

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on
Artificial Intelligence (23CS5PCAIN)

Submitted by

Afreeen Anz(1BM23CS016)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Afreen Anz (1BM23CS016)**, 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. Swati Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	---

Index

Sl. No.	Date	Experiment Title	Page No.
1	20-08-2025	Implement Tic – Tac – Toe Game Implement vacuum cleaner agent	6
2	03-09-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	16
3	10-09-2025	Implement A* search algorithm	27
4	08-10-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	32
5	08-10-2025	Simulated Annealing to Solve 8-Queens problem	38
6	15-10-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	41
7	29-10-2025	Implement unification in first order logic	44
8	12-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	49
9	12-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	53
10	12-11-2025	Implement Alpha-Beta Pruning.	59

Github Link:

<https://github.com/1BM23CS047/AI-lab>



CERTIFICATE OF ACHIEVEMENT

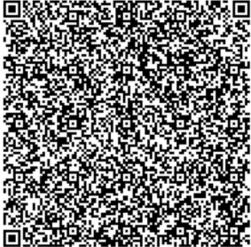
The certificate is awarded to

Afreen Anz

for successfully completing

Artificial Intelligence Foundation Certification

on November 22, 2025



Issued on: Saturday, November 22, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>



Congratulations! You make us proud!

Satheesh B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited



COURSE COMPLETION CERTIFICATE

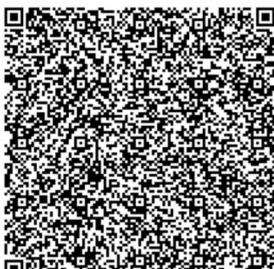
The certificate is awarded to

Afreen Anz

for successfully completing the course

Introduction to Natural Language Processing

on November 22, 2025



Issued on: Monday, November 24, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>



Congratulations! You make us proud!

Satheesh B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited



Navigate your next

||||| COURSE COMPLETION CERTIFICATE |||||

The certificate is awarded to

Afreen Anz

for successfully completing the course

Introduction to Deep Learning

on November 22, 2025



Issued on: Monday, November 24, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>



Congratulations! You make us proud!

Sathesha B.N.
Sathesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited



More actions

Navigate your next

||||| COURSE COMPLETION CERTIFICATE |||||

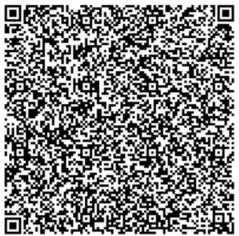
The certificate is awarded to

Afreen Anz

for successfully completing the course

Introduction to Artificial Intelligence

on November 22, 2025



Issued on: Monday, November 24, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>



Congratulations! You make us proud!

Sathesha B.N.
Sathesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Program 1

Implement Tic - Tac - Toe Game

Implement vacuum cleaner agent

Tic-Tac-Toe

Algorithm

Bafna Gold
Date: _____
Page: 1

Tic - Tac - Toe Game
Week Lab - 1

Pseudocode: Algorithm.

- start
- initialize 3x3 board with spaces that are empty
- Let computer play 'X' 'O'
- Let user play 'X'
- Let ^{Users} and computer choose from row (1 - 3) and column number (1 - 3)
- When the chosen cell is free the user can enter 'X'
- If the chosen cell is not free the system should display error and the user should have to choose another cell that is free to display enter 'X'
- After that Dis every move display the board
- If the user wins the system should display 'you win'
- If the rows and columns are full then the system should display 'draw'

For the Computer:-

Min Max Algorithm

- call bestMove() func
- best move (board) is initialized
- computer must always play in the most optimal manner
- Initialize best score = ∞
- for each cell place '-' showing an empty row line

- call minmax(board, depth, ismaximizing = false)
- undo move
- If $\text{score} > \text{best_score}$
let best_score be best score
- minmax is recursively called to consider all moves to find the move of best score, that move is made by computer
- return the best move
- place 'O' in the best move cell
- if computer wins → print "computer win"
- if board is full print draw
- (5) minmax(board, depth, ismaximizing)
 - If ~~X~~ "O" wins return 1
 - If "X" wins return -1
 - If board is full return 0
 - If ismaximizing = true
 - Let bestscore = -∞
 - For each empty cell place "O"
 - recursively call minmax with ismaximizing = false
 - undo move
 - update best cell = $\max(\text{best_score}, \text{score})$
 - return the best move best score
 - else user's turn

place X on every possible empty space

initialize best_score = +∞

call minmax

ismaximizing = true

call minmax(board, depth, ismaximizing)

undo move

}
 if score > best-score
 update
 return best-score

- minmax is recursively called to consider all moves
- return the best move

- stop

Code

```
import math

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board, player):
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def minimax(board, depth, is_maximizing, ai, human):
    if check_winner(board, ai):
        return 1
    if check_winner(board, human):
        return -1
    if is_full(board):
```

```

return 0

if is_maximizing:
    best = -math.inf
    for r in range(3):
        for c in range(3):
            if board[r][c] == " ":
                board[r][c] = ai
                score = minimax(board, depth + 1, False, ai, human)
                board[r][c] = " "
                best = max(best, score)
    return best
else:
    best = math.inf
    for r in range(3):
        for c in range(3):
            if board[r][c] == " ":
                board[r][c] = human
                score = minimax(board, depth + 1, True, ai, human)
                board[r][c] = " "
                best = min(best, score)
    return best

def best_move(board, ai, human):
    best_score = -math.inf
    move = None
    for r in range(3):
        for c in range(3):
            if board[r][c] == " ":
                board[r][c] = ai
                score = minimax(board, 0, False, ai, human)
                board[r][c] = " "
                if score > best_score:
                    best_score = score
                    move = (r, c)
    return move

def tic_tac_toe():
    board = [[ " " for _ in range(3)] for _ in range(3)]
    human, ai = "X", "O"

    print("Welcome to Tic-Tac-Toe!")
    print("You are X, Computer is O (Unbeatable AI)")
    print_board(board)

    while True:
        # Human move

```

```

try:
    row = int(input("Enter row (0-2): "))
    col = int(input("Enter col (0-2): "))
except ValueError:
    print("Invalid input. Try again.")
    continue

if row not in [0,1,2] or col not in [0,1,2] or board[row][col] != " ":
    print("Invalid move! Try again.")
    continue

board[row][col] = human
print_board(board)

if check_winner(board, human):
    print("🎉 You win!")
    break
if is_full(board):
    print("It's a draw!")
    break

# Computer move
print("Computer's turn...")
row, col = best_move(board, ai, human)
board[row][col] = ai
print_board(board)

if check_winner(board, ai):
    print("💻 Computer wins!")
    break
if is_full(board):
    print("It's a draw!")
    break

if __name__ == "__main__":
    tic_tac_toe()

```

Output

```
Welcome to Tic-Tac-Toe!
You are X, Computer is O (Unbeatable AI)
```

```
| |
```

```
-----
```

```
| |
```

```
-----
```

```
| |
```

```
-----
```

```
Enter row (0-2): 1
```

```
Enter col (0-2): 2
```

```
| |
```

```
-----
```

```
| | X
```

```
-----
```

```
| |
```

```
-----
```

```
Computer's turn...
```

```
| | 0
```

```
-----
```

```
| | X
```

```
-----
```

```
| |
```

```
-----
```

```
Enter row (0-2): 1
```

```
Enter col (0-2): 0
```

```
| | 0
```

```
-----
```

```
X | | X
```

```
-----
```

```
| |
```

```
-----
```

```
Computer's turn...
```

```
| | 0
```

```
-----
```

```
X | 0 | X
```

```
-----
```

```
| |
```

```
-----
```

```
Enter row (0-2): 0
Enter col (0-2): 0
X |   | 0
-----
X | 0 | X
-----
|   |
-----
Computer's turn...
X |   | 0
-----
X | 0 | X
-----
0 |   |
-----
💻 Computer wins!
```

Vacuum Cleaner

Algorithm

Vaccum Cleaner Algorithm

```
Function vacuum_cleaner (room-array) :  
    for each room in room-array:  
        if room is duty:  
            print "Room [label] is cleaned"  
        else  
            print "Room [label] is already clean"  
    More to next room
```

```
def vacuum_cleaner (room-array):  
    for room in room array:  
        label, status = room[0], room[1].lower()  
        print(f"Current room {label}, room[3].lower")  
        print(f"Currently in room {label}")  
        if status == "dirty":  
            print(f"Room {label} is cleaned")  
        else:  
            print(f"Room {label} is already  
            cleaned")
```

```
rooms = [('A', 'duty'), ('B', 'clean'), ('C', 'clean'),  
         ('D', 'duty'), ('A', 'clean'))  
vacuum_cleaner (rooms)
```

Code

```
def vacuum_cleaner(room_array):  
    for room in room_array:  
        label, status = room[0], room[1].lower()  
        print(f"Currently in Room {label}")  
        if status == "dirty":  
            print(f"Room {label} is cleaned")  
        else:
```

```
    print(f"Room {label} is already clean")
    print("Moving to next room...\n")

# Example input: list of rooms with their status
rooms = [('A', 'dirty'), ('B', 'clean'), ('A', 'clean'), ('B', 'dirty')]
vacuum_cleaner(rooms)
```

