(state, goal):

return sum([1 if state[i] != 0 and state[i] != goal[i] else 0 for i in range(9)])

def print\_state\_formatted(state):

for i in range(0, 9, 3):

row = state[i:i+3]

print(" ".join(str(x) if x != 0 else " " for x in row))

print()

def a\_star\_misplaced(start, goal):

open\_set = []

closed\_set = set()

g = 0

h = misplaced\_tiles(start, goal)

f = g + h

heappush(open\_set, (f, g, start, [], None)) *# last element: move that got here*

while open\_set:

f, g, current, path, move\_made = heappop(open\_set)

*# Print detailed info with matrix format and move made*

if move\_made is None:

print(f"Expanding node with f={f}, g={g} - Start state")

else:

print(f"Expanding node with f={f}, g={g} - Move: {move\_made}")

print\_state\_formatted(current)

if current == goal:

print("Goal reached!")

print("Solution path (moves):", path)

print(f"Final depth (number of moves): {g}")

return path, g

closed\_set.add(current)

zero\_pos = current.index(0)

for move, shift in MOVES.items():

if is\_valid\_move(zero\_pos, move):

new\_zero\_pos = zero\_pos + shift

new\_state = list(current)

new\_state[zero\_pos], new\_state[new\_zero\_pos] = new\_state[new\_zero\_pos], new\_state[zero\_pos]

new\_state = tuple(new\_state)

if new\_state in closed\_set:

continue

new\_g = g + 1

new\_h = misplaced\_tiles(new\_state, goal)

new\_f = new\_g + new\_h

heappush(open\_set, (new\_f, new\_g, new\_state, path + [move], move))

print("No solution found.")

return None, None

def get\_state\_matrix(prompt):

print(prompt)

matrix = []

for i in range(3):

while True:

try:

row = list(map(int, input().strip().split()))

if len(row) != 3:

raise ValueError("Each row must have exactly 3 numbers.")

matrix.extend(row)

break

except Exception as e:

print("Invalid input:", e)

if set(matrix) != set(range(9)):

print("Error: The numbers must be from 0 to 8 without repetition.")

return get\_state\_matrix(prompt)

return tuple(matrix)

if \_\_name\_\_ == "\_\_main\_\_":

start\_state = get\_state\_matrix("Enter the initial state (3 rows, each with 3 numbers separated by spaces):")

goal\_state = get\_state\_matrix("Enter the goal state (3 rows, each with 3 numbers separated by spaces):")

print("\nInitial State:")

print\_state\_formatted(start\_state)

print("Goal State:")

print\_state\_formatted(goal\_state)

a\_star\_misplaced(start\_state, goal\_state)

Enter the initial state (3 rows, each with 3 numbers separated by spaces):

2 8 3

1 6 4

7 0 5

Enter the goal state (3 rows, each with 3 numbers separated by spaces):

1 2 3

8 0 4

7 6 5

Initial State:

2 8 3

1 6 4

7 5

Goal State:

1 2 3

8 4

7 6 5

Expanding node with f=4, g=0 - Start state

2 8 3

1 6 4

7 5

Expanding node with f=4, g=1 - Move: up

2 8 3

1 4

7 6 5

Expanding node with f=5, g=2 - Move: up

2 3

1 8 4

7 6 5

Expanding node with f=5, g=2 - Move: left

2 8 3

1 4

7 6 5

Expanding node with f=5, g=3 - Move: left

2 3

1 8 4

7 6 5

Expanding node with f=5, g=4 - Move: down

1 2 3

8 4

7 6 5

Expanding node with f=5, g=5 - Move: right

1 2 3

8 4

7 6 5

Goal reached!

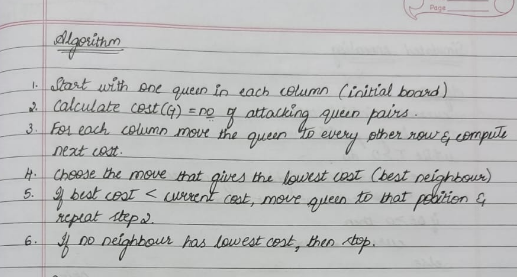
Solution path (moves): ['up', 'up', 'left', 'down', 'right']

Final depth (number of moves): 5

**Program 4**

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

def get\_user\_input(n):

board = []

print(f"Enter the row positions (0 to {n-1}) of the queens for each column:")

for col in range(n):

while True:

try:

row = int(input(f"Column {col}: "))

if 0 <= row < n:

board.append(row)

break

else:

print(f"Invalid input. Please enter a number between 0 and {n-1}.")

except ValueError:

print("Invalid input. Please enter an integer.")

return board

def print\_board(board):

n = len(board)

for row in range(n):

line = ""

for col in range(n):

if board[col] == row:

line += " Q "

else:

line += " . "

print(line)

print()

def heuristic(board):

n = len(board)

attacks = 0

for i in range(n):

for j in range(i+1, n):

*# Check same row*

if board[i] == board[j]:

attacks += 1

*# Check same diagonal*

if abs(board[i] - board[j]) == abs(i - j):

attacks += 1

return attacks

def get\_best\_neighbor(board):

n = len(board)

current\_heuristic = heuristic(board)

best\_board = list(board)

best\_heuristic = current\_heuristic

for col in range(n):

for row in range(n):

if board[col] != row:

new\_board = list(board)

new\_board[col] = row

new\_heuristic = heuristic(new\_board)

if new\_heuristic < best\_heuristic:

best\_heuristic = new\_heuristic

best\_board = new\_board

return best\_board, best\_heuristic

def hill\_climbing\_with\_user\_input(n):

board = get\_user\_input(n)

current\_heuristic = heuristic(board)

steps = 0

while True:

print(f"Step {steps}: Heuristic = {current\_heuristic}")

print\_board(board)

if current\_heuristic == 0:

print("Solution found!")

return board

neighbor, neighbor\_heuristic = get\_best\_neighbor(board)

*# If no improvement, stuck at local minimum*

if neighbor\_heuristic >= current\_heuristic:

print("Reached local minimum (no better neighbors).")

return board

board = neighbor

current\_heuristic = neighbor\_heuristic

steps += 1

*# Run the algorithm*

if \_\_name\_\_ == "\_\_main\_\_":

n = 4

solution = hill\_climbing\_with\_user\_input(n)

print("Final solution:")

print\_board(solution)

Enter the row positions (0 to 3) of the queens for each column:

Column 0: 3

Column 1: 1

Column 2: 2

Column 3: 0

Step 0: Heuristic = 2

. . . Q

. Q . .

. . Q .

Q . . .

Reached local minimum (no better neighbors).

Final solution:

. . . Q

. Q . .

