Artificial Intelligence

Lab Report

NIIT University

Neemrana, Rajasthan

Submitted By:

Ankita Kapoor	**BT22GCS280**	**Btech CSE**

Table of Content:

Lab 1 Instructions:

Tic -Tac-Toe Game

Code:

```python
def dom(arr):  # dom = dominance
    x_count = arr.count('x')
    o_count = arr.count('o')

    # Check for winning conditions
    if x_count == 3:
        return 100  # Winning condition for 'x'
    elif o_count == 3:
        return -100  # Winning condition for 'o'

    # Calculate dominance
    return x_count - o_count

def strength(board):
    x_strength = 0
    o_strength = 0

    # Check rows
    for row in board:
        dom_value = dom(row)
        if dom_value > 0:
            x_strength += dom_value
        elif dom_value < 0:
            o_strength -= dom_value

    # Check columns
    for col in range(3):
        column = [board[row][col] for row in range(3)]
        dom_value = dom(column)
        if dom_value > 0:
```

```python
            x_strength += dom_value
        elif dom_value < 0:
            o_strength -= dom_value

    # Check diagonals
    diagonal1 = [board[i][i] for i in range(3)]
    diagonal2 = [board[i][2-i] for i in range(3)]

    for diagonal in [diagonal1, diagonal2]:
        dom_value = dom(diagonal)
        if dom_value > 0:
            x_strength += dom_value
        elif dom_value < 0:
            o_strength -= dom_value

    return x_strength, o_strength

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

def main():
    board = [
        ['x', 'b', 'b'],
        ['b', 'x', 'b'],
        ['o', 'b', 'b']
    ]

    print("Current Board:")
    print_board(board)

    x_strength, o_strength = strength(board)

    print(f"Strength of 'x': {x_strength}, Strength of 'o': {o_strength}")

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

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

tificial Intelligence\Lab Work'; & 'c:\Python\Python312\python.exe' 'c:\Users\Ankita Kapoor\.vscode\extensio
\adapter/../..\debugpy\launcher' '60976' '--' 'c:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Se
Current Board:
x|b|b
----------
b|x|b
----------
o|b|b
----------
Strength of 'x': 5, Strength of 'o': 1
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```

Lab 2 Instructions:

Given a n*n grid(n>=2) the player starts at location (0,0). Each cell of the grid can either be '_', 'G', 'C'.
'_' means the cell is empty.
'G' means the cell contains gold(exactly one cell of this kind).
'C' _______"_______ charge (_______"_______)
The player can move around in a car which has an initil charge I(an integer), the car can move in one of four directions(NSEW) and
it takes one unit of charge to make a move and find the minimum number of steps it takes to reac 'G'.
Note that the player might have to stop for charging as he may not be able to reach 'G' with the initial charge.

Code:

```
from collections import deque

def minimum_steps(grid, initial_charge):
    n = len(grid)
    print(n)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # East, West, South, North
    queue = deque([(0, 0, initial_charge, 0)])  # (x, y, charge, steps)
    visited = {(0, 0, initial_charge)}

    while queue:
        x, y, charge, steps = queue.popleft()
        if grid[x][y] == 'G':
            return steps
        if grid[x][y] == 'C':
            charge = initial_charge
        if charge <= 2:  # if charge is low, prioritize moving to charging point
            for dx, dy in directions:
                jx, jy = x + dx, y + dy
```

```python
                if 0 <= jx < n and 0 <= jy < n and grid[jx][jy] == 'C' and (jx, jy, charge - 1) not in visited:
                    queue.append((jx, jy, charge - 1, steps + 1))
                    visited.add((jx, jy, charge - 1))
        else:
            for dx, dy in directions:
                jx, jy = x + dx, y + dy
                if 0 <= jx < n and 0 <= jy < n and (jx, jy, charge - 1) not in visited:
                    queue.append((jx, jy, charge - 1, steps + 1))
                    visited.add((jx, jy, charge - 1))
    return -1

# Example:
grid = [
    ['_', '_', '_', '_'],
    ['_', '_', '_', '_'],
    ['C', '_', '_', '_'],
    ['_', '_', 'G', '_']
]
initial_charge = 1
print(minimum_steps(grid, initial_charge))
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work> & C:/Python/P
ademics/3rd Year/Sem 5/Artificial Intelligence/Lab Work/Lab2.py"
4
-1
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work> & C:/Python/P
ademics/3rd Year/Sem 5/Artificial Intelligence/Lab Work/Lab2.py"
4
-1
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```

Lab 3 Instructions:

In today's lab you will solve the "wolf, cabbage, goat" puzzle. Recall that the puzzle is –

- Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage.
- On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage.
- If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.
- The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?

Note that there are four entities - the farmer, the wolf, the goat and the cabbage. Any combination of them can be at the left bank and the remaining ones will be on the right bank. This observation will help you to solve the problem.