Output

```
Currently in Room A
Room A is cleaned
Moving to next room...

Currently in Room B
Room B is already clean
Moving to next room...

Currently in Room A
Room A is already clean
Moving to next room...

Currently in Room B
Room B is cleaned
Moving to next room...
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

8 Puzzle Problem

Algorithm

Week Lab 2

- Q) 8 puzzle problem using misplaced tiles and Manhattan distance and IDDFS

Initialize 8 puzzle problem using misplaced tiles and Manhattan distance

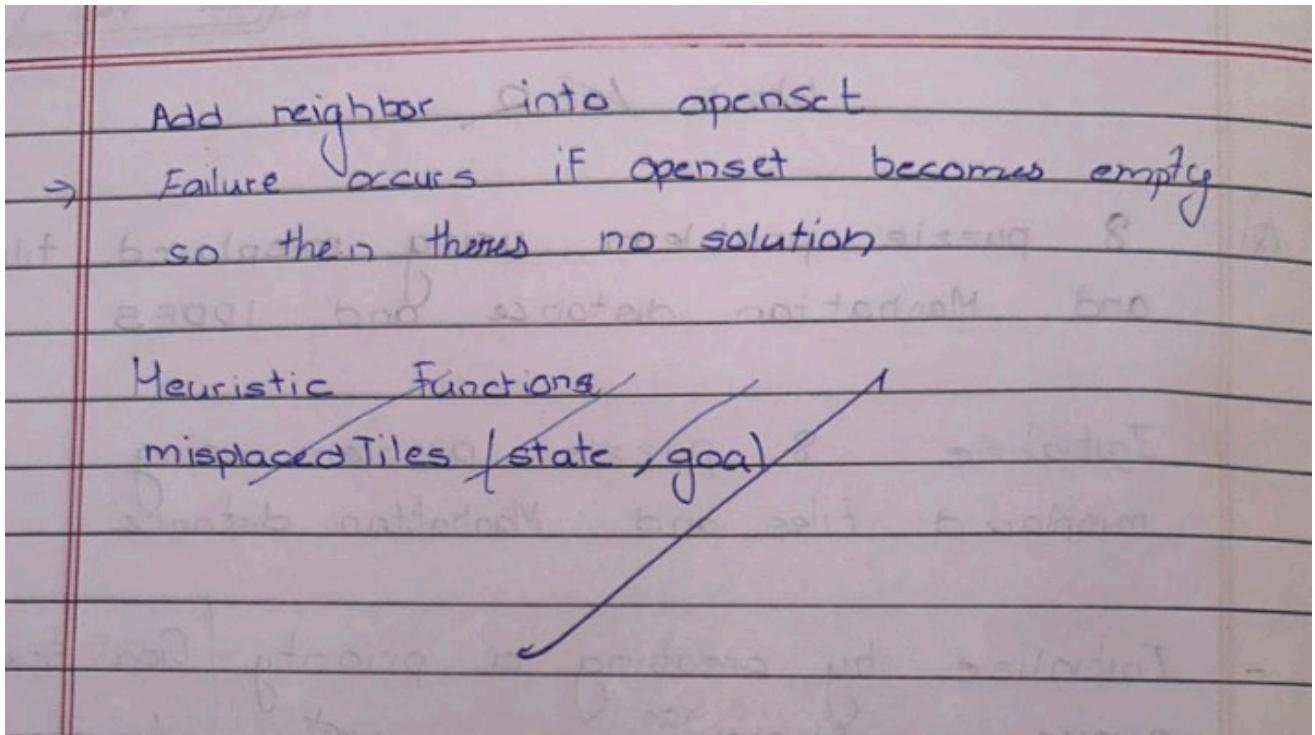
- Initialize by creating a priority queue
- $F(n) = g(n) + h(n)$
- Set $g(\text{start}) = 0$
- Set $F(\text{start}) = \text{heuristic}(\text{start}, \text{goal})$
- Store parent [start] = null
- Search loop
- insert start into openset
- Search loop while openset is not empty
- Remove the current with lowest F from openset

If current = goal :

Return path by reconstructing from parent
else expand current for each valid move of
the blank tile.

- Generate neighbour with next move
tentative $g = g(\text{current}) + 1$
- If neighbour is not in gscore or tentative $g < g(\text{neighbour})$:
- Update $g(\text{neighbour}) = \text{tentative } g$
- Compute $F(\text{neighbor}) = g(\text{neighbor}) + \text{heuristic}(\text{neighbor}, \text{goal})$

Set parent [neighbor] = current



Code

```

import time

def find_possible_moves(state):
    index = state.index('_')
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [6, 8, 4],
        8: [5, 7],
    }
    return moves.get(index, [])
}

def dfs(initial_state, goal_state, max_depth=50):
    stack = [(initial_state, [], 0)]
    visited = {tuple(initial_state)}
    states_explored = 0
    printed_depths = set()

    while stack:
        state, path, depth = stack.pop()
        if state == goal_state:
            print(f"Goal state found at depth {depth}!")
            break
        if depth > max_depth:
            continue
        moves = find_possible_moves(state)
        for move in moves:
            new_state = state[:move] + state[move+1]
            if tuple(new_state) not in visited:
                stack.append((new_state, path + [move], depth + 1))
                visited.add(tuple(new_state))
        states_explored += 1
        if states_explored % 1000000 == 0:
            print(f"States explored: {states_explored}")

```

```

current_state, path, depth = stack.pop()

if depth > max_depth:
    continue

if depth not in printed_depths:
    print(f"\n--- Depth {depth} ---")
    printed_depths.add(depth)

states_explored += 1
print(f"State # {states_explored}: {current_state}")

if current_state == goal_state:
    print(f"\n Goal reached at depth {depth} after exploring {states_explored} states.\n")
    return path, states_explored

possible_moves_indices = find_possible_moves(current_state)

for move_index in reversed(possible_moves_indices): # Reverse for DFS order
    next_state = list(current_state)
    blank_index = next_state.index('_')
    next_state[blank_index], next_state[move_index] = next_state[move_index],
    next_state[blank_index]

    if tuple(next_state) not in visited:
        visited.add(tuple(next_state))
        stack.append((next_state, path + [next_state], depth + 1))

print(f"\n Goal state not reachable within depth {max_depth}. Explored {states_explored} states.\n")
return None, states_explored

# ----- TEST -----
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]

goal_state = [1, 2, 3,
              4, 5, 6,
              7, 8, '_']

# Measure execution time
start_time = time.time()
solution_path, explored = dfs(initial_state, goal_state, max_depth=50)
end_time = time.time()

```

```

if solution_path is None:
    print("No solution found.")
else:
    print("Solution path:")
    for step, state in enumerate(solution_path, start=1):
        print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end_time - start_time))
print("Total states explored:", explored)

```

Output

```

--- Depth 0 ---
State #1: [1, 2, 3, 4, 8, '_', 7, 6, 5]

--- Depth 1 ---
State #2: [1, 2, '_', 4, 8, 3, 7, 6, 5]

--- Depth 2 ---
State #3: [1, '_', 2, 4, 8, 3, 7, 6, 5]

--- Depth 3 ---
State #4: ['_', 1, 2, 4, 8, 3, 7, 6, 5]

--- Depth 4 ---
State #5: [4, 1, 2, '_', 8, 3, 7, 6, 5]

--- Depth 5 ---
State #6: [4, 1, 2, 8, '_', 3, 7, 6, 5]

--- Depth 6 ---
State #7: [4, '_', 2, 8, 1, 3, 7, 6, 5]

--- Depth 7 ---
State #8: ['_', 4, 2, 8, 1, 3, 7, 6, 5]

--- Depth 8 ---
State #9: [8, 4, 2, '_', 1, 3, 7, 6, 5]

```

```
--- Depth 9 ---
State #10: [8, 4, 2, 1, '_', 3, 7, 6, 5]

--- Depth 10 ---
State #11: [8, '_', 2, 1, 4, 3, 7, 6, 5]

--- Depth 11 ---
State #12: ['_', 8, 2, 1, 4, 3, 7, 6, 5]

--- Depth 12 ---
State #13: [1, 8, 2, '_', 4, 3, 7, 6, 5]

--- Depth 13 ---
State #14: [1, 8, 2, 7, 4, 3, '_', 6, 5]