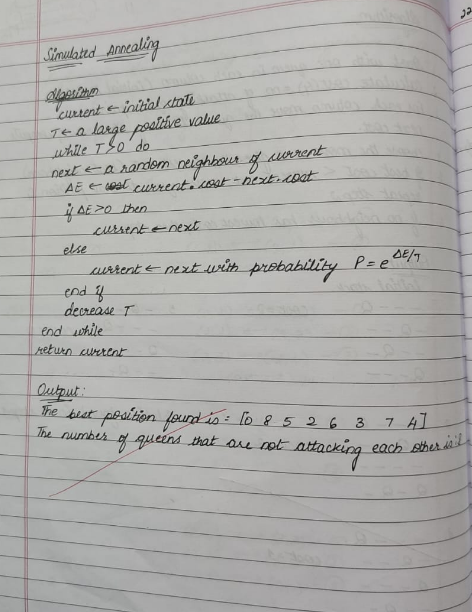
. . Q .

Q . . .

**Program 5**

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

import random

import math

def get\_user\_input(n):

board = []

print(f"Enter the row positions (0 to {n-1}) of the queens for each column:")

for col in range(n):

while True:

try:

row = int(input(f"Column {col}: "))

if 0 <= row < n:

board.append(row)

break

else:

print(f"Invalid input. Please enter a number between 0 and {n-1}.")

except ValueError:

print("Invalid input. Please enter an integer.")

return board

def print\_board(board):

n = len(board)

for row in range(n):

line = ""

for col in range(n):

line += " Q " if board[col] == row else " . "

print(line)

print()

def heuristic(board):

n = len(board)

attacks = 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):

attacks += 1

return attacks

def random\_neighbor(board):

n = len(board)

neighbor = list(board)

col = random.randint(0, n-1)

row = random.randint(0, n-1)

while row == neighbor[col]:

row = random.randint(0, n-1)

neighbor[col] = row

return neighbor

def simulated\_annealing(n=8, max\_iter=100000, initial\_temp=100, cooling\_rate=0.995):

current\_board = get\_user\_input(n)

current\_heuristic = heuristic(current\_board)

temperature = initial\_temp

iteration = 0

print(f"Initial heuristic: {current\_heuristic}")

print\_board(current\_board)

while temperature > 0.1 and current\_heuristic > 0 and iteration < max\_iter:

neighbor = random\_neighbor(current\_board)

neighbor\_heuristic = heuristic(neighbor)

delta\_e = current\_heuristic - neighbor\_heuristic

if delta\_e > 0:

current\_board = neighbor

current\_heuristic = neighbor\_heuristic

else:

probability = math.exp(delta\_e / temperature)

if random.random() < probability:

current\_board = neighbor

current\_heuristic = neighbor\_heuristic

temperature \*= cooling\_rate

iteration += 1

if iteration % 1000 == 0:

print(f"Iteration {iteration}, Temperature: {temperature:.2f}, Heuristic: {current\_heuristic}")

print\_board(current\_board)

if current\_heuristic == 0:

print("Solution found!")

else:

print("Stopped without full solution. Best board found:")

print(f"Final heuristic: {current\_heuristic}")

print\_board(current\_board)

if \_\_name\_\_ == "\_\_main\_\_":

simulated\_annealing()

Enter the row positions (0 to 7) of the queens for each column:

Column 0: 4

Column 1: 6

Column 2: 1

Column 3: 5

Column 4: 2

Column 5: 0

Column 6: 3

Column 7: 7

Initial heuristic: 0

. . . . . Q . .

. . Q . . . . .

. . . . Q . . .

. . . . . . Q .

Q . . . . . . .

. . . Q . . . .

. Q . . . . . .

. . . . . . . Q

Solution found!

Final heuristic: 0

. . . . . Q . .

. . Q . . . . .

. . . . Q . . .

. . . . . . Q .

Q . . . . . . .

. . . Q . . . .

. Q . . . . . .

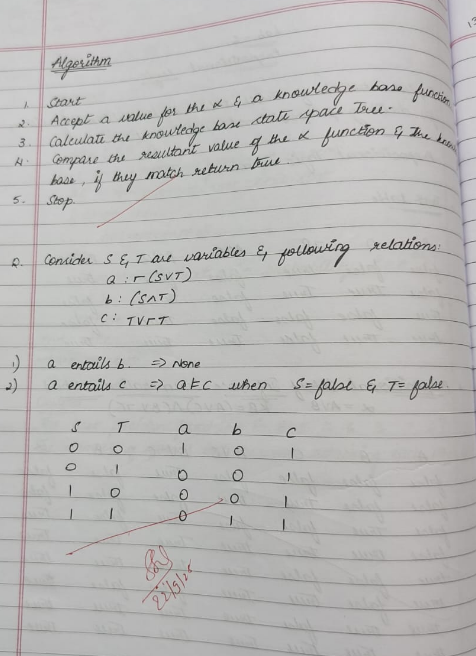
. . . . . . . Q

**Program 6:**

Create a knowledge base using propositional logic and show

that the given query entails the knowledge base or not.

Algorithm:



Code:

import itertools

import pandas as pd

import re

def replace\_implications(expr):

"""

Replace every X => Y with (not X or Y).

This uses regex with a callback to avoid partial string overwrites.

"""

*# Pattern: capture left side and right side around =>*

*# Made more flexible to handle various expressions*

pattern = r'([^=><]+?)\s\*=>\s\*([^=><]+?)(?=\s|$|[&|)])'

while re.search(pattern, expr):

expr = re.sub(pattern,

lambda m: f"(not {m.group(1).strip()} or {m.group(2).strip()})",

expr,

count=1)

return expr

def pl\_true(sentence, model):

expr = sentence.strip()

expr = expr.replace("<=>", "==")

expr = replace\_implications(expr)

*# Replace propositional symbols with their truth values safely*

for sym, val in model.items():

expr = re.sub(rf'\b{sym}\b', str(val), expr)

*# Clean up spacing and add proper spacing for boolean operators*

expr = re.sub(r'\s+', ' ', expr) *# Remove extra spaces*

expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace(" not ", " not ")

return eval(expr)

def get\_symbols(KB, alpha):

symbols = set()

for sentence in KB + [alpha]:

*# Find all alphabetic tokens (propositional variables)*

for token in re.findall(r'\b[A-Za-z]+\b', sentence):

if token not in ['and', 'or', 'not']: *# Exclude boolean operators*

symbols.add(token)

return sorted(list(symbols))

def tt\_entails(KB, alpha):

symbols = get\_symbols(KB, alpha)

rows = []

entails = True

for values in itertools.product([True, False], repeat=len(symbols)):

model = dict(zip(symbols, values))

try:

kb\_val = all(pl\_true(sentence, model) for sentence in KB)

alpha\_val = pl\_true(alpha, model)

rows.append({\*\*model, "KB": kb\_val, "alpha": alpha\_val})

if kb\_val and not alpha\_val:

entails = False

except Exception as e:

print(f"Error evaluating with model {model}: {e}")

return False

df = pd.DataFrame(rows)