Your tasks are as follows:

1. Define the states of the problem – on pen and paper
2. Define the possible legal actions for transforming the states – on pen and paper
3. Define a data structure for representing the states
4. Define the initial state – in the representation chosen by you
   - Call this state as the current_state
5. Define the goal state – in the representation chosen by you

6. Apply the available actions and generate a set of child states of the current_state

7. Mark the current state as *visited*

8. For all child states of the current_state

    Check if the child state is allowed by the problem constraints

    If yes – check if the child state is a goal state

    If yes – print "goal reached" and stop

    If no – check if the child state is marked as visited

    If no – add the child state in a queue

9. Get a state from the queue – call it the current_state

10. Go to step 6

Code:

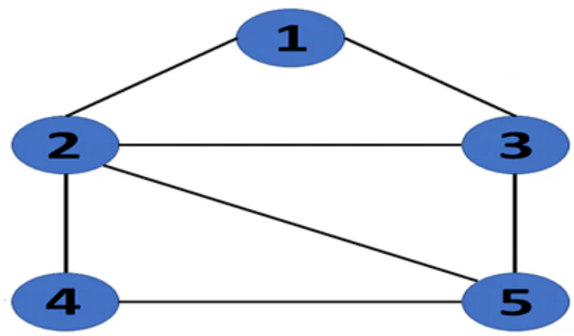
```
from collections import deque

def is_valid(state):
    farmer, wolf, goat, cabbage = state
    if wolf == goat and farmer != wolf:
        return False
    if goat == cabbage and farmer != goat:
        return False
    return True

def get_child_states(current_state):
    farmer, wolf, goat, cabbage = current_state
    moves = []

    if farmer == 'L':
        # Take wolf
        moves.append(('R', 'R' if wolf == 'L' else 'L', goat, cabbage))
        # Take goat
        moves.append(('R', wolf, 'R' if goat == 'L' else 'L', cabbage))
        # Take cabbage
```

```python
        moves.append(('R', wolf, goat, 'R' if cabbage == 'L' else 'L'))
        # Go alone
        moves.append(('R', wolf, goat, cabbage))
    else:

        # Bring wolf back
        moves.append(('L', 'L' if wolf == 'R' else 'R', goat, cabbage))
        # Bring goat back
        moves.append(('L', wolf, 'L' if goat == 'R' else 'R', cabbage))
        # Bring cabbage back
        moves.append(('L', wolf, goat, 'L' if cabbage == 'R' else 'R'))
        # Return alone
        moves.append(('L', wolf, goat, cabbage))

    return moves

def solve_puzzle():
    initial_state = ('L', 'L', 'L', 'L')
    goal_state = ('R', 'R', 'R', 'R')
    queue = deque([initial_state])
    visited = set()
    while queue:
        current_state = queue.popleft()
        print("Exploring state:", current_state)

        if current_state == goal_state:
            print("Goal reached!")
            return

        visited.add(current_state)

        for state in get_child_states(current_state):
            if is_valid(state) and state not in visited:
                queue.append(state)

    print("Visited States:", visited)

solve_puzzle()
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>  & 'c:\Python\
ns\ms-python.debugpy-2024.13.2024112901-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '51598' '--'
m 5\Artificial Intelligence\Lab Work\LAB3.py'
Exploring state: ('L', 'L', 'L', 'L')
Exploring state: ('R', 'L', 'R', 'L')
Exploring state: ('L', 'L', 'R', 'L')
Exploring state: ('R', 'R', 'R', 'L')
Exploring state: ('R', 'L', 'R', 'R')
Exploring state: ('L', 'R', 'L', 'L')
Exploring state: ('L', 'L', 'L', 'R')
Exploring state: ('R', 'R', 'L', 'R')
Exploring state: ('R', 'R', 'L', 'R')
Exploring state: ('L', 'R', 'L', 'R')
Exploring state: ('L', 'R', 'L', 'R')
Exploring state: ('R', 'R', 'R', 'R')
Goal reached!
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```

Lab 4 Instructions:



[0,1,1,0,0],

[1,0,1,1,1],

[1,1,0,0,1],

[0,1,0,0,1],

[0,0,1,1,0]

In today's assignment you will perform a BFS on the graph. Your program should accept the graph as an adjacency matrix. Your program should also be able to take any node as the initial node and any other node as the goal node. At each stage you will have to output the node lists present in the BFS queue and the total number of nodes in the queue. The total number of nodes will be the sum of number of nodes in each list in the queue. The successor() function should have a node as a parameter and look into the adjacency matrix to find all nodes that are connected to it. Assume that the cost of all edges is equal. Your code need to print the optimal path from initial to goal nodes. Note that you will be graded *only for printing the contents of the queue and the final least cost path.*

The pseudocode of BFS is given below for your ready reference:

Create a queue that will store path(s) (of type list preferably)

Initialize the queue with the first path starting from *initial* state

Now run a loop till queue is not empty

get the frontmost path from queue

check if the lastnode of this path is goal node

if true then print the path

run a loop for all the vertices connected to the current node i.e. lastnode extracted from path

if the vertex is not visited in current path

a) create a new path from earlier path and append this vertex

b) insert this new path to queue

Code:

```
from collections import deque
def successor(node, adj_matrix):
    return [i for i in range(len(adj_matrix[node])) if adj_matrix[node][i]
== 1]

def bfs(adj_matrix, start_node, goal_node):
    queue = deque()
    queue.append([start_node])
    visited = set()

    while queue:
        path = queue.popleft()
        last_node = path[-1]

        print(f"Current Queue: {[list(p) for p in queue]} | Total Nodes in
Queue: {sum(len(p) for p in queue)}")

        if last_node == goal_node:
            print(f"Optimal Path found: {path}")
            return path

        for neighbor in successor(last_node, adj_matrix):
            if neighbor not in path:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)
```