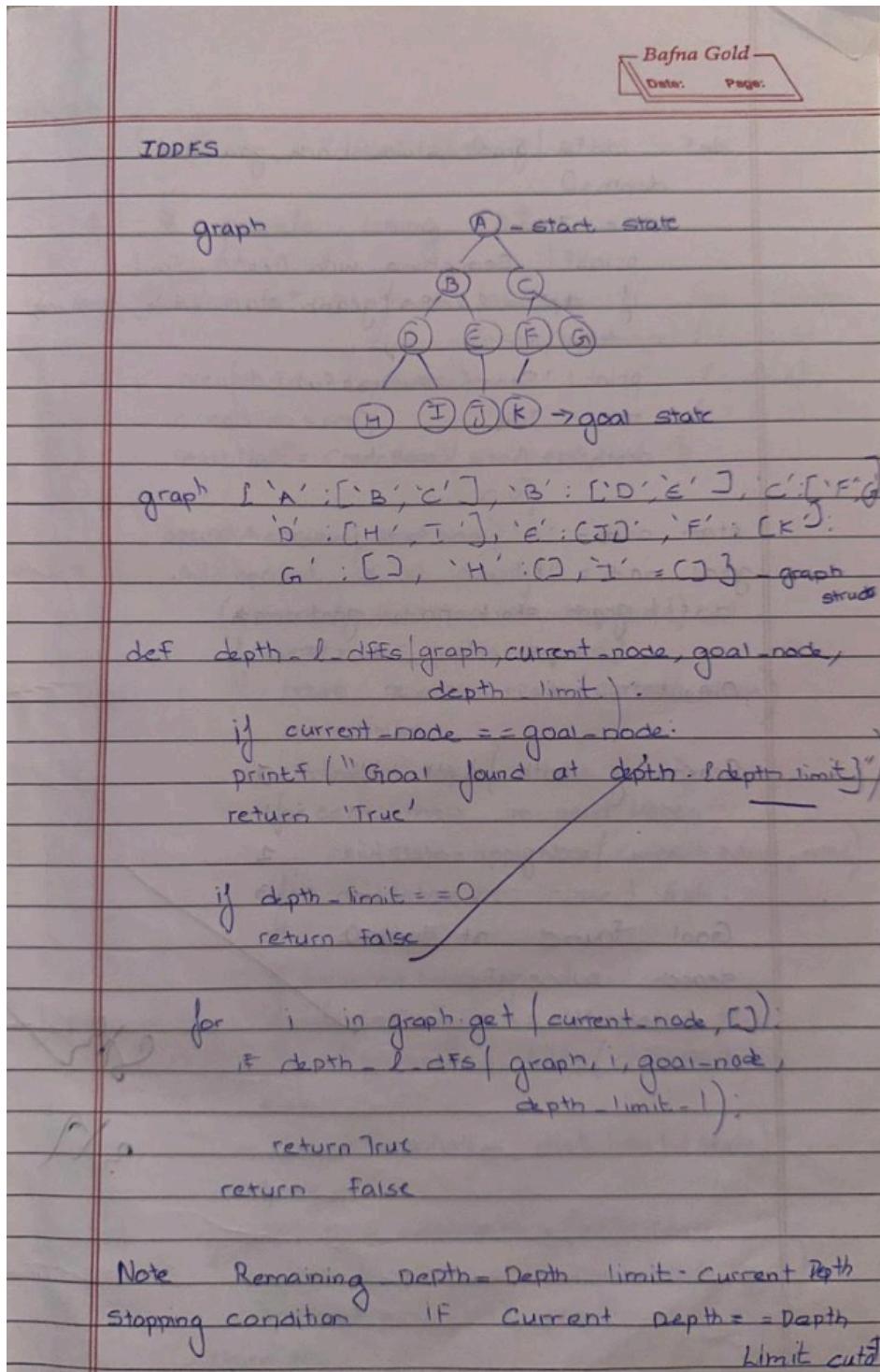
--- Depth 14 ---
State #15: [1, 8, 2, 7, 4, 3, 6, '_', 5]

--- Depth 15 ---
State #16: [1, 8, 2, 7, 4, 3, 6, 5, '_']

--- Depth 16 ---
State #17: [1, 8, 2, 7, 4, '_', 6, 5, 3]
```

IDDFS

Algorithm



```

def idDFS (graph, start_node, goal_node)
    depth = 0
    while True:
        printf (" Searching with Depth limit
if depth - l - DFS (graph, start_node, goal_node,
depth)
print (" Search successful ! ")
return True
depth += 1

start_node = 'A'
goal_node = 'K'
idDFS (graph, start_node, goal_node)

```

Searching with depth limit : 0

Goal found at depth 0
Search successful

Code

import time

```

def find_possible_moves(state):
    index = state.index('_')

    if index == 0:
        return [1, 3]
    elif index == 1:
        return [0, 2, 4]
    elif index == 2:
        return [1, 5]
    elif index == 3:
        return [0, 4, 6]
    elif index == 4:
        return [1, 3, 5, 7]
    elif index == 5:
        return [2, 4, 8]
    elif index == 6:
        return [3, 7]
    elif index == 7:
        return [4, 6, 8]
    elif index == 8:
        return [5, 7]
    return []

def depth_limited_dfs(state, goal_state, limit, path, visited):
    if state == goal_state:
        return path

    if limit <= 0:
        return None

    visited.add(tuple(state))

    for move_index in find_possible_moves(state):
        next_state = list(state)
        blank_index = next_state.index('_')
        next_state[blank_index], next_state[move_index] = next_state[move_index], next_state[blank_index]

        if tuple(next_state) not in visited:
            result = depth_limited_dfs(next_state, goal_state, limit - 1, path + [next_state], visited)
            if result is not None:
                return result
    return None

def iddfs(initial_state, goal_state, max_depth=30):
    for depth in range(max_depth):
        print(f"Searching at depth limit = {depth}")

```

```

visited = set()
result = depth_limited_dfs(initial_state, goal_state, depth, [initial_state], visited)
if result is not None:
    return result, depth
return None, max_depth

# ----- TEST -----
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]

goal_state = [1, 2, 3,
              4, 5, 6,
              7, 8, '_']

# Measure execution time
start_time = time.time()
solution_path, depth_reached = iddfs(initial_state, goal_state, max_depth=30)
end_time = time.time()

if solution_path is None:
    print("Goal state is not reachable within given depth limit.")
else:
    print("\n\nSolution path found:")
    for step, state in enumerate(solution_path, start=0):
        print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end_time - start_time))
print("Depth reached:", depth_reached)

```

Output

```
Searching at depth limit = 0
Searching at depth limit = 1
Searching at depth limit = 2
Searching at depth limit = 3
Searching at depth limit = 4
Searching at depth limit = 5

Solution path found:
Step 0: [1, 2, 3, 4, 8, '_', 7, 6, 5]
Step 1: [1, 2, 3, 4, 8, 5, 7, 6, '_']
Step 2: [1, 2, 3, 4, 8, 5, 7, '_', 6]
Step 3: [1, 2, 3, 4, '_', 5, 7, 8, 6]
Step 4: [1, 2, 3, 4, 5, '_', 7, 8, 6]
Step 5: [1, 2, 3, 4, 5, 6, 7, 8, '_']

Execution time: 0.000194 seconds
Depth reached: 5

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

Program 3

Implement A* search algorithm

Algorithm

Bajna Gold
Date: _____ Page: _____

10/9/25 Week 3 assignment

Q 8 puzzle using A* search algorithm

```

function a* astarSearch (startState, goalState, heuristicType)
    openList = priorityQueue()
    closedList = set()
    startNode = CreateBoard (startState, null, 0, 0)
    openList.enqueue (startNode)
    while openList is not empty:
        currentNode = openList.dequeue()
        if currentNode == openList.dequeue():
            return reconstructPath (currentNode)
        closedList.add (currentNode)
        for each move in valid moves:
            childState = applyMove (currentNode, move)
            if childState in closed list:
                continue
            g = currentNode.g + 1
            if heuristicType == 'Misplaced Tiles':
                h = misplacedTiles (childState)
            else:
                h = manhattan_dist (childState)
            f = g + h
            childNode = createNode (childState, currentNode, g, f)
            openList.push (childNode)
    return NULL
  
```

function reconstructPath(node):

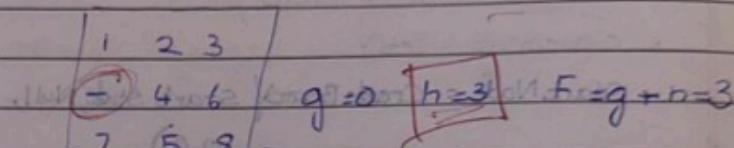
path = []

while node is not null:

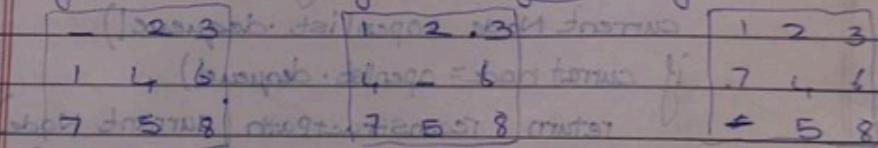
path.append(node)

node = node.parent

return reverse(path)



g=1, h=4, f=5; g=1, h=2, f=3; g=1, h=1, f=5



(child 1) 1 2 3
1 0 2 3 4

(child 2) 1 2 3
7 5 8

(child 3) 1 2 3
7 5 8

g=2, h=1, f=3; g=2, h=3, f=5; g=2, h=3, f=5

(child 1) 1 2 3
4 0 5 6

(child 2) 1 2 3
4 5 6

(child 3) 1 2 3
7 8 -

(child 4) 1 2 3
7 8 -

(child 5) 1 2 3
7 8 -

(child 6) 1 2 3
7 8 -

(child 7) 1 2 3
7 8 -

(child 8) 1 2 3
7 8 -

(child 9) 1 2 3
7 8 -

(child 10) 1 2 3
7 8 -

(child 11) 1 2 3
7 8 -

(child 12) 1 2 3
7 8 -

(child 13) 1 2 3
7 8 -

(child 14) 1 2 3
7 8 -

(child 15) 1 2 3
7 8 -

(child 16) 1 2 3
7 8 -

(child 17) 1 2 3
7 8 -

(child 18) 1 2 3
7 8 -

(child 19) 1 2 3
7 8 -

(child 20) 1 2 3
7 8 -

(child 21) 1 2 3
7 8 -

(child 22) 1 2 3
7 8 -

(child 23) 1 2 3
7 8 -

(child 24) 1 2 3
7 8 -

(child 25) 1 2 3
7 8 -

(child 26) 1 2 3
7 8 -

(child 27) 1 2 3
7 8 -

(child 28) 1 2 3
7 8 -

(child 29) 1 2 3
7 8 -

(child 30) 1 2 3
7 8 -

(child 31) 1 2 3
7 8 -

(child 32) 1 2 3
7 8 -

(child 33) 1 2 3
7 8 -

(child 34) 1 2 3
7 8 -

(child 35) 1 2 3
7 8 -

(child 36) 1 2 3
7 8 -

(child 37) 1 2 3
7 8 -

(child 38) 1 2 3
7 8 -

(child 39) 1 2 3
7 8 -

(child 40) 1 2 3
7 8 -

(child 41) 1 2 3
7 8 -

(child 42) 1 2 3
7 8 -

(child 43) 1 2 3
7 8 -

(child 44) 1 2 3
7 8 -

(child 45) 1 2 3
7 8 -

(child 46) 1 2 3
7 8 -

(child 47) 1 2 3
7 8 -

(child 48) 1 2 3
7 8 -

(child 49) 1 2 3
7 8 -

(child 50) 1 2 3
7 8 -

(child 51) 1 2 3
7 8 -

(child 52) 1 2 3
7 8 -

(child 53) 1 2 3
7 8 -

(child 54) 1 2 3
7 8 -

(child 55) 1 2 3
7 8 -

(child 56) 1 2 3
7 8 -

(child 57) 1 2 3
7 8 -

(child 58) 1 2 3
7 8 -

(child 59) 1 2 3
7 8 -

(child 60) 1 2 3
7 8 -

(child 61) 1 2 3
7 8 -

(child 62) 1 2 3
7 8 -

(child 63) 1 2 3
7 8 -

(child 64) 1 2 3
7 8 -

(child 65) 1 2 3
7 8 -

(child 66) 1 2 3
7 8 -

(child 67) 1 2 3
7 8 -

(child 68) 1 2 3
7 8 -

(child 69) 1 2 3
7 8 -

(child 70) 1 2 3
7 8 -

(child 71) 1 2 3
7 8 -

(child 72) 1 2 3
7 8 -

(child 73) 1 2 3
7 8 -

(child 74) 1 2 3
7 8 -

(child 75) 1 2 3
7 8 -

(child 76) 1 2 3
7 8 -

(child 77) 1 2 3
7 8 -

(child 78) 1 2 3
7 8 -

(child 79) 1 2 3
7 8 -

(child 80) 1 2 3
7 8 -

(child 81) 1 2 3
7 8 -

(child 82) 1 2 3
7 8 -

(child 83) 1 2 3
7 8 -

(child 84) 1 2 3
7 8 -

(child 85) 1 2 3
7 8 -

(child 86) 1 2 3
7 8 -

(child 87) 1 2 3
7 8 -

(child 88) 1 2 3
7 8 -

(child 89) 1 2 3
7 8 -

(child 90) 1 2 3
7 8 -

(child 91) 1 2 3
7 8 -

(child 92) 1 2 3
7 8 -

(child 93) 1 2 3
7 8 -

(child 94) 1 2 3
7 8 -

(child 95) 1 2 3
7 8 -

(child 96) 1 2 3
7 8 -

(child 97) 1 2 3
7 8 -

(child 98) 1 2 3
7 8 -

(child 99) 1 2 3
7 8 -

(child 100) 1 2 3
7 8 -

(child 101) 1 2 3
7 8 -

(child 102) 1 2 3
7 8 -

(child 103) 1 2 3
7 8 -

(child 104) 1 2 3
7 8 -

(child 105) 1 2 3
7 8 -

(child 106) 1 2 3
7 8 -

(child 107) 1 2 3
7 8 -

(child 108) 1 2 3
7 8 -

(child 109) 1 2 3
7 8 -

(child 110) 1 2 3
7 8 -

(child 111) 1 2 3
7 8 -

(child 112) 1 2 3
7 8 -

(child 113) 1 2 3
7 8 -

(child 114) 1 2 3
7 8 -

(child 115) 1 2 3
7 8 -

(child 116) 1 2 3
7 8 -

(child 117) 1 2 3
7 8 -

(child 118) 1 2 3
7 8 -

(child 119) 1 2 3
7 8 -

(child 120) 1 2 3
7 8 -

(child 121) 1 2 3
7 8 -

(child 122) 1 2 3
7 8 -

(child 123) 1 2 3
7 8 -

(child 124) 1 2 3
7 8 -

(child 125) 1 2 3
7 8 -

(child 126) 1 2 3
7 8 -

(child 127) 1 2 3
7 8 -

(child 128) 1 2 3
7 8 -

(child 129) 1 2 3
7 8 -

(child 130) 1 2 3
7 8 -

(child 131) 1 2 3
7 8 -

(child 132) 1 2 3
7 8 -

(child 133) 1 2 3
7 8 -

(child 134) 1 2 3
7 8 -

(child 135) 1 2 3
7 8 -

(child 136) 1 2 3
7 8 -

(child 137) 1 2 3
7 8 -

(child 138) 1 2 3
7 8 -

(child 139) 1 2 3
7 8 -

(child 140) 1 2 3
7 8 -

(child 141) 1 2 3
7 8 -

(child 142) 1 2 3
7 8 -

(child 143) 1 2 3
7 8 -

(child 144) 1 2 3
7 8 -

(child 145) 1 2 3
7 8 -

(child 146) 1 2 3
7 8 -

(child 147) 1 2 3
7 8 -

(child 148) 1 2 3
7 8 -

(child 149) 1 2 3
7 8 -

(child 150) 1 2 3
7 8 -

(child 151) 1 2 3
7 8 -

(child 152) 1 2 3
7 8 -

(child 153) 1 2 3
7 8 -

(child 154) 1 2 3
7 8 -

(child 155) 1 2 3
7 8 -

(child 156) 1 2 3
7 8 -

(child 157) 1 2 3
7 8 -

(child 158) 1 2 3
7 8 -

(child 159) 1 2 3
7 8 -

(child 160) 1 2 3
7 8 -

(child 161) 1 2 3
7 8 -

(child 162) 1 2 3
7 8 -

(child 163) 1 2 3
7 8 -

(child 164) 1 2 3
7 8 -

(child 165) 1 2 3
7 8 -

(child 166) 1 2 3
7 8 -

(child 167) 1 2 3
7 8 -

(child 168) 1 2 3
7 8 -

(child 169) 1 2 3
7 8 -

(child 170) 1 2 3
7 8 -

(child 171) 1 2 3
7 8 -

(child 172) 1 2 3
7 8 -

(child 173) 1 2 3
7 8 -

(child 174) 1 2 3
7 8 -

(child 175) 1 2 3
7 8 -

(child 176) 1 2 3
7 8 -

(child 177) 1 2 3
7 8 -

(child 178) 1 2 3
7 8 -

(child 179) 1 2 3
7 8 -

(child 180) 1 2 3
7 8 -

(child 181) 1 2 3
7 8 -

(child 182) 1 2 3
7 8 -

(child 183) 1 2 3
7 8 -

(child 184) 1 2 3
7 8 -

(child 185) 1 2 3
7 8 -

(child 186) 1 2 3
7 8 -

(child 187) 1 2 3
7 8 -

(child 188) 1 2 3
7 8 -

(child 189) 1 2 3
7 8 -

(child 190) 1 2 3
7 8 -

(child 191) 1 2 3
7 8 -

(child 192) 1 2 3
7 8 -

(child 193) 1 2 3
7 8 -

(child 194) 1 2 3
7 8 -

(child 195) 1 2 3
7 8 -

(child 196) 1 2 3
7 8 -

Code

```
import heapq
import time