*# Create a beautiful formatted table*

print("\n" + "="\*50)

print(" TRUTH TABLE")

print("="\*50)

*# Get column widths for proper alignment*

col\_widths = {}

for col in df.columns:

col\_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())

*# Calculate total table width*

table\_width = sum(col\_widths.values()) + len(df.columns) \* 3 - 1

*# Print top border*

print("┌" + "─" \* table\_width + "┐")

*# Print header*

header = "│"

for col in df.columns:

header += f" {col:^{col\_widths[col]}} │"

print(header)

*# Print separator*

separator = "├"

for col in df.columns:

separator += "─" \* (col\_widths[col] + 2) + "┼"

separator = separator[:-1] + "┤"

print(separator)

*# Print rows*

for \_, row in df.iterrows():

row\_str = "│"

for col in df.columns:

value = str(row[col])

row\_str += f" {value:^{col\_widths[col]}} │"

print(row\_str)

*# Print bottom border*

print("└" + "─" \* table\_width + "┘")

*# Print result with styling*

print("\n" + "="\*50)

result\_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"

print(f"{result\_text:^50}")

print("="\*50)

return entails

*# --- Interactive input ---*

print("Enter Knowledge Base (KB) sentences, separated by commas.")

print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")

kb\_input = input("KB: ").strip()

KB = [x.strip() for x in kb\_input.split(",")]

alpha = input("Enter query (alpha): ").strip()

result = tt\_entails(KB, alpha)

print(f"Result: {result}")

*#*

Enter Knowledge Base (KB) sentences, separated by commas.

Use symbols like A, B, C and operators: and, or, not, =>, <=>

KB: not (S or T)

Enter query (alpha): T or (not T)

==================================================

TRUTH TABLE

==================================================

┌───────────────────────────────┐

│ S │ T │ KB │ alpha │

├───────┼───────┼───────┼───────┤

│ True │ True │ False │ True │

│ True │ False │ False │ True │

│ False │ True │ False │ True │

│ False │ False │ True │ True │

└───────────────────────────────┘

==================================================

KB ENTAILS ALPHA: ✓ YES

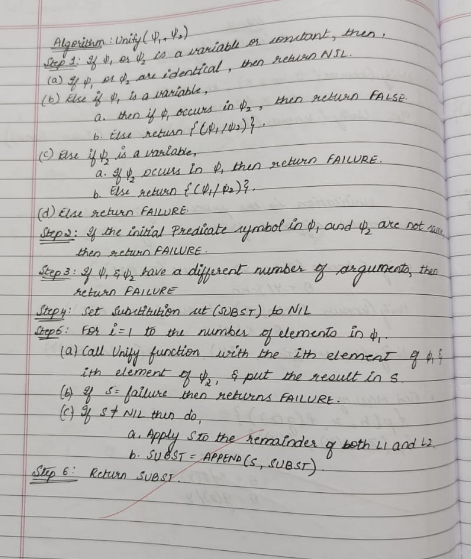
==================================================

Result: True

**Program 7**

Implement unification in first order logic

Algorithm:



Code:

class Term:

"""Base class for terms in first-order logic"""

pass

class Constant(Term):

"""Represents a constant"""

def \_\_init\_\_(self, name):

self.name = name

def \_\_eq\_\_(self, other):

return isinstance(other, Constant) and self.name == other.name

def \_\_repr\_\_(self):

return self.name

def \_\_hash\_\_(self):

return hash(('Constant', self.name))

class Variable(Term):

"""Represents a variable"""

def \_\_init\_\_(self, name):

self.name = name

def \_\_eq\_\_(self, other):

return isinstance(other, Variable) and self.name == other.name

def \_\_repr\_\_(self):

return self.name

def \_\_hash\_\_(self):

return hash(('Variable', self.name))

class Predicate(Term):

"""Represents a predicate with arguments"""

def \_\_init\_\_(self, name, args):

self.name = name

self.args = args if isinstance(args, list) else [args]

def \_\_eq\_\_(self, other):

return (isinstance(other, Predicate) and

self.name == other.name and

len(self.args) == len(other.args) and

all(a == b for a, b in zip(self.args, other.args)))

def \_\_repr\_\_(self):

return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

def occurs\_check(var, term, subst):

"""Check if variable occurs in term (prevents infinite structures)"""

if var == term:

return True

elif isinstance(term, Variable) and term in subst:

return occurs\_check(var, subst[term], subst)

elif isinstance(term, Predicate):

return any(occurs\_check(var, arg, subst) for arg in term.args)

return False

def apply\_substitution(term, subst):

"""Apply substitution to a term"""

if isinstance(term, Variable):

if term in subst:

return apply\_substitution(subst[term], subst)

return term

elif isinstance(term, Predicate):

new\_args = [apply\_substitution(arg, subst) for arg in term.args]

return Predicate(term.name, new\_args)

else:

return term

def unify(term1, term2, subst=None):

"""

Unification Algorithm

Returns substitution set if unification succeeds, None if it fails

"""

if subst is None:

subst = {}

*# Apply existing substitutions*

term1 = apply\_substitution(term1, subst)

term2 = apply\_substitution(term2, subst)

*# Step 1: If term1 or term2 is a variable or constant*

*# Step 1a: If both are identical*

if term1 == term2:

return subst

*# Step 1b: If term1 is a variable*

elif isinstance(term1, Variable):

if occurs\_check(term1, term2, subst):

return None *# FAILURE*

else:

new\_subst = subst.copy()

new\_subst[term1] = term2

return new\_subst

*# Step 1c: If term2 is a variable*

elif isinstance(term2, Variable):

if occurs\_check(term2, term1, subst):

return None *# FAILURE*

else:

new\_subst = subst.copy()

new\_subst[term2] = term1

return new\_subst

*# Step 1d: Both are constants but not equal*

elif isinstance(term1, Constant) or isinstance(term2, Constant):

return None *# FAILURE*

*# Step 2: Check if both are predicates with same name*

elif isinstance(term1, Predicate) and isinstance(term2, Predicate):

if term1.name != term2.name:

return None *# FAILURE*

*# Step 3: Check if they have same number of arguments*

if len(term1.args) != len(term2.args):

return None *# FAILURE*

*# Step 4 & 5: Unify arguments recursively*

current\_subst = subst.copy()

for arg1, arg2 in zip(term1.args, term2.args):

current\_subst = unify(arg1, arg2, current\_subst)

if current\_subst is None: *# If unification fails*

return None

return current\_subst

else:

return None *# FAILURE*

def print\_substitution(subst):