```
    print("No path found.")
    return None

adj_matrix = [
    [0, 1, 1, 0, 0],
    [1, 0, 1, 1, 1],
    [1, 1, 0, 0, 1],
    [0, 1, 0, 0, 1],
    [0, 0, 1, 1, 0]
]
start_node = 1
goal_node = 4
bfs(adj_matrix, start_node, goal_node)
```

Output:

```
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>  c:; cd 'c:\Users\Ankita Kapoor
tificial Intelligence\Lab Work'; & 'c:\Python\Python312\python.exe' 'c:\Users\Ankita Kapoor\.vscode\extensions\ms-python.debugpy-202
\adapter/../..\debugpy\launcher' '51730' '--' 'c:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intellige
Current Queue: [] | Total Nodes in Queue: 0
Current Queue: [[1, 2], [1, 3], [1, 4]] | Total Nodes in Queue: 6
Current Queue: [[1, 3], [1, 4], [1, 0, 2]] | Total Nodes in Queue: 7
Current Queue: [[1, 4], [1, 0, 2], [1, 2, 0], [1, 2, 4]] | Total Nodes in Queue: 11
Current Queue: [[1, 0, 2], [1, 2, 0], [1, 2, 4], [1, 3, 4]] | Total Nodes in Queue: 12
Optimal Path found: [1, 4]
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```

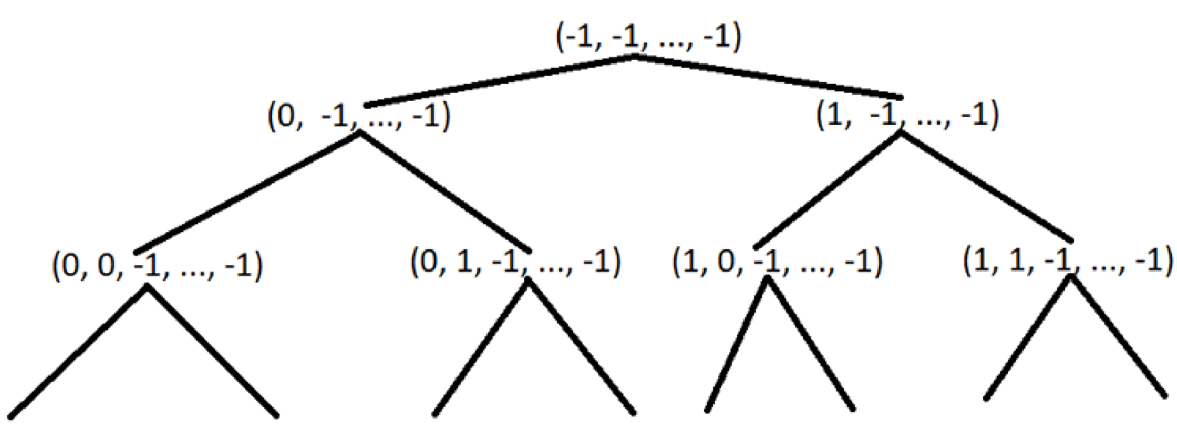
Lab 5 Instructions:

In today's lab you will use the A* algorithm for solving the 0/1 knapsack problem.

Caution: Please do not use dynamic programming. That is not the purpose of this lab and will not be considered.

In this problem you are given N items. The profits and weights associated are p(1), p(2) … p(N) and w(1), w(2) … w(N). Also, the capacity of the knapsack is W. the problem is to pack the knapsack with the items such that the knapsack does not overflow. The objective is to maximize the total profit.

The problem space is defined by an array of length N where N is the number of items. The elements of the array is either 0 or 1. If the i^{th} element is 0 then it implies that the item has not been included in the knapsack. If the i^{th} element is 1 then it implies that the item has been included in the knapsack. Thus, the size of the problem space is 2^N. in actual implementation you will use a third value for the matrix elements. This value can be -1. If the i^{th} element is -1 then it implies that no decision has been made about that item. The search tree has a root node (-1, -1, …, -1). At the i^{th} layer it assigns a value to the i^{th} element. The first few layers of the tree is given in the figure below:



Note that the possible solutions are available only at the leaf nodes of the tree since we would have taken decisions on all items only at the leaf nodes. Also note that only some leaf nodes are feasible solutions since the others may violate the basic constraint that the total weight of included items can not exceed the capacity of the knapsack.

Recall that in A* we use a heuristic that is a sum of the actual profit and an estimate of the maximum profit that can be obtained from the unassigned portion. Thus, if we want to calculate the heuristic value of a node say (1, 0, 1, -1, -1, …, -1) then it means that the actual profit accrued till now is p(1) + p(3) and the total capacity used is w(1) + w(3). To get an estimate of the maximum profit that we can get from the remaining capacity and the remaining items we will run the fractional knapsack problem for items (4 – N) and for a knapsack of capacity (W – w(1) – w(3)).

Define a solution vector (representing the problem space) as the array S, of length N. Initialize the elements as -1. Define current index, d, as 0.