# Heuristic: Manhattan Distance
def heuristic(state, goal):
    distance = 0
    for i in range(1, 9): # tile numbers 1 to 8
        x1, y1 = divmod(state.index(i), 3)
        x2, y2 = divmod(goal.index(i), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

# Get neighbors by sliding blank (0) up/down/left/right
def get_neighbors(state):
    neighbors = []
    i = state.index(0) # position of blank
    x, y = divmod(i, 3)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            j = new_x * 3 + new_y
            new_state = list(state)
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(tuple(new_state))
    return neighbors

# A* Search for 8-puzzle
def astar(start, goal):
    open_set = []
    heapq.heappush(open_set, (heuristic(start, goal), 0, start))

    came_from = {}
    g_score = {start: 0}

    while open_set:
        _, cost, current = heapq.heappop(open_set)

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(goal)
            path.reverse()
            return path

        for neighbor in get_neighbors(current):
            tentative_g_score = cost + 1
            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                came_from[neighbor] = current
                f_score = g_score[neighbor] + heuristic(neighbor, goal)
                heapq.heappush(open_set, (f_score, tentative_g_score, neighbor))

    return None
```

```

        current = came_from[current]
        path.append(start)
        return path[::-1]

for neighbor in get_neighbors(current):
    tentative_g = g_score[current] + 1
    if neighbor not in g_score or tentative_g < g_score[neighbor]:
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g
        f_score = tentative_g + heuristic(neighbor, goal)
        heapq.heappush(open_set, (f_score, tentative_g, neighbor))

return None # no solution

# ----- TEST -----
start = (1, 2, 3,
         4, 8, 0,
         7, 6, 5)

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

# Measure execution time
start_time = time.time()
path = astar(start, goal)
end_time = time.time()

if path:
    print("Steps to solve ({} moves):".format(len(path)-1))
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()
else:
    print("No solution found")

print("Execution time: {:.6f} seconds".format(end_time - start_time))

```

Output

```
Steps to solve (5 moves):
```

```
(1, 2, 3)
```

```
(4, 8, 0)
```

```
(7, 6, 5)
```

```
(1, 2, 3)
```

```
(4, 8, 5)
```

```
(7, 6, 0)
```

```
(1, 2, 3)
```

```
(4, 8, 5)
```

```
(7, 0, 6)
```

```
(1, 2, 3)
```

```
(4, 0, 5)
```

```
(7, 8, 6)
```

```
(1, 2, 3)
```

```
(4, 5, 0)
```

```
(7, 8, 6)
```

```
(1, 2, 3)
```

```
(4, 5, 6)
```

```
(7, 8, 0)
```

```
Execution time: 0.000111 seconds
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm

9/10/25

Bafna Gold -

Date: _____
Page: _____

Week 4

Q N Queens problem, Using Hill Climbing

function Hill-Climbing (Problem) returns local
minima

 current \leftarrow Make Node (Problem.initialState)

 while True:

 next \leftarrow get best neighbor (current)

 if cost (current) \leq cost (next)

 then break

 end if

 current \leftarrow next

 end while

return current

Eg) Initial State =

0	1	2	3
			Q
	Q		
Q			

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

Cost = 2

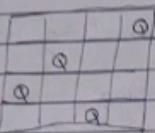
Possible children

i) Swap (x_0, x_1)

		Q	
	Q		
Q			

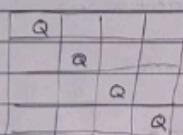
Choose this

$x_0 = 1, x_1 = 3,$

i) Swap (x_0, x_2) 

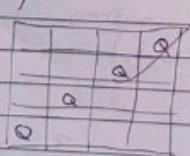
$$x_0 = 2, x_1 = 1, x_2 = 3, x_3 = 0$$

Cost = 1

ii) Swap (x_0, x_3) 

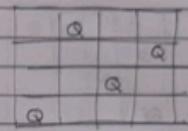
$$x_0 = 0, x_1 = 1, x_2 = 2, x_3 = 3$$

Cost = 6

iii) Swap (x_1, x_2) 

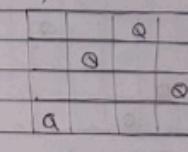
$$x_0 = 3, x_1 = 2, x_2 = 1, x_3 = 0$$

Cost = 6

iv) Swap (x_1, x_3) 

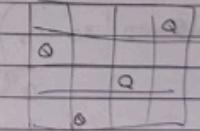
$$x_0 = 3, x_1 = 0, x_2 = 2, x_3 = 1$$

Cost = 1

v) Swap (x_2, x_3) 

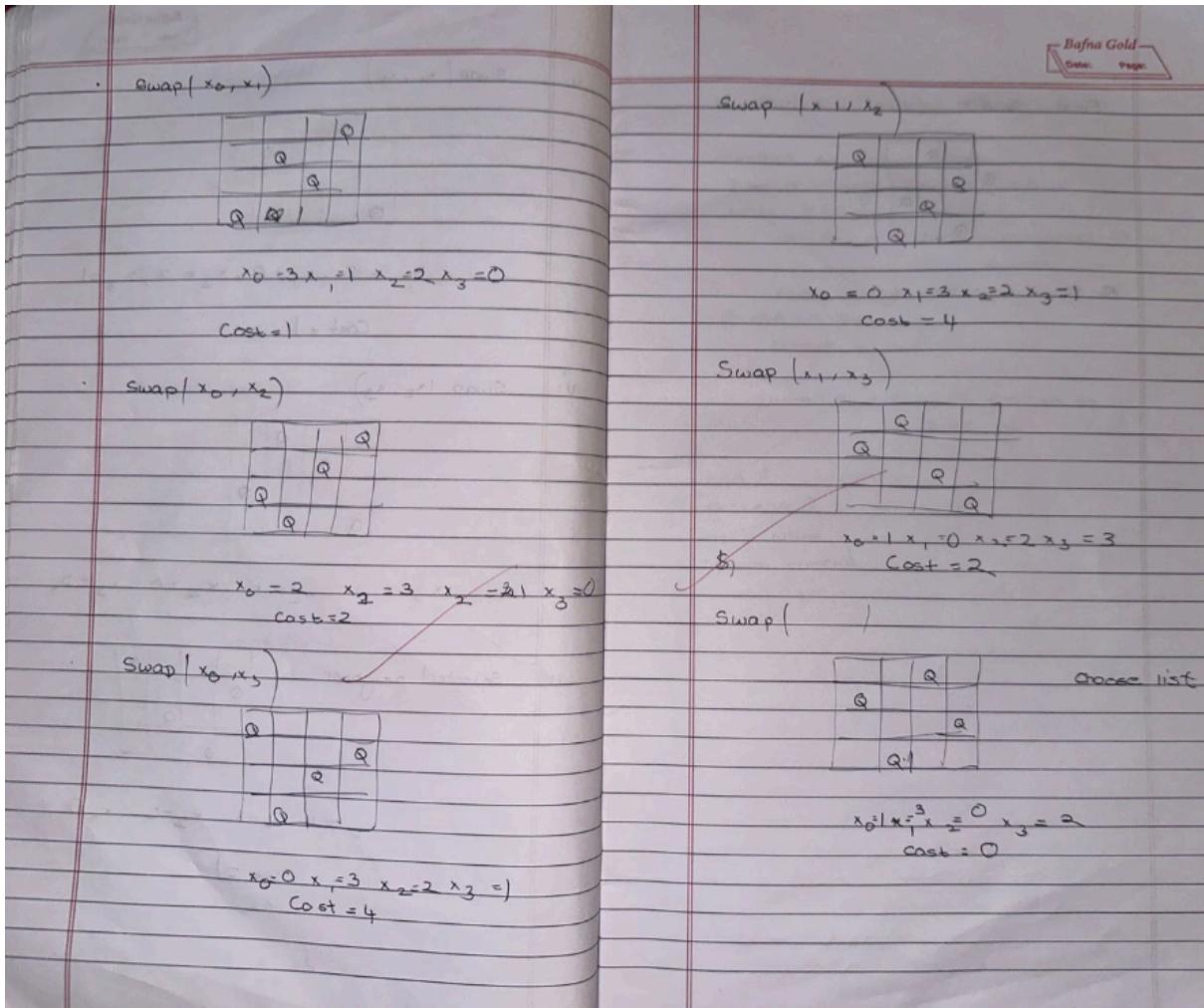
$$x_0 = 3, x_1 = 1, x_2 = 0, x_3 = 2$$

vi) Selected neighbor =



$$x_0 = 1, x_1 = 3, x_2 = 2, x_3 = 0$$

Cost = 1



Code

```

import random
import math

def compute_cost(state):
    """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_permutation(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr

```

```

def neighbors_by_swaps(state):
    """All neighbors obtained by swapping two columns (keeps permutation property)."""
    n = len(state)
    for i in range(n - 1):
        for j in range(i + 1, n):
            nb = state.copy()
            nb[i], nb[j] = nb[j], nb[i]
            yield nb

def hill_climb_with_restarts(n, max_restarts=None):
    """Hill climbing on permutations with random restart on plateau (no revisits)."""
    visited = set()
    total_states = math.factorial(n)
    restarts = 0

    while True:
        # pick a random unvisited start permutation
        if len(visited) >= total_states:
            raise RuntimeError("All states visited — giving up (no solution found.)")

        state = random_permutation(n)
        while tuple(state) in visited:
            state = random_permutation(n)
        visited.add(tuple(state))