"""Pretty print substitution set"""

if subst is None:

print("FAILURE: Unification failed")

elif not subst:

print("NIL: Terms are already unified")

else:

print("Substitution:")

for var, term in subst.items():

print(f" {var} -> {term}")

def parse\_term(term\_str):

"""Parse a string representation of a term into Term objects"""

term\_str = term\_str.strip()

*# Check if it's a predicate (contains parentheses)*

if '(' in term\_str:

paren\_idx = term\_str.index('(')

pred\_name = term\_str[:paren\_idx].strip()

*# Extract arguments between parentheses*

args\_str = term\_str[paren\_idx+1:term\_str.rindex(')')].strip()

*# Split arguments by comma (handle nested predicates)*

args = []

depth = 0

current\_arg = ""

for char in args\_str:

if char == ',' and depth == 0:

args.append(parse\_term(current\_arg))

current\_arg = ""

else:

if char == '(':

depth += 1

elif char == ')':

depth -= 1

current\_arg += char

if current\_arg.strip():

args.append(parse\_term(current\_arg))

return Predicate(pred\_name, args)

*# Check if it's a variable (lowercase first letter or starts with ?)*

elif term\_str[0].islower() or term\_str[0] == '?':

return Variable(term\_str)

*# Otherwise it's a constant (uppercase first letter)*

else:

return Constant(term\_str)

def run\_interactive():

"""Interactive mode for user input"""

print("=== Unification Algorithm (Interactive Mode) ===")

print("Enter terms to unify. Use:")

print(" - Variables: lowercase letters (x, y, z) or ?x, ?y")

print(" - Constants: uppercase letters (John, Mary, A)")

print(" - Predicates: Name(arg1, arg2, ...) e.g., P(x, y)")

print(" - Type 'quit' to exit\n")

while True:

print("-" \* 50)

term1\_str = input("Enter first term: ").strip()

if term1\_str.lower() == 'quit':

print("Exiting...")

break

term2\_str = input("Enter second term: ").strip()

if term2\_str.lower() == 'quit':

print("Exiting...")

break

try:

term1 = parse\_term(term1\_str)

term2 = parse\_term(term2\_str)

print(f"\nUnifying: {term1} and {term2}")

result = unify(term1, term2)

print\_substitution(result)

print()

except Exception as e:

print(f"Error parsing terms: {e}")

print("Please check your input format.\n")

def run\_examples():

"""Run predefined examples"""

print("=== Unification Algorithm Examples ===\n")

*# Example 1: Unifying variables*

print("Example 1: Unify(x, y)")

x = Variable('x')

y = Variable('y')

result = unify(x, y)

print\_substitution(result)

print()

*# Example 2: Unifying variable with constant*

print("Example 2: Unify(x, John)")

x = Variable('x')

john = Constant('John')

result = unify(x, john)

print\_substitution(result)

print()

*# Example 3: Unifying predicates*

print("Example 3: Unify(P(x, y), P(John, z))")

p1 = Predicate('P', [Variable('x'), Variable('y')])

p2 = Predicate('P', [Constant('John'), Variable('z')])

result = unify(p1, p2)

print\_substitution(result)

print()

*# Example 4: Unifying complex predicates*

print("Example 4: Unify(P(x, f(y)), P(a, f(b)))")

p1 = Predicate('P', [Variable('x'), Predicate('f', [Variable('y')])])

p2 = Predicate('P', [Constant('a'), Predicate('f', [Constant('b')])])

result = unify(p1, p2)

print\_substitution(result)

print()

*# Example 5: Failure case - occurs check*

print("Example 5: Unify(x, f(x)) - Occurs Check")

x = Variable('x')

fx = Predicate('f', [x])

result = unify(x, fx)

print\_substitution(result)

print()

*# Example 6: Failure case - different predicates*

print("Example 6: Unify(P(x), Q(x)) - Different Predicates")

p1 = Predicate('P', [Variable('x')])

p2 = Predicate('Q', [Variable('x')])

result = unify(p1, p2)

print\_substitution(result)

print()

*# Example 7: Failure case - different constants*

print("Example 7: Unify(John, Mary) - Different Constants")

john = Constant('John')

mary = Constant('Mary')

result = unify(john, mary)

print\_substitution(result)

*# Main program*

if \_\_name\_\_ == "\_\_main\_\_":

print("Choose mode:")

print("1. Run predefined examples")

print("2. Interactive mode (enter your own terms)")

choice = input("\nEnter choice (1 or 2): ").strip()

print()

if choice == '1':

run\_examples()

elif choice == '2':

run\_interactive()

else:

print("Invalid choice. Running examples by default...\n")

run\_examples()

Choose mode:

1. Run predefined examples

2. Interactive mode (enter your own terms)

Enter choice (1 or 2): 1

=== Unification Algorithm Examples ===

Example 1: Unify(x, y)

Substitution:

x -> y

Example 2: Unify(x, John)

Substitution:

x -> John

Example 3: Unify(P(x, y), P(John, z))

Substitution:

x -> John

y -> z

Example 4: Unify(P(x, f(y)), P(a, f(b)))

Substitution:

x -> a

y -> b

Example 5: Unify(x, f(x)) - Occurs Check

FAILURE: Unification failed

Example 6: Unify(P(x), Q(x)) - Different Predicates

FAILURE: Unification failed

Example 7: Unify(John, Mary) - Different Constants

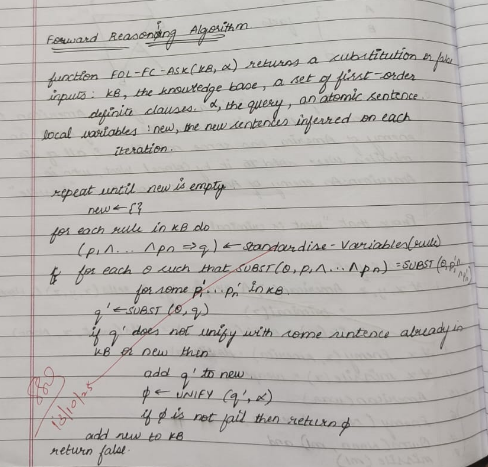
FAILURE: Unification failed

**Program 8**

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

reasoning.