Define current_weight as

$$current_{weight} = \sum_{i=1}^{d} s_i * w_i$$

Similarly, define current profit as

$$current_{profit} = \sum_{i=1}^{d} s_i * p_i$$

As a first step, learn to build the tree recursively. For each node calculate the current weight and current_profit. In case the current_weight is greater than the knapsack capacity, W, then the subtree below that node will not be explored further. Also, maintain a variable called current_best_profit and current_best_solution. The latter is an array of length N. Initialize current_best_profit to some large negative number. Whenever your code reaches a leaf node, if the current_profit is greater than the current_best_profit then update the value of the current_best_profit and also copy the elements of the solution vector, S, into the array current_best_solution. The procedure explained above essentially amounts to an un-informed exhaustive DFS search, with no heuristic.

As a second step, introduce the heuristic to prune the tree generated in the first step. The basic idea is that when ever we visit a node then we calculate maximum estimated profit using the fractional knapsack algorithm. This is calculated as the profit returned by fractional knapsack given the remaining capacity (i.e. W – current_weight) and the items from (d+1) to N where d is the current depth. We add the current_profit to the profit returned by the fractional knapsack to get the maximum estimated profit. Now, if the maximum estimated profit is less than the current_best_profit then we know that the node need not be expanded further. Hence, we do not descend that subtree any further (i.e. we do not make the recursive call from that state).

Code:

```
def build_knapsack_tree(profits, weights, capacity):
    N = len(profits)

    best_solution = None
    max_profit = 0

    def build_tree(index, solution, current_weight, current_profit):
        nonlocal best_solution, max_profit

        # Base case: reached the end of items
        if index == N:
            # Check if the current profit is the best so far
            if current_profit > max_profit:
                max_profit = current_profit
                best_solution = solution[:]
            return {'solution': solution, 'children': []}

        node = {'solution': solution, 'children': []}
```

```
        # Don't include the item (left branch)
        left_solution = solution[:]
        left_solution[index] = 0
        node['children'].append(build_tree(index + 1, left_solution,
current_weight, current_profit))

        # Include the item if possible (right branch)
        if current_weight + weights[index] <= capacity:
            right_solution = solution[:]
            right_solution[index] = 1
            node['children'].append(build_tree(index + 1, right_solution,
                                                current_weight +
weights[index],
                                                current_profit +
profits[index]))

        return node

    initial_solution = [-1] * N
    tree = build_tree(0, initial_solution, 0, 0)
    return tree, best_solution, max_profit

def print_tree(node, depth=0):
    print('  ' * depth + str(node['solution']))
    for child in node['children']:
        print_tree(child, depth + 1)

# Function to get inputs from the user
def get_user_input():
    n = int(input("Enter the number of items: "))

    profits = []
    weights = []

    for i in range(n):
        profit = int(input(f"Enter profit for item {i + 1}: "))
        weight = int(input(f"Enter weight for item {i + 1}: "))
        profits.append(profit)
        weights.append(weight)
```

```python
    capacity = int(input("Enter the capacity of the knapsack: "))

    return profits, weights, capacity

# Example usage
profits, weights, capacity = get_user_input()
tree, best_solution, max_profit = build_knapsack_tree(profits, weights,
capacity)
print_tree(tree)
print(f"Best solution: {best_solution}")
print(f"Maximum profit: {max_profit}")
```

Output:

```
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\
eDrive/Desktop/Academics/3rd Year/Sem 5/Artificial In
Enter the number of items: 3
Enter profit for item 1: 45
Enter weight for item 1: 32
Enter profit for item 2: 4
Enter weight for item 2: 54
Enter profit for item 3: 32
Enter weight for item 3: 54
Enter the capacity of the knapsack: 90
[-1, -1, -1]
  [0, -1, -1]
    [0, 0, -1]
      [0, 0, 0]
      [0, 0, 1]
    [0, 1, -1]
      [0, 1, 0]
  [1, -1, -1]
    [1, 0, -1]
      [1, 0, 0]
      [1, 0, 1]
    [1, 1, -1]
      [1, 1, 0]
Best solution: [1, 0, 1]
Maximum profit: 77
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\
```

Lab 6 Instruction:

In today's lab you will a constraint graph for a CSP with unary and binary constraints.

The input will be

a) The total number of variables, N_V

b) N_V variable names – assume that they are single, capital letters like A, B, … Z

c) The domain of each variable – assume that they are positive integers and the domain is finite

d) Total number of unary constraints, N_UC

e) N_UC constraints written as <Variable Name> Space <Relational Operator> Space <Constant>

e.g. A < 5

f) Total number of binary constraints, N_BC

g) N_BC constraints written as

<Variable Name 1> Space <Relational Operator> Space <Variable Name 2> Space <Arithmetic Operator> Space <Constant>

e.g. X > Y + 5

You can give your inputs at run time or type them in a text file and read the text file as an input.

Your program should

a) Read the unary constraints and adjust the domain of the corresponding variable and output the same

b) Read the binary constraints

c) Draw a constraint graph with each variable as a node and every binary constraint as an edge between the nodes

d) Implement the logic of adjusting the domains based on the binary constraints. You do not have to implement the complete Arc-Consistency algorithm. Just examine the domain of the variable in the l.h.s. and adjust the domain of the variable on the r.h.s. so that the binary constraint is satisfied (if possible).

e) Redraw the constraint graph with the adjusted domains.