        # climb from this start
        while True:
            cost = compute_cost(state)
            if cost == 0:
                return state, restarts

            # find best neighbor (swap-based neighbors)
            best_neighbor = None
            best_cost = float("inf")
            for nb in neighbors_by_swaps(state):
                c = compute_cost(nb)
                if c < best_cost:
                    best_cost = c
                    best_neighbor = nb

            # if strictly better, move; otherwise it's a plateau/local optimum -> restart
            if best_cost < cost:
                state = best_neighbor
                visited.add(tuple(state))
            else:
                # plateau or local optimum -> restart

```

```

restarts += 1
if max_restarts is not None and restarts >= max_restarts:
    raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).")
break # go pick a new unvisited start

def format_board(state):
    n = len(state)
    lines = []
    for r in range(n):
        lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
    return "\n".join(lines)

if __name__ == "__main__":
    n = 4
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))

```

Output

```

Found solution: [2, 0, 3, 1]
- Q -
- - - Q
Q - - -
- - Q -

```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm

N Queens Simulated Annealing

```
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
        p = e^(-ΔE/T)
    end if
    decrease T
end while
return current
```

Code

```
import random
import math
```

```
def cost(state):
```

```
    attacks = 0
```

```

n = len(state)
for i in range(n):
    for j in range(i + 1, n):
        if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
            attacks += 1
return attacks

```

```

def get_neighbor(state):

    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

```

```
def simulated_annealing(n=8, max_iter=10000, temp=100.0, cooling=0.95):
```

```

    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

```

```

    temperature = temp
    cooling_rate = cooling

```

```

    best = current[:]
    best_cost = current_cost

```

```

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

```

```

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

```

```

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor[:], neighbor_cost

```

```
temperature *= cooling_rate
```

```
return best, best_cost
```

```
def print_board(state):
```

```

    n = len(state)
    for row in range(n):

```

```
line = " ".join("Q" if state[col] == row else "." for col in range(n))
print(line)
print()

n = 8
solution, cost_val = simulated_annealing(n, max_iter=20000)
print("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

Output

```
Best position found: [6, 3, 1, 7, 5, 0, 2, 4]
Number of non-attacking pairs: 28

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

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

Program 6

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

Algorithm

<p style="text-align: center;">Week 5 - Propositional logic</p> <p>a) Create a knowledge base using propositional logic and query details the knowledge base</p> <p>→ Algorithm $\text{TT_entails_user_input}()$</p> <p>Input:</p> <ul style="list-style-type: none"> $\text{KB} \leftarrow \text{sentences} \leftarrow \text{list of propositional sentences from user}$ $\text{query} \leftarrow \text{query sentence from user}$ <p>Process:</p> <pre> Symbols ← all unique propositional symbols appearing in KB sentences return $\text{TT_checkAll}(\text{KB sentences}, \text{query}, \text{symbols}, \text{empty model})$ </pre> <p>→ Function $\text{TT_check_all}(\text{KB}, \alpha, \text{symbols}, \text{model})$</p> <pre> if symbols is empty then if $\text{PL_true?}(\text{KB}, \text{model})$ then return $\text{PL_true?}(\alpha, \text{model})$ else return true else p ← first(symbols) rest ← symbols - {p} model_true = model ∪ {p = true} model_false = model ∪ {p = false} </pre>	<p>Bajna Gold Date: _____ Page: _____</p> <p>return $\text{TT_check_all}(\text{KB}, \alpha, \text{rest}, \text{model}) \text{ AND } \text{TT_check_all}(\text{KB}, \neg\alpha, \text{rest}, \text{model_false})$</p> <p>→ function $\text{PL_true?}(\text{sentence_set}, \text{model})$</p> <pre> for each sentence S in sentence set if S evaluate to false under model return False return True </pre> <p>a) Consider a knowledge base KB that contains the following propositional layered logic sentences:</p> $Q \rightarrow P$ $P \rightarrow TQ$ <p>QVR</p> <p>construct a truth table that shows the value of each sentence in KB and is:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>P</th> <th>Q</th> <th>R</th> <th>$Q \rightarrow P$</th> <th>$P \rightarrow TQ$</th> <th>QVR</th> </tr> </thead> <tbody> <tr> <td>T</td> <td>T</td> <td>T</td> <td>T</td> <td>F</td> <td>T</td> </tr> <tr> <td>T</td> <td>T</td> <td>F</td> <td>T</td> <td>F</td> <td>T</td> </tr> <tr> <td>T</td> <td>F</td> <td>T</td> <td>T</td> <td>T</td> <td>T</td> </tr> <tr> <td>T</td> <td>F</td> <td>F</td> <td>T</td> <td>T</td> <td>F</td> </tr> <tr> <td>F</td> <td>T</td> <td>T</td> <td>F</td> <td>T</td> <td>T</td> </tr> <tr> <td>F</td> <td>T</td> <td>F</td> <td>F</td> <td>T</td> <td>T</td> </tr> <tr> <td>F</td> <td>F</td> <td>T</td> <td>T</td> <td>T</td> <td>T</td> </tr> <tr> <td>F</td> <td>F</td> <td>F</td> <td>T</td> <td>T</td> <td>T</td> </tr> </tbody> </table> <p>ii) Does KB entail R</p> <p>In all models where KB is true, R = T \therefore KB entails R</p> <p>iii) Does KB entail $R \rightarrow P$</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>P</th> <th>Q</th> <th>R</th> <th>$R \rightarrow P$</th> </tr> </thead> <tbody> <tr> <td>T</td> <td>F</td> <td>T</td> <td>T</td> </tr> <tr> <td>F</td> <td>F</td> <td>T</td> <td>F</td> </tr> </tbody> </table> <p>} Both false \Rightarrow KB does not entail $R \rightarrow P$</p>	P	Q	R	$Q \rightarrow P$	$P \rightarrow TQ$	QVR	T	T	T	T	F	T	T	T	F	T	F	T	T	F	T	T	T	T	T	F	F	T	T	F	F	T	T	F	T	T	F	T	F	F	T	T	F	F	T	T	T	T	F	F	F	T	T	T	P	Q	R	$R \rightarrow P$	T	F	T	T	F	F	T	F
P	Q	R	$Q \rightarrow P$	$P \rightarrow TQ$	QVR																																																														
T	T	T	T	F	T																																																														
T	T	F	T	F	T																																																														
T	F	T	T	T	T																																																														
T	F	F	T	T	F																																																														
F	T	T	F	T	T																																																														
F	T	F	F	T	T																																																														
F	F	T	T	T	T																																																														
F	F	F	T	T	T																																																														
P	Q	R	$R \rightarrow P$																																																																
T	F	T	T																																																																
F	F	T	F																																																																

iii Does KB entail $Q \Rightarrow R$

PQR	$Q \Rightarrow R$	Always true \Rightarrow KB entails $Q \Rightarrow R$
TFT	T	
FFT	T	

$\alpha = A \vee B$

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

A	B	C	$A \vee C$	$B \vee C$	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	(T)	(T)
T	F	F	T	T	(T)	(T)
T	F	T	T	F	F	T
T	T	F	T	T	(T)	(T)
T	T	T	T	T	(T)	(T)

$\therefore KB \text{ entails } \alpha$

Ch 11/15

Code

```

import itertools
def evaluate_formula(formula, truth_assignment):
    eval_formula = formula
    for symbol, value in truth_assignment.items():
        eval_formula = eval_formula.replace(symbol, str(value))
    return eval(eval_formula)

def generate_truth_table(variables):
    return list(itertools.product([False, True], repeat=len(variables)))

def is_entailed(KB_formula, alpha_formula, variables):
    truth_combinations = generate_truth_table(variables)

```

```

print(f"{''.join(variables)} | KB Result | Alpha Result")
print("-" * (len(variables) * 2 + 15))
for combination in truth_combinations:
    truth_assignment = dict(zip(variables, combination))
    KB_value = evaluate_formula(KB_formula, truth_assignment)
    alpha_value = evaluate_formula(alpha_formula, truth_assignment)
    result_str = " ".join(["T" if value else "F" for value in combination])
    print(f'{result_str} | {KB_value} | {alpha_value}')
    if KB_value and not alpha_value:
        return False
return True