Algorithm:



Code:

class Term:

"""Base class for terms in first-order logic"""

pass

class Constant(Term):

"""Represents a constant"""

def \_\_init\_\_(self, name):

self.name = name

def \_\_eq\_\_(self, other):

return isinstance(other, Constant) and self.name == other.name

def \_\_repr\_\_(self):

return self.name

def \_\_hash\_\_(self):

return hash(('Constant', self.name))

class Variable(Term):

"""Represents a variable"""

def \_\_init\_\_(self, name):

self.name = name

def \_\_eq\_\_(self, other):

return isinstance(other, Variable) and self.name == other.name

def \_\_repr\_\_(self):

return self.name

def \_\_hash\_\_(self):

return hash(('Variable', self.name))

class Predicate(Term):

"""Represents a predicate with arguments"""

def \_\_init\_\_(self, name, args):

self.name = name

self.args = args if isinstance(args, list) else [args]

def \_\_eq\_\_(self, other):

return (isinstance(other, Predicate) and

self.name == other.name and

len(self.args) == len(other.args) and

all(a == b for a, b in zip(self.args, other.args)))

def \_\_repr\_\_(self):

return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

def \_\_hash\_\_(self):

return hash((self.name, tuple(self.args)))

class Rule:

"""Represents an implication rule: premises => conclusion"""

def \_\_init\_\_(self, premises, conclusion):

self.premises = premises if isinstance(premises, list) else [premises]

self.conclusion = conclusion

def \_\_repr\_\_(self):

premises\_str = ' ∧ '.join(str(p) for p in self.premises)

return f"{premises\_str} => {self.conclusion}"

*# Variable counter for standardization*

\_var\_counter = 0

def get\_new\_variable():

"""Generate a new unique variable"""

global \_var\_counter

\_var\_counter += 1

return Variable(f"v{\_var\_counter}")

def standardize\_variables(rule):

"""Replace all variables in rule with new unique variables"""

var\_mapping = {}

def replace\_vars(term):

if isinstance(term, Variable):

if term not in var\_mapping:

var\_mapping[term] = get\_new\_variable()

return var\_mapping[term]

elif isinstance(term, Predicate):

new\_args = [replace\_vars(arg) for arg in term.args]

return Predicate(term.name, new\_args)

else:

return term

new\_premises = [replace\_vars(p) for p in rule.premises]

new\_conclusion = replace\_vars(rule.conclusion)

return Rule(new\_premises, new\_conclusion)

def occurs\_check(var, term, subst):

"""Check if variable occurs in term"""

if var == term:

return True

elif isinstance(term, Variable) and term in subst:

return occurs\_check(var, subst[term], subst)

elif isinstance(term, Predicate):

return any(occurs\_check(var, arg, subst) for arg in term.args)

return False

def apply\_substitution(term, subst):

"""Apply substitution to a term"""

if isinstance(term, Variable):

if term in subst:

return apply\_substitution(subst[term], subst)

return term

elif isinstance(term, Predicate):

new\_args = [apply\_substitution(arg, subst) for arg in term.args]

return Predicate(term.name, new\_args)

else:

return term

def unify(term1, term2, subst=None):

"""Unification algorithm"""

if subst is None:

subst = {}

term1 = apply\_substitution(term1, subst)

term2 = apply\_substitution(term2, subst)

if term1 == term2:

return subst

elif isinstance(term1, Variable):

if occurs\_check(term1, term2, subst):

return None

else:

new\_subst = subst.copy()

new\_subst[term1] = term2

return new\_subst

elif isinstance(term2, Variable):

if occurs\_check(term2, term1, subst):

return None

else:

new\_subst = subst.copy()

new\_subst[term2] = term1

return new\_subst

elif isinstance(term1, Constant) or isinstance(term2, Constant):

return None

elif isinstance(term1, Predicate) and isinstance(term2, Predicate):

if term1.name != term2.name or len(term1.args) != len(term2.args):

return None

current\_subst = subst.copy()

for arg1, arg2 in zip(term1.args, term2.args):

current\_subst = unify(arg1, arg2, current\_subst)

if current\_subst is None:

return None

return current\_subst

else:

return None

def unify\_all(premises, kb\_facts, subst=None):

"""Try to unify all premises with facts in KB"""

if subst is None:

subst = {}

if not premises:

return [subst]

first\_premise = premises[0]

remaining\_premises = premises[1:]

all\_substitutions = []

for fact in kb\_facts:

theta = unify(first\_premise, fact, subst.copy())

if theta is not None:

*# Apply substitution to remaining premises*

substituted\_remaining = [apply\_substitution(p, theta) for p in remaining\_premises]

*# Recursively unify remaining premises*

result\_substs = unify\_all(substituted\_remaining, kb\_facts, theta)

all\_substitutions.extend(result\_substs)

return all\_substitutions

def fol\_fc\_ask(kb\_facts, kb\_rules, query, max\_iterations=100):

"""

Forward Chaining Algorithm for First-Order Logic

Args:

kb\_facts: List of atomic sentences (facts) in KB

kb\_rules: List of implication rules in KB

query: The query to prove (atomic sentence)

max\_iterations: Maximum number of iterations to prevent infinite loops

Returns:

Substitution if query can be proved, None otherwise

"""

print("=== Forward Chaining Algorithm ===\n")

print(f"Query: {query}\n")

print("Initial KB Facts:")

for fact in kb\_facts:

print(f" {fact}")

print("\nKB Rules:")

for rule in kb\_rules:

print(f" {rule}")

print("\n" + "="\*50 + "\n")

iteration = 0

while iteration < max\_iterations:

iteration += 1

new = []

print(f"Iteration {iteration}:")

*# For each rule in KB*

for rule in kb\_rules:

*# Standardize variables in the rule*

std\_rule = standardize\_variables(rule)

*# Try to find substitutions that satisfy all premises*

substitutions = unify\_all(std\_rule.premises, kb\_facts)

*# For each valid substitution*

for theta in substitutions:

*# Apply substitution to conclusion*

inferred = apply\_substitution(std\_rule.conclusion, theta)

*# Check if this fact is new*

if inferred not in kb\_facts and inferred not in new:

new.append(inferred)

print(f" Inferred: {inferred}")

print(f" From rule: {std\_rule}")

print(f" With substitution: {theta}")

*# Check if inferred fact unifies with query*

result = unify(inferred, query)

if result is not None:

print(f"\n\*\*\* Query proved! \*\*\*")

print(f"Substitution: {result}")

return result

*# If no new facts inferred, we're done*

if not new:

print(" No new facts inferred.")

print("\nForward chaining completed. Query cannot be proved.")

return None

*# Add new facts to KB*

kb\_facts.extend(new)

print()

print(f"Maximum iterations ({max\_iterations}) reached.")

return None

def parse\_term(term\_str):

"""Parse a string into a Term object"""

term\_str = term\_str.strip()

if '(' in term\_str:

paren\_idx = term\_str.index('(')

pred\_name = term\_str[:paren\_idx].strip()

args\_str = term\_str[paren\_idx+1:term\_str.rindex(')')].strip()

args = []

depth = 0

current\_arg = ""

for char in args\_str:

if char == ',' and depth == 0:

args.append(parse\_term(current\_arg))

current\_arg = ""

else:

if char == '(':

depth += 1

elif char == ')':

depth -= 1

current\_arg += char

if current\_arg.strip():

args.append(parse\_term(current\_arg))

return Predicate(pred\_name, args)

elif term\_str[0].islower():

return Variable(term\_str)

else:

return Constant(term\_str)

def parse\_rule(rule\_str):