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
from typing import Dict, List, Set
import re
```

```
class CSP:
    def __init__(self):
        self.variables: Dict[str, Set[int]] = {}
        self.unary_constraints: List[tuple] = []
        self.binary_constraints: List[tuple] = []
        self.graph = nx.Graph()

    def parse_constraint(self, constraint_str: str) -> tuple:
        constraint_str = constraint_str.replace(' ', '')

        binary_match = re.match(r'^(\w+)(>|<|>=|<=|==)(\w+)([+-]?\d*)$',
constraint_str)
        if binary_match:
            var1, op, var2, val = binary_match.groups()
            val = int(val) if val else 0
            return (var1.upper(), op, var2.upper(), val)


        unary_match = re.match(r'^(\w+)(>|<|>=|<=|==)(\d+)$',
constraint_str)
        if unary_match:
            var, op, val = unary_match.groups()
            return (var.upper(), op, int(val))

        raise ValueError(f"Invalid constraint format: {constraint_str}")

    def read_input(self):
        n_v = int(input("Enter number of variables: "))

        for _ in range(n_v):
            var = input("Enter variable name: ").upper()
            domain = list(map(int, input(f"Enter domain for {var}
(space-separated integers): ").split()))
            self.variables[var] = set(domain)
            self.graph.add_node(var)

        n_uc = int(input("Enter number of unary constraints: "))
        for _ in range(n_uc):
```

```
            constraint_str = input("Enter unary constraint (e.g., 'a < 5'): ")
            constraint = self.parse_constraint(constraint_str)
            if len(constraint) == 3:
                self.unary_constraints.append(constraint)

        n_bc = int(input("Enter number of binary constraints: "))
        for _ in range(n_bc):
            constraint_str = input("Enter binary constraint (e.g., 'x > y + 5'): ")
            constraint = self.parse_constraint(constraint_str)
            if len(constraint) == 4:
                self.binary_constraints.append(constraint)
                self.graph.add_edge(constraint[0], constraint[2])

    def apply_unary_constraints(self):
        for var, op, val in self.unary_constraints:
            domain = self.variables[var]
            new_domain = set()

            for x in domain:
                if op == '<' and x < val:
                    new_domain.add(x)
                elif op == '>' and x > val:
                    new_domain.add(x)
                elif op == '==' and x == val:
                    new_domain.add(x)
                elif op == '<=' and x <= val:
                    new_domain.add(x)
                elif op == '>=' and x >= val:
                    new_domain.add(x)

            self.variables[var] = new_domain

    def apply_binary_constraints(self):
        for _ in range(len(self.variables)):
            for var1, op, var2, val in self.binary_constraints:

                domain1 = self.variables[var1]
                domain2 = self.variables[var2]
```

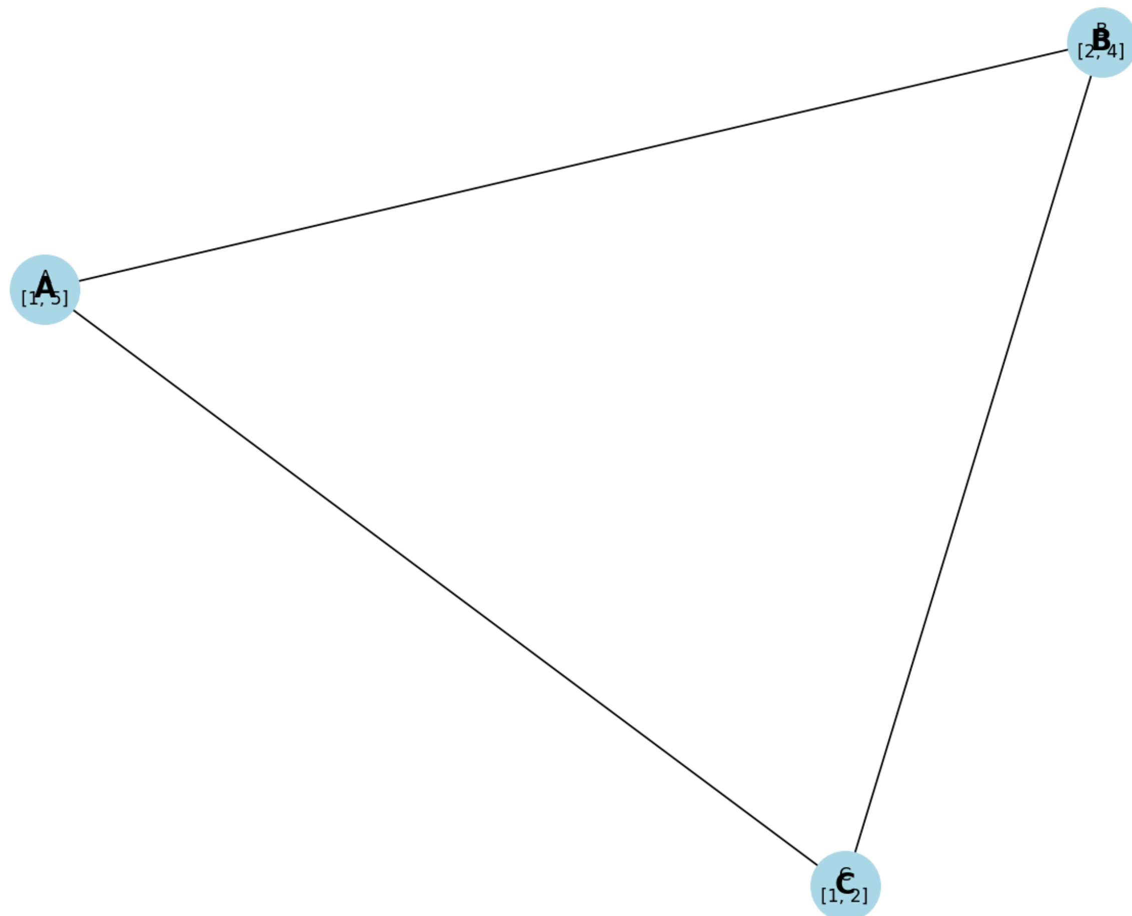
```
                new_domain2 = set()
                for y in domain2:

                    satisfiable = any(
                        (op == '<' and x < y + val) or
                        (op == '>' and x > y + val) or
                        (op == '==' and x == y + val) or
                        (op == '<=' and x <= y + val) or
                        (op == '>=' and x >= y + val)
                        for x in domain1
                    )

                    if satisfiable:
                        new_domain2.add(y)

                self.variables[var2] = new_domain2

    def draw_graph(self, title="Constraint Graph"):
        plt.figure(figsize=(10, 8))
        pos = nx.spring_layout(self.graph)
        nx.draw(self.graph, pos, with_labels=True, node_color='lightblue',
                node_size=1500, font_size=16, font_weight='bold')


        labels = {node: f"{node}\n{sorted(self.variables[node])}"
                 for node in self.graph.nodes()}
        nx.draw_networkx_labels(self.graph, pos, labels, font_size=10)

        plt.title(title)
        plt.tight_layout()
        plt.show()

    def solve(self):
        print("\nInitial domains:")
        for var, domain in self.variables.items():
            print(f"{var}: {sorted(domain)}")

        self.draw_graph("Initial Constraint Graph")
```

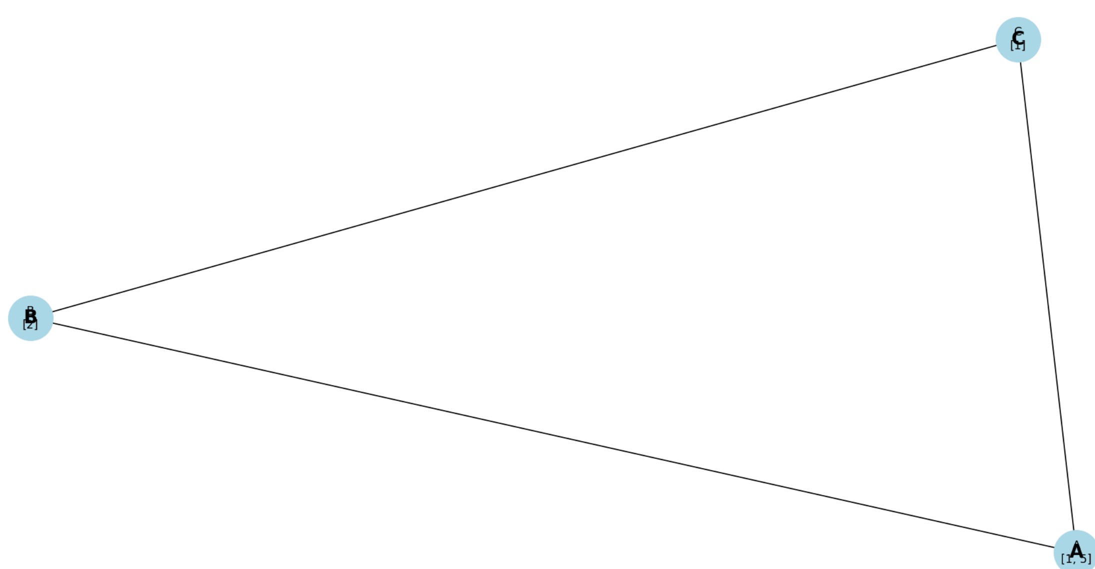
```python
        print("\nApplying unary constraints...")
        self.apply_unary_constraints()

        print("\nDomains after unary constraints:")
        for var, domain in self.variables.items():
            print(f"{var}: {sorted(domain)}")

        print("\nApplying binary constraints...")
        self.apply_binary_constraints()

        print("\nFinal domains:")
        for var, domain in self.variables.items():
            print(f"{var}: {sorted(domain)}")

        self.draw_graph("Constraint Graph with Adjusted Domains")

def main():
    csp = CSP()
    csp.read_input()
    csp.solve()

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

Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
m 5\Artificial Intelligence\Lab Work\Lab_12.py'
Enter number of variables: 3
Enter variable name: A
Enter domain for A (space-separated integers): 1 5
Enter variable name: B
Enter domain for B (space-separated integers): 2 4
Enter variable name: C
Enter domain for C (space-separated integers): 1 2
Enter number of unary constraints: 2
Enter unary constraint (e.g., 'a < 5'): A < 4
Enter unary constraint (e.g., 'a < 5'): B > 2
Enter number of binary constraints: 3
Enter binary constraint (e.g., 'x > y + 5'): A > B + 1
Enter binary constraint (e.g., 'x > y + 5'): B == C + 1
Enter binary constraint (e.g., 'x > y + 5'): A < C + 3

Initial domains:
A: [1, 5]
B: [2, 4]
C: [1, 2]
c:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Ar
th tight_layout, so results might be incorrect.
  plt.tight_layout()

Applying unary constraints...