KB = "(A or C) and (B or not C)"
alpha = "A or B"
variables = ['A', 'B', 'C']

if is_entailed(KB, alpha, variables):
    print("\nThe knowledge base entails alpha.")
else:
    print("\nThe knowledge base does not entail alpha.")

```

Output

A	B	C		KB Result		Alpha Result
<hr/>						
F	F	F		F		F
F	F	T		F		F
F	T	F		F		T
F	T	T		T		T
T	F	F		T		T
T	F	T		F		T
T	T	F		T		T
T	T	T		T		T
 The knowledge base entails alpha.						

Program 7

Implement unification in first order logic

Algorithm

Bafna Gold
Date: _____ Page: _____

29-10-25 Week 6 - Unification of FOL

① $P(f(x) \rightarrow g(y), y)$
 $P(f(g(z)), g(f(a)), f(a))$
 Find Θ (Mgu)

② $Q(x, F(x))$
 $Q(F(y), y)$

③ $H(x, g(x))$
 $H(g(y), g(g(z)))$

Answer

1 Compare x with $g(z)$, $\Theta_1 = \{x/g(z)\}$
 y with $F(a)$, $\Theta_2 = \{y/F(a)\}$

Substituting Θ in $P(F(x), g(y), y)$
 $= P(F(g(z)), g(F(a)), f(a))$

i. both are identical \therefore unified

$\Theta = \{x/g(z), y/F(a)\}$

2 Comparing x with $F(y)$

Substituting Θ in $Q(x, F(x))$
 $= Q(F(y), F(F(y)))$

$$f(f(y)) = y$$

it is cyclic so not unifiable

3) comparing n with $g(y)$

$$n = g(y)$$

comparing $g(z)$ & $g(g(z))$

$$n = g(y)$$

$$\Theta_1 = \{n/g(y)\}$$

$$g(x) = g(g(y))$$

If $y = z$, $g(g(y))$ can be unified but $g(z)$

$$\Theta_2 = \{y, z\}$$

put Θ_1 in $H(x, g(x))$

$$\Theta_1(\text{MGU}) = [x, gy], y, z]$$

$$= H(gy), g(g(y))$$

$$= H(gy), g(g(z))$$

∴ it is identical

∴ it is unifiable.

eventually
unifiable

without
NP proof

of puzzle.

Code

```
import re

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

def parse(term):
    term = term.strip()
    if '(' not in term:
        return term
    name, args = term.split('(', 1)
    args = args[:-1] # remove closing parenthesis
    return name.strip(), [parse(a.strip()) for a in args.split(',')]

def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        _, args = expr
        return any(occurs_check(var, a) for a in args)
    return False

def substitute(subs, expr):
    if isinstance(expr, str):
        if expr in subs:
            return substitute(subs, subs[expr])
        return expr
    else:
        func, args = expr
        return (func, [substitute(subs, a) for a in args])

def unify(x, y, subs=None):
    if subs is None:
        subs = {}
    x = substitute(subs, x)
    y = substitute(subs, y)
    if x == y:
```

```

return subs

if isinstance(x, str) and is_variable(x):
    if occurs_check(x, y):
        return None
    subs[x] = y
    return subs

if isinstance(y, str) and is_variable(y):
    if occurs_check(y, x):
        return None
    subs[y] = x
    return subs

if isinstance(x, tuple) and isinstance(y, tuple):
    if x[0] != y[0] or len(x[1]) != len(y[1]):
        return None
    for a, b in zip(x[1], y[1]):
        subs = unify(a, b, subs)
        if subs is None:
            return None
    return subs

return None

def term_to_str(t):
    if isinstance(t, str):
        return t
    func, args = t
    return f'{func}({",".join(term_to_str(a) for a in args)})'

def pretty_print(subs):
    return ','.join(f'{v} : {term_to_str(t)}' for v, t in subs.items())

pairs = [
    ("P(f(x),g(y),y)", "P(f(g(z)),g(f(a)),f(a))"),
    ("Q(x,f(x))", "Q(f(y),y)"),
    ("H(x,g(x))", "H(g(y),g(g(z))))")
]

```

]

```
for s1, s2 in pairs:  
    print(f"\nUnifying: {s1} and {s2}")  
    result = unify(parse(s1), parse(s2))  
    if result:  
        print("=> Substitution:", pretty_print(result))  
    else:  
        print("=> Not unifiable.")
```

Output:

```
Unifying: P(f(x),g(y),y) and P(f(g(z)),g(f(a)),f(a))  
=> Substitution: x : g(z), y : f(a)  
  
Unifying: Q(x,f(x)) and Q(f(y),y)  
=> Not unifiable.  
  
Unifying: H(x,g(x)) and H(g(y),g(g(z)))  
=> Substitution: x : g(y), y : z
```

Program 8

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

Algorithm

Bafna Gold
Date: _____ Page: _____

Forward Reasoning Algorithm

Function $FOR_FC - ASK(KB, \alpha)$ returns a substitution or false

Inputs: KB, the knowledge base, a set of first-order default clauses.

Let the query, an atomic sentence local variables: new, the new set of sentences inferred on each iteration.

repeat until new is empty

new = {}

for each rule in KB do

$$(p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE}$$

- VARIABLES (rule)

for each θ such that $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = q$

= $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n)$

for some p'_1, \dots, p'_n in KB

 ~~$q' \leftarrow \text{SUBST}(\theta, q)$~~

if q' does not unify with some sentence already in new then

add q' to new

$\theta \leftarrow \text{UNIFY}(q', \alpha)$

if θ is not fail then return θ

add new to KB

return false

Code

```
import re
```

```
def match_pattern(pattern, fact):  
    """
```

Checks if a fact matches a rule pattern using regex-style variable substitution.

Variables are lowercase words like p, q, x, r etc.
 Returns a dict of substitutions or None if not matched.

```

"""
# Extract predicate name and arguments
pattern_pred, pattern_args = re.match(r'(\w+)
', pattern).groups()
fact_pred, fact_args = re.match(r'(\w+)
', fact).groups()

if pattern_pred != fact_pred:
    return None # predicate mismatch

pattern_args = [a.strip() for a in pattern_args.split(",")]
fact_args = [a.strip() for a in fact_args.split(",")]

if len(pattern_args) != len(fact_args):
    return None

subst = {}
for p_arg, f_arg in zip(pattern_args, fact_args):
    if re.fullmatch(r'[a-z]\w*', p_arg): # variable
        subst[p_arg] = f_arg
    elif p_arg != f_arg: # constants mismatch
        return None
return subst

def apply_substitution(expr, subst):
    """Replaces all variable names in expr using the given substitution dict."""
    for var, val in subst.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr

```

----- Knowledge Base -----

```

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

facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",

```

```

    "Missile(T1)"
}

goal = "Criminal(Robert)"

def forward_chain(rules, facts, goal):
    added = True
    while added:
        added = False
        for premises, conclusion in rules:

            possible_substs = []
            for p in premises:
                for f in facts:
                    subst = match_pattern(p, f)
                    if subst:
                        possible_substs.append(subst)
                        break
                else:
                    break
            else:
                combined = {}
                for s in possible_substs:
                    combined.update(s)

                new_fact = apply_substitution(conclusion, combined)

                if new_fact not in facts:
                    facts.add(new_fact)
                    print(f"Inferred: {new_fact}")
                    added = True
                if new_fact == goal:
                    return True
    return goal in facts

print("Goal achieved:", forward_chain(rules, facts, goal))

```

Output

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Goal achieved: True
```

Program 9

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

Algorithm

Bafna Gold
Date: _____ Page: _____

12-11-23 Week 7

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

procedure TO-CNF(F):

Step 1: Eliminate implications and biconditionals, for each subformula in F :

if subformula is $(A \rightarrow B)$:
 replace with $(\neg A \vee B)$

if subformula is $(A \leftrightarrow B)$:
 replace with $((\neg A \vee B) \wedge (\neg B \vee A))$

Step 2: Move negations inward while there exists a negation applied to a compound formula:

apply De Morgan's laws:
 $\neg(A \wedge B) \rightarrow (\neg A \vee \neg B)$
 $\neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$

push \neg through quantifiers:

$\neg \forall x P(x) \rightarrow \exists x \neg P(x)$
 $\neg \exists x P(x) \rightarrow \forall x \neg P(x)$

remove ^{double} negations:

$\neg(\neg A) \rightarrow A$

Step 3: Standardize variables for each quantified variable x in F :

if variable name repeats:
 rename it to a unique variable

Step 4: Skolemize / remove existential quantifiers

For each $\exists x$ in F:

if $\exists x$ is inside $\forall y_1, \forall y_2, \dots, \forall y_n$,

replace x with a new skolem

Function $F(y_1, y_2, \dots, y_n)$

else:

replace x with a new skolem constant

remove \exists quantifier

Step 5: Drop universal quantifiers

remove all \forall quantifiers (all variables now

implicitly universal)

Step 6: Distribute V over A

repeat until no distribution is possible

apply rules:

$$(A \vee (B \wedge C)) \rightarrow ((A \vee B) \wedge (A \vee C))$$

$$((A \wedge B) \vee C) \rightarrow ((A \vee C) \wedge (B \vee C))$$

Step 7: Simplify

remove duplicate literals in clauses

remove tautological clauses (where A and $\neg A$ appear

together)

return F

Output:

Function $(A \rightarrow (B \vee C)) \wedge \neg D \rightarrow E$

Output: $(\neg A \vee B \vee C) \wedge \neg D \wedge E$

Code

```
from copy import deepcopy
```

```
def print_step(title, content):
    print(f"\n{'='*45}\n{title}\n{'='*45}")
    if isinstance(content, list):
        for i, c in enumerate(content, 1):
            print(f'{i}. {c}')
```

```

else:
    print(content)

KB = [
    ["¬Food(x)", "Likes(John,x)"],
    ["Food(Apple)"],
    ["Food(Vegetable)"],
    ["¬Eats(x,y)", "Killed(x)", "Food(y)"],
    ["Eats(Anil,Peanuts)"],
    ["Alive(Anil)"],
    ["¬Alive(x)", "¬Killed(x)"],
    ["Killed(x)", "Alive(x)"]
]
QUERY = ["Likes(John,Peanuts)"]

def negate(literal):
    if literal.startswith("¬"):
        return literal[1:]
    return "¬" + literal

def substitute(clause, subs):
    new_clause = []
    for lit in clause:
        for var, val in subs.items():
            lit = lit.replace(var, val)
        new_clause.append(lit)
    return new_clause

def unify(lit1, lit2):
    """Small unifier for patterns like Food(x) and Food(Apple)."""
    if "(" not in lit1 or "(" not in lit2:
        return None
    pred1, args1 = lit1.split("(")
    pred2, args2 = lit2.split("(")
    args1 = args1[:-1].split(",")
    args2 = args2[:-1].split ","
    if pred1 != pred2 or len(args1) != len(args2):
        return None
    subs = {}
    for a, b in zip(args1, args2):
        if a == b:
            continue
        if a.islower():
            subs[a] = b
        elif b.islower():
            subs[b] = a

```

```

else:
    return None
return subs

def resolve(ci, cj):
    """Return list of (resolvent, substitution, pair)."""
    resolvents = []
    for li in ci:
        for lj in cj:
            if li == negate(lj):
                new_clause = [x for x in ci if x != li] + [x for x in cj if x != lj]
                resolvents.append((list(set(new_clause)), {}, (li, lj)))
            else:
                # same predicate, opposite sign
                if li.startswith("¬") and not lj.startswith("¬") and li[1:].split("(")[0] == lj.split("(")[0]:
                    subs = unify(li[1:], lj)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
                        resolvents.append((list(set(new_clause)), subs, (li, lj)))
                elif lj.startswith("¬") and not li.startswith("¬") and lj[1:].split("(")[0] == li.split("(")[0]:
                    subs = unify(lj[1:], li)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
                        resolvents.append((list(set(new_clause)), subs, (li, lj)))
    return resolvents

def resolution(kb, query):
    clauses = deepcopy(kb)
    negated_query = [negate(q) for q in query]
    clauses.append(negated_query)
    print_step("Initial Clauses", clauses)

    steps = []
    new = []
    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses))
                  for j in range(i + 1, len(clauses))]
        for (ci, cj) in pairs:
            for r, subs, pair in resolve(ci, cj):
                if not r:
                    steps.append({
                        "parents": (ci, cj),
                        "resolvent": r,
                        "subs": subs
                    })
        print_tree(steps)
        print("\n✓ Empty clause derived — query proven.")

```

```

        return True
    if r not in clauses and r not in new:
        new.append(r)
        steps.append({
            "parents": (ci, cj),
            "resolvent": r,
            "subs": subs
        })
    if all(r in clauses for r in new):
        print_step("No New Clauses", "Query cannot be proven ✗")
        print_tree(steps)
        return False
    clauses.extend(new)

def print_tree(steps):
    print("\n" + "="*45)
    print("Resolution Proof Trace")
    print("="*45)
    for i, s in enumerate(steps, 1):
        p1, p2 = s["parents"]
        r = s["resolvent"]
        subs = s["subs"]
        subs_text = f" Substitution: {subs}" if subs else ""
        print(f" Resolve {p1} and {p2}")
        if subs_text:
            print(subs_text)
        if r:
            print(f" ⇒ {r}")
        else:
            print(" ⇒ {} (empty clause)")
    print("-"*45)

def main():
    print_step("Knowledge Base in CNF", KB)
    print_step("Negated Query", [negate(q) for q in QUERY])
    proven = resolution(KB, QUERY)
    if proven:
        print("\n✓ Query Proven by Resolution: John likes peanuts.")
    else:
        print("\n✗ Query cannot be proven from KB.")

if __name__ == "__main__":
    main()

```

Output

```
Unifying: P(f(x),g(y),y)  and  P(f(g(z)),g(f(a)),f(a))
          ))
=> Substitution: x : g(z), y : f(a)

Unifying: Q(x,f(x))  and  Q(f(y),y)
=> Not unifiable.

Unifying: H(x,g(x))  and  H(g(y),g(g(z)))
=> Substitution: x : g(y), y : z

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

Program 10

Implement Alpha-Beta Pruning

Algorithm

Alpha beta Search

Algorithm

Function ALPHA-BETA-SEARCH(state) returns an action
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$
 return the action in ACTIONS(state) with value v

Function MAX-VALUE(state, α, β) returns a utility value
 if TERMINAL TEST(state) then return UTILITY(state)

$v \leftarrow -\infty$

for each a in ACTIONS(state) do

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \geq \beta$ then return v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return v

Function MIN-VALUE(state, α, β) returns a utility value

if terminal TEST(state) then return UTILITY(state)

$v \leftarrow +\infty$

for each action a in ACTIONS(state) do

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), -\infty, \beta))$

if $v \leq \beta$ then return v

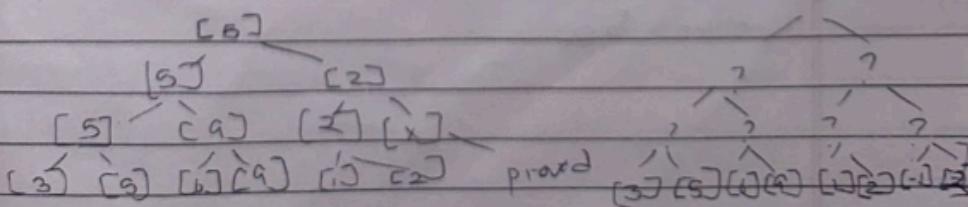
$\beta \leftarrow \text{MIN}(\beta, v)$

return v

Output

leaf Node Values: [3, 5, 6, 9, 1, 2, 0, -1]

Optimal Value: 5



Code

```
def unify(a, b):
    """Very simple unification for small terms like ('line', [X,O,X])"""
    if a == b:
        return {}
    if isinstance(a, str) and a.islower(): # variable
        return {a: b}
    if isinstance(b, str) and b.islower():
        return {b: a}
    if isinstance(a, tuple) and isinstance(b, tuple):
        if a[0] != b[0] or len(a[1]) != len(b[1]):
            return None
        subs = {}
        for x, y in zip(a[1], b[1]):
            s = unify(x, y)
            if s is None:
                return None
            subs.update(s)
        return subs
    return None

# Winning triples (rows, cols, diagonals)
WIN_TRIPLES = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def winner(board):
    pattern = ('line', ['X','X','X'])
    for i,j,k in WIN_TRIPLES:
        term = ('line', [board[i], board[j], board[k]])
        if unify(term, pattern):
            return 'X'
        if unify(term, ('line',['O','O','O'])):
            return 'O'
    return None

def is_full(board): return all(c != '_' for c in board)

def evaluate(board):
    w = winner(board)
    if w == 'X': return 1
    if w == 'O': return -1
    if is_full(board): return 0
    return None
```

```

def alpha_beta(board, player, alpha=-float('inf'), beta=float('inf')):
    val = evaluate(board)
    if val is not None:
        return val, None

    moves = [i for i,c in enumerate(board) if c == '_']
    best_move = None
    if player == 'X':
        max_eval = -float('inf')
        for m in moves:
            new_board = board[:]
            new_board[m] = 'X'
            eval_ = alpha_beta(new_board, 'O', alpha, beta)
            if eval_ > max_eval:
                max_eval, best_move = eval_, m
            alpha = max(alpha, eval_)
            if beta <= alpha: break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        for m in moves:
            new_board = board[:]
            new_board[m] = 'O'
            eval_ = alpha_beta(new_board, 'X', alpha, beta)
            if eval_ < min_eval:
                min_eval, best_move = eval_, m
            beta = min(beta, eval_)
            if beta <= alpha: break
        return min_eval, best_move

def print_board(b):
    for i in range(0,9,3):
        print(''.join(b[i:i+3]))
    print()

# --- Example usage ---
board = '_'*9
score, move = alpha_beta(board, 'X')
print("Best first move for X:", move)
board[move] = 'X'
print_board(board)

```

Output

<pre>You are X. AI is O. ---+---+ ---+---+ Enter your move (0-8): 2 X ---+---+ ---+---+ AI is thinking... X ---+---+ O ---+---+ Enter your move (0-8): 6 X ---+---+ O ---+---+ X </pre>	<pre>AI is thinking... O X ---+---+ O ---+---+ X Enter your move (0-8): 7 O X ---+---+ O ---+---+ X X AI is thinking... O X ---+---+ O ---+---+ X X O Enter your move (0-8): 0 X O X ---+---+ O ---+---+ X X O AI is thinking... x o x ---+---+ o o ---+---+ x x o</pre>	<pre>Enter your move (0-8): 5 X O X ---+---+ O O X ---+---+ X X O Result: Draw</pre>
---	--	--