"""Parse a rule string like 'P(x) ∧ Q(x) => R(x)'"""

if '=>' in rule\_str:

parts = rule\_str.split('=>')

conclusion\_str = parts[1].strip()

premises\_str = parts[0].strip()

*# Split premises by ∧ or AND*

premise\_parts = [p.strip() for p in premises\_str.replace('AND', '∧').split('∧')]

premises = [parse\_term(p) for p in premise\_parts]

conclusion = parse\_term(conclusion\_str)

return Rule(premises, conclusion)

else:

*# It's just a fact*

return parse\_term(rule\_str)

*# Example usage*

if \_\_name\_\_ == "\_\_main\_\_":

print("Choose mode:")

print("1. Run example (Animal reasoning)")

print("2. Interactive mode")

choice = input("\nEnter choice (1 or 2): ").strip()

print()

if choice == '1':

*# Example: Animal reasoning*

*# Facts*

kb\_facts = [

Predicate('Animal', [Constant('Dog')]),

Predicate('Animal', [Constant('Cat')]),

Predicate('Loves', [Constant('John'), Constant('Dog')]),

Predicate('Owns', [Constant('John'), Constant('Dog')])

]

*# Rules*

kb\_rules = [

*# Animal(x) ∧ Loves(y, x) => Loves(x, y)*

Rule([Predicate('Animal', [Variable('x')]),

Predicate('Loves', [Variable('y'), Variable('x')])],

Predicate('Loves', [Variable('x'), Variable('y')])),

*# Owns(x, y) ∧ Animal(y) => KeepsAsPet(x, y)*

Rule([Predicate('Owns', [Variable('x'), Variable('y')]),

Predicate('Animal', [Variable('y')])],

Predicate('KeepsAsPet', [Variable('x'), Variable('y')]))

]

*# Query: Does Dog love John?*

query = Predicate('Loves', [Constant('Dog'), Constant('John')])

result = fol\_fc\_ask(kb\_facts, kb\_rules, query)

elif choice == '2':

print("=== Interactive Forward Chaining ===")

print("Enter facts and rules for the knowledge base.\n")

kb\_facts = []

kb\_rules = []

*# Input facts*

print("Enter facts (one per line, empty line to finish):")

print("Example: Animal(Dog), Loves(John, Dog)")

while True:

fact\_str = input("Fact: ").strip()

if not fact\_str:

break

try:

fact = parse\_term(fact\_str)

kb\_facts.append(fact)

except Exception as e:

print(f"Error parsing fact: {e}")

*# Input rules*

print("\nEnter rules (one per line, empty line to finish):")

print("Example: Animal(x) ∧ Loves(y,x) => Loves(x,y)")

print("You can also use 'AND' instead of ∧")

while True:

rule\_str = input("Rule: ").strip()

if not rule\_str:

break

try:

rule = parse\_rule(rule\_str)

kb\_rules.append(rule)

except Exception as e:

print(f"Error parsing rule: {e}")

*# Input query*

print("\nEnter query:")

query\_str = input("Query: ").strip()

try:

query = parse\_term(query\_str)

result = fol\_fc\_ask(kb\_facts, kb\_rules, query)

except Exception as e:

print(f"Error parsing query: {e}")

else:

print("Invalid choice.")

Choose mode:

1. Run example (Animal reasoning)

2. Interactive mode

Enter choice (1 or 2): 2

=== Interactive Forward Chaining ===

Enter facts and rules for the knowledge base.

Enter facts (one per line, empty line to finish):

Example: Animal(Dog), Loves(John, Dog)

Fact: Owns(A, T1)

Fact: Missile(T1)

Fact: American(Robert)

Fact: Enemy(A, America)

Fact:

Enter rules (one per line, empty line to finish):

Example: Animal(x) ∧ Loves(y,x) => Loves(x,y)

You can also use 'AND' instead of ∧

Rule: American(p) ∧ Weapon(q) ∧ Sells(p,q,r) ∧ Hostile(r) => Criminal(p)

Rule: Missile(x) ∧ Owns(A,x) => Sells(Robert,x,A)

Rule: Missile(x) => Weapon(x)

Rule: Enemy(x,America) => Hostile(x)

Rule:

Enter query:

Query: Criminal(Robert)

=== Forward Chaining Algorithm ===

Query: Criminal(Robert)

Initial KB Facts:

Owns(A, T1)

Missile(T1)

American(Robert)

Enemy(A, America)

KB Rules:

American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) => Criminal(p)

Missile(x) ∧ Owns(A, x) => Sells(Robert, x, A)

Missile(x) => Weapon(x)

Enemy(x, America) => Hostile(x)

==================================================

Iteration 1:

Inferred: Sells(Robert, T1, A)

From rule: Missile(v4) ∧ Owns(A, v4) => Sells(Robert, v4, A)

With substitution: {v4: T1}

Inferred: Weapon(T1)

From rule: Missile(v5) => Weapon(v5)

With substitution: {v5: T1}

Inferred: Hostile(A)

From rule: Enemy(v6, America) => Hostile(v6)

With substitution: {v6: A}

Iteration 2:

Inferred: Criminal(Robert)

From rule: American(v7) ∧ Weapon(v8) ∧ Sells(v7, v8, v9) ∧ Hostile(v9) => Criminal(v7)

With substitution: {v7: Robert, v8: T1, v9: A}

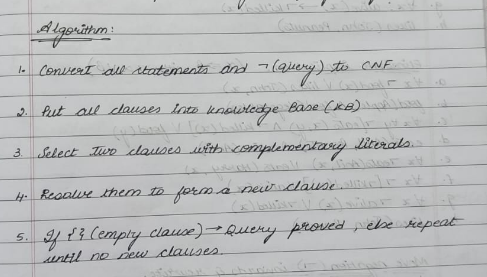
\*\*\* Query proved! \*\*\*

Substitution: {}

**Program 9**

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

Algorithm:



Code:

import re

import itertools

VAR\_RE = re.compile(r'^[a-z](\_\d+)?$') *# single-letter variable optionally standardized (x\_0, y\_3)*

def is\_variable(token: str) -> bool:

return bool(VAR\_RE.fullmatch(token))

def parse\_literal(text):

text = text.strip()

neg = False

if text.startswith('¬') or text.startswith('~'):

neg = True

text = text[1:].strip()

if '(' in text:

pred = text[:text.index('(')].strip()

args = [a.strip() for a in text[text.index('(')+1:-1].split(',')]

else:

pred = text

args = []

return {'neg': neg, 'pred': pred, 'args': args}

def clause\_to\_str(clause):

if clause == []:

return '⊥'

parts = []

for lit in clause:

s = ('¬' if lit['neg'] else '') + (lit['pred'] + '(' + ', '.join(lit['args']) + ')' if lit['args'] else lit['pred'])

parts.append(s)

return ' ∨ '.join(parts)

def standardize\_apart\_clause(clause, idx):