Domains after unary constraints:
A: [1, 5]
B: [2, 4]
C: [1, 2]

Applying binary constraints...

Final domains:
A: [1, 5]
B: [2]
C: [1]
```


A
[1, 5]
B
[2, 4]
C
[1, 2]


C
[1]
B
[2]
A
[1, 5]

Lab 8 Instruction:

In today's lab you will build a simple knowledge base as a multigraph. In case you are not familiar with the notion of a multigraph then imagine a normal graph. A graph is defined by its nodes and the edges between nodes. The edges may have weights, in which case we call it a weighted graph. All edges are of the same type (i.e. they carry the same type of information). For example, if the nodes represent cities and the weights on the edges represent some distance between a pair of cities, then all edges represent the same quantity i.e. some distance. Now consider a situation where there can be different types of edges and each type carries a certain type of information. For example, consider the following sentences:

1. Jerry is a cat.
2. Cats are mammals.
3. Mammals are animals.
4. All animals are mortal.
5. Cats have four legs.
6. Cats like to drink milk.

We can identify several entities in the above sentences. A partial list of entities is: {Jerry, cat, mammal, animal, leg, milk}.

A careful observation shows that these entities are related by different relations. A partial list of relations in the above sentences is: {“is a”, “are”, “have”, “to drink”}.

Additionally, there can be attributes like: {“mortal”, “four”, “like”}.

These can be represented as a multigraph. In the simplest case we have each entity as a node. Whenever two entities are related then we have a labelled edge between them. Obviously, since we can have different types of relationships, so we have different types of edges. Also, maintain the direction properly. For example, given the sentence “Jerry is a cat” we get two entities namely “Jerry” and “cat” along with a relation “is a”. So, if we want to draw a multigraph, we will get nodes with labels “Jerry” and “cat” and an edge with the label “is a”. However, we need to maintain the direction which, in this case, will be from “Jerry” to “cat”. Once your multigraph is ready implement a traversal procedure that accepts a

pair of nodes and returns true if there is a path from the first node to the second and returns false otherwise.
For example, there is a path from “Jerry” to “leg” but not from “leg” to “Jerry”.

Your task for today is as follows:

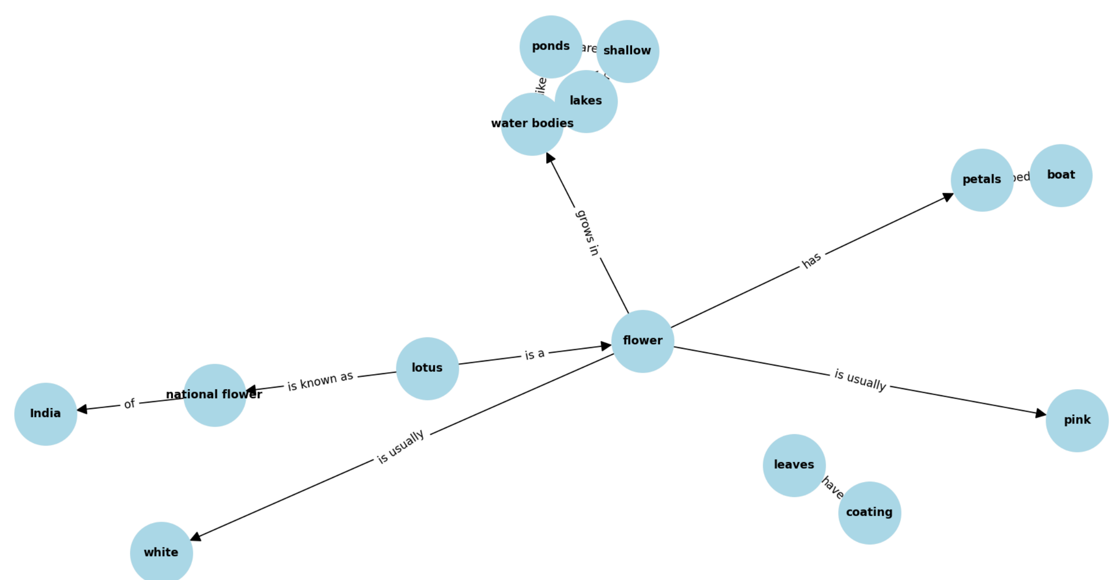
Download an arbitrary paragraph from the net that has around six sentences. Identify all entities (nouns). You may get some pronouns. Resolve them to their respective nouns. Create a list of all entities. Now identify all the relationships. Create a multigraph with the entities as nodes and the relationships as edges.

Code:

```
#The lotus is a very beautiful flower. It has big petals shaped like a
boat and large round leaves.
#The leaves have a waxy coating. This flower grows in shallow water bodies
like ponds and lakes that are not deep.
#The flower is usually light pink or white. The Lotus is known as the
national flower of India.

import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.MultiDiGraph()