*# only rename variables (single-letter) to var\_index form*

mapping = {}

new\_clause = []

for lit in clause:

new\_args = []

for a in lit['args']:

if is\_variable(a):

if a not in mapping:

mapping[a] = f"{a}\_{idx}"

new\_args.append(mapping[a])

else:

new\_args.append(a)

new\_clause.append({'neg': lit['neg'], 'pred': lit['pred'], 'args': new\_args})

return new\_clause

*# ----- Unification for flat args (no nested function terms) -----*

def occurs\_check(var, val, subs):

*# var and val are token strings*

if var == val:

return True

if is\_variable(val) and val in subs:

return occurs\_check(var, subs[val], subs)

return False

def apply\_subs\_token(tok, subs):

if is\_variable(tok):

while tok in subs:

tok = subs[tok]

return tok

return tok

def apply\_subs\_literal(lit, subs):

new\_args = [apply\_subs\_token(a, subs) for a in lit['args']]

return {'neg': lit['neg'], 'pred': lit['pred'], 'args': new\_args}

def unify\_tokens(x, y, subs):

*# x,y are token strings (variables or constants)*

if x == y:

return subs

if is\_variable(x):

if x in subs:

return unify\_tokens(subs[x], y, subs)

if occurs\_check(x, y, subs):

return None

new = subs.copy()

new[x] = y

return new

if is\_variable(y):

return unify\_tokens(y, x, subs)

*# both constants and different => fail*

return None

def unify\_arg\_lists(a\_list, b\_list):

if len(a\_list) != len(b\_list):

return None

subs = {}

for a, b in zip(a\_list, b\_list):

a\_ap = a if not is\_variable(a) else a

b\_ap = b if not is\_variable(b) else b

subs = unify\_tokens(apply\_subs\_token(a\_ap, subs), apply\_subs\_token(b\_ap, subs), subs)

if subs is None:

return None

return subs

*# ----- Resolution -----*

def is\_tautology\_clause(clause):

*# clause is a list of literals (after substitution). If it contains A and ¬A same args -> tautology*

seen = {}

for lit in clause:

key = (lit['pred'], tuple(lit['args']))

if key in seen:

if seen[key] != lit['neg']:

return True

else:

seen[key] = lit['neg']

return False

def resolve\_pair(c1, c2):

*# c1, c2 are lists of literals (each literal dict)*

for i, l1 in enumerate(c1):

for j, l2 in enumerate(c2):

if l1['pred'] == l2['pred'] and l1['neg'] != l2['neg']:

*# try to unify their args*

subs = unify\_arg\_lists(l1['args'], l2['args'])

if subs is None:

continue

*# apply substitution to the remainder of both clauses*

new\_clause = []

for k, lit in enumerate(c1):

if k == i: continue

new\_clause.append(apply\_subs\_literal(lit, subs))

for k, lit in enumerate(c2):

if k == j: continue

new\_clause.append(apply\_subs\_literal(lit, subs))

*# remove duplicates (syntactic)*

uniq = []

for lit in new\_clause:

if not any(lit['pred']==u['pred'] and lit['neg']==u['neg'] and lit['args']==u['args'] for u in uniq):

uniq.append(lit)

if is\_tautology\_clause(uniq):

continue

return uniq, subs, (i, j)

return None, None, None

*# ----- Build derivation tree nodes -----*

class Node:

def \_\_init\_\_(self, clause, parents=None, label=None):

self.clause = clause

self.parents = parents if parents else []

self.label = label

def resolution\_with\_tree(initial\_clauses, goal\_clause):

*# standardize apart initial clauses*

clauses\_nodes = []

for idx, c in enumerate(initial\_clauses):

std = standardize\_apart\_clause(c, idx)

clauses\_nodes.append(Node(std, parents=[] , label=f"C{idx}"))

*# add negated goal as a fresh clause (standardize apart too)*

neg\_goal = []

*# goal\_clause is a clause list (we take its first literal if single-literal goal)*

for lit in goal\_clause:

*# negate each literal in goal clause (if goal is a single positive literal user passed)*

neg\_goal.append({'neg': not lit['neg'], 'pred': lit['pred'], 'args': lit['args'][:]})

neg\_goal\_std = standardize\_apart\_clause(neg\_goal, len(clauses\_nodes))

goal\_node = Node(neg\_goal\_std, parents=[], label="¬Goal")

clauses\_nodes.append(goal\_node)

*# mapping from index -> Node*

idx = len(clauses\_nodes)

seen\_clauses = {clause\_to\_str(n.clause): i for i, n in enumerate(clauses\_nodes)}

*# perform breadth-first-ish resolution (pairwise), record parents as indices*

for a\_index in range(len(clauses\_nodes)):

pass *# placeholder, we'll use dynamic loop below*

frontier\_changed = True

while True:

new\_added = False

*# iterate pairs over current clauses*

n = len(clauses\_nodes)

pairs = [(i,j) for i in range(n) for j in range(i+1, n)]

for i,j in pairs:

c1 = clauses\_nodes[i].clause

c2 = clauses\_nodes[j].clause

resolvent, subs, which = resolve\_pair(c1, c2)

if resolvent is None:

continue

s = clause\_to\_str(resolvent)

if s in seen\_clauses:

continue

*# add node*

new\_node = Node(resolvent, parents=[i, j], label=f"R{idx}")

clauses\_nodes.append(new\_node)

seen\_clauses[s] = idx

new\_added = True

idx += 1

if resolvent == []:

*# build bottom-up tree node for ⊥*

root = new\_node

return clauses\_nodes, seen\_clauses, idx-1 *# return nodes, map, index of empty clause node*

if not new\_added:

return clauses\_nodes, seen\_clauses, None

*# ----- ASCII print bottom-up (root bottom) -----*

def print\_bottom\_up\_tree(nodes, root\_index):

*# recursively print node; ensure parents printed above*

def recurse(node\_index, prefix="", is\_last=True):

node = nodes[node\_index]

connector = "└── " if is\_last else "├── "

print(prefix + connector + clause\_to\_str(node.clause))

*# if this node has parents, print them above (parents as children in recursion so they appear above)*

parents = node.parents

for k, pidx in enumerate(parents):

recurse(pidx, prefix + (" " if is\_last else "│ "), k == len(parents)-1)

recurse(root\_index, "", True)

*# ----- Runner -----*

if \_\_name\_\_ == "\_\_main\_\_":

print("="\*70)

print("FIRST-ORDER LOGIC RESOLUTION SYSTEM (FIXED)")

print("="\*70)

print("Enter CNF clauses (one per line). End with a blank line.")

raw = []

while True:

try:

line = input().strip()

except EOFError:

break

if line == "":

break

raw.append(line)

clauses = [ [parse\_literal(tok.strip()) for tok in re.split(r"∨", line) ] for line in raw ]

*# read goal*

goal\_line = input("\nEnter GOAL clause (single literal form): ").strip()

goal\_clause = [parse\_literal(goal\_line)]

nodes, seen\_map, root\_idx = resolution\_with\_tree(clauses, goal\_clause)

if root\_idx is None:

print("\nNo empty clause could be derived — goal not entailed by KB.")

else:

print("\nDERIVATION TREE (bottom-up):")

print\_bottom\_up\_tree(nodes, root\_idx)

print("\nResolution complete — ⊥ derived.")

======================================================================

FIRST-ORDER LOGIC RESOLUTION SYSTEM

======================================================================

Enter CNF clauses (one per line). End with a blank line.

¬human(x) ∨ mortal(x)

human(Socrates)

Enter GOAL clause (single literal form): mortal(Socrates)

DERIVATION TREE (bottom-up):

└── ⊥

├── human(Socrates)

└── ¬human(Socrates)

├── ¬human(x\_0) ∨ mortal(x\_0)

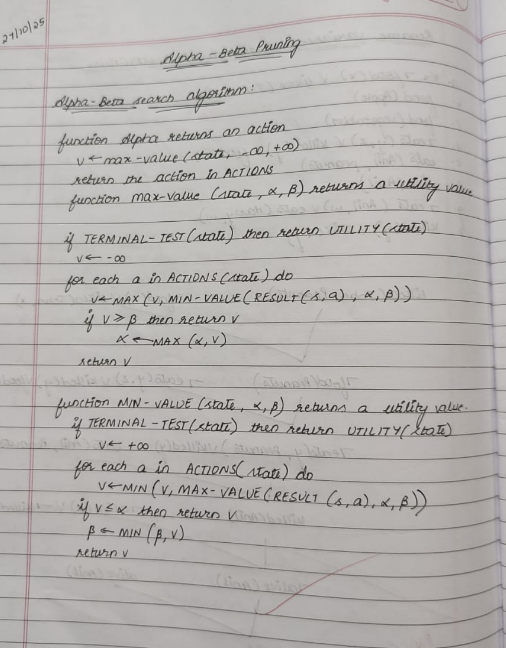
└── ¬mortal(Socrates)

Resolution complete — ⊥ derived.

**Program 10**

Implement Alpha-Beta Pruning.

Algorithm:



Code:

import networkx as nx

import matplotlib.pyplot as plt

import math

*# --- Alpha-Beta Pruning ---*

def alpha\_beta(node, depth, alpha, beta, maximizing, tree, values, pruned\_nodes, path):

*# Leaf node*

if depth == 0 or node not in tree:

return values.get(node, None)

if maximizing:

value = -math.inf

for child in tree[node]:

val = alpha\_beta(child, depth - 1, alpha, beta, False, tree, values, pruned\_nodes, path)

if val is None:

continue

value = max(value, val)

alpha = max(alpha, value)

if beta <= alpha:

*# Prune remaining children*

prune\_index = tree[node].index(child) + 1

for c in tree[node][prune\_index:]:

pruned\_nodes.append(c)

break

values[node] = value

return value

else:

value = math.inf

for child in tree[node]:

val = alpha\_beta(child, depth - 1, alpha, beta, True, tree, values, pruned\_nodes, path)

if val is None:

continue

value = min(value, val)

beta = min(beta, value)

if beta <= alpha:

prune\_index = tree[node].index(child) + 1

for c in tree[node][prune\_index:]:

pruned\_nodes.append(c)

break

values[node] = value

return value

*# --- Draw Game Tree ---*

def draw\_game\_tree(G, path, pruned):

pos = nx.nx\_agraph.graphviz\_layout(G, prog="dot")

plt.figure(figsize=(9, 6))

edge\_colors = []

for (u, v) in G.edges():

if u in path and v in path:

edge\_colors.append('green')

elif v in pruned:

edge\_colors.append('red')

else:

edge\_colors.append('black')

node\_colors = []

for node in G.nodes():

if node in path:

node\_colors.append('green')

elif node in pruned:

node\_colors.append('red')

else:

node\_colors.append('skyblue')

nx.draw(

G, pos, with\_labels=True,

node\_color=node\_colors,

edge\_color=edge\_colors,

node\_size=1200,

font\_size=10

)

plt.title("Alpha-Beta Pruning Game Tree\nGreen = Optimal Path | Red = Pruned Nodes")

plt.show()

*# --- Main Program ---*

def main():

tree = {}

G = nx.DiGraph()

n = int(input("Enter number of non-leaf nodes: "))

for \_ in range(n):

parent = input("\nEnter parent node: ").strip()

children = input("Enter children of " + parent + " (space separated): ").split()

tree[parent] = children

for c in children:

G.add\_edge(parent, c)

leaf\_count = int(input("\nEnter number of leaf nodes: "))

values = {}

for \_ in range(leaf\_count):

leaf, val = input("Enter leaf node and its value (e.g. E 3): ").split()

values[leaf] = int(val)

root = input("\nEnter root node: ").strip()

depth = int(input("Enter total depth of tree: "))

pruned\_nodes = []

path = []

print("\n--------------------------------")

result = alpha\_beta(root, depth, -math.inf, math.inf, True, tree, values, pruned\_nodes, path)

print(f"Final Optimal Value: {result}")

print(f"Pruned Nodes: {pruned\_nodes}")

print("--------------------------------")

draw\_game\_tree(G, path=[root, 'C', 'G'], pruned=pruned\_nodes)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Enter number of non-leaf nodes: 4

Enter parent node: A

Enter children of A (space separated): B C D

Enter parent node: B

Enter children of B (space separated): E F

Enter parent node: C

Enter children of C (space separated): G H

Enter parent node: D

Enter children of D (space separated): I J

Enter number of leaf nodes: 6

Enter leaf node and its value (e.g. E 3): E 3

Enter leaf node and its value (e.g. E 3): F 5

Enter leaf node and its value (e.g. E 3): G 6

Enter leaf node and its value (e.g. E 3): H 9

Enter leaf node and its value (e.g. E 3): I 1

Enter leaf node and its value (e.g. E 3): J 2

Enter root node: A

Enter total depth of tree: 3

--------------------------------

Final Optimal Value: 6

Pruned Nodes: ['J']

--------------------------------