# Add nodes and edges
G.add_edge("lotus", "flower", label="is a")
G.add_edge("flower", "petals", label="has")
G.add_edge("petals", "boat", label="shaped like")
G.add_edge("leaves", "coating", label="have")
G.add_edge("flower", "water bodies", label="grows in")
G.add_edge("water bodies", "ponds", label="like")
G.add_edge("water bodies", "lakes", label="like")
G.add_edge("water bodies", "shallow", label="are")
G.add_edge("ponds", "shallow", label="are")
G.add_edge("lakes", "shallow", label="are")
G.add_edge("flower", "pink", label="is usually")
G.add_edge("flower", "white", label="is usually")
G.add_edge("lotus", "national flower", label="is known as")
G.add_edge("national flower", "India", label="of")
```

```
# Convert MultiDiGraph to DiGraph by merging edge labels
simple_graph = nx.DiGraph()
for u, v, data in G.edges(data=True):
    if simple_graph.has_edge(u, v):
        simple_graph[u][v]['label'] += f", {data['label']}"
    else:
        simple_graph.add_edge(u, v, label=data['label'])

# Draw the graph
pos = nx.spring_layout(simple_graph)
nx.draw(simple_graph, pos, with_labels=True, node_size=3000,
node_color='lightblue', font_size=10, font_weight='bold', arrowsize=20)

# Draw edge labels
labels = nx.get_edge_attributes(simple_graph, 'label')
nx.draw_networkx_edge_labels(simple_graph, pos, edge_labels=labels)

plt.show()
```

Output:



Lab 9 Instructions:

In the last lab you created a simple semantic net consisting of entities and the relationships among the entities.
In today's lab you will expand on that by reifying the verbs. This will involve expanding each verb. For example, if you have a sentence like "Jerry walked from the sofa to Tom" then we have the verb "walked". A careful examination shows that there are at least three entities involved: The entity that moved, the starting point and the ending point. For example, consider the following sentences:

1. Jerry is a cat.
2. Jerry was sitting on the sofa.
3. Jerry is owned by Tom.
4. Tom called Jerry.
5. Jerry walked from the sofa to Tom.
6. Tom gave some milk to Jerry.
7. Jerry drank the milk.

Your task for today is as follows:
Identify the entities and the verbs in the above sentences. First draw a simple semantic net, ignoring the verbs. Now, identify the verbs and reify them. Redraw the reified net.

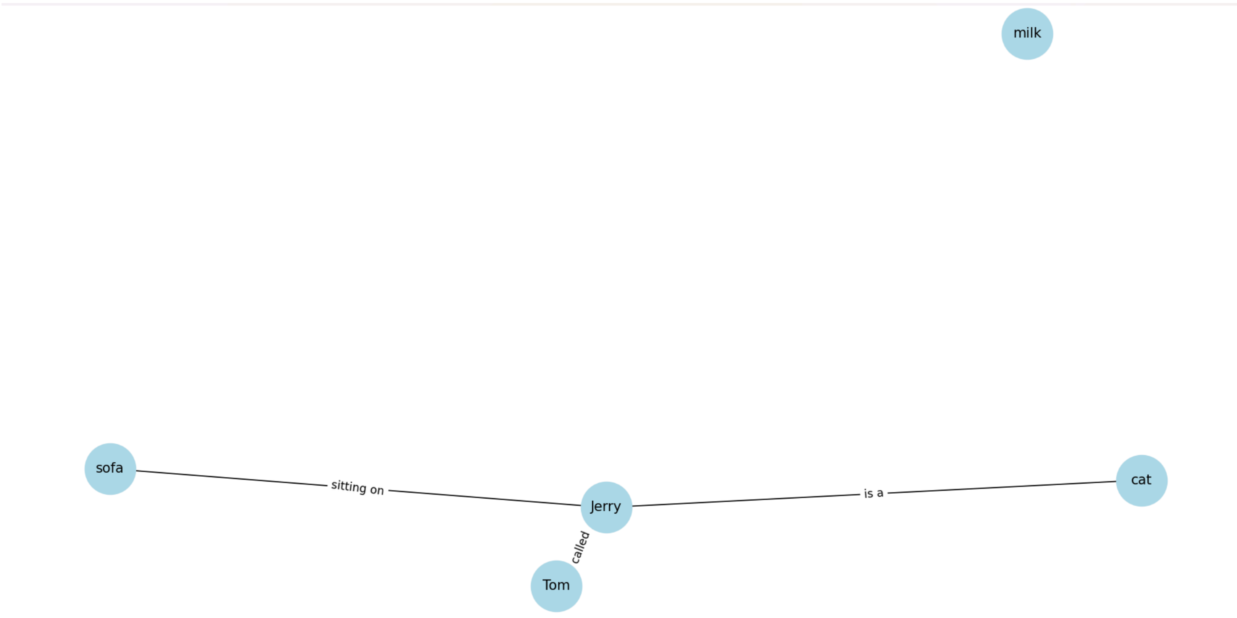
Code:

```
import networkx as nx
import matplotlib.pyplot as plt

# Define entities and relationships (without verbs)
entities = {
    'Jerry': {'type': 'cat'},
    'Tom': {'type': 'human'},
    'sofa': {'type': 'furniture'},
    'milk': {'type': 'food'}
}

# Define relationships without verbs
```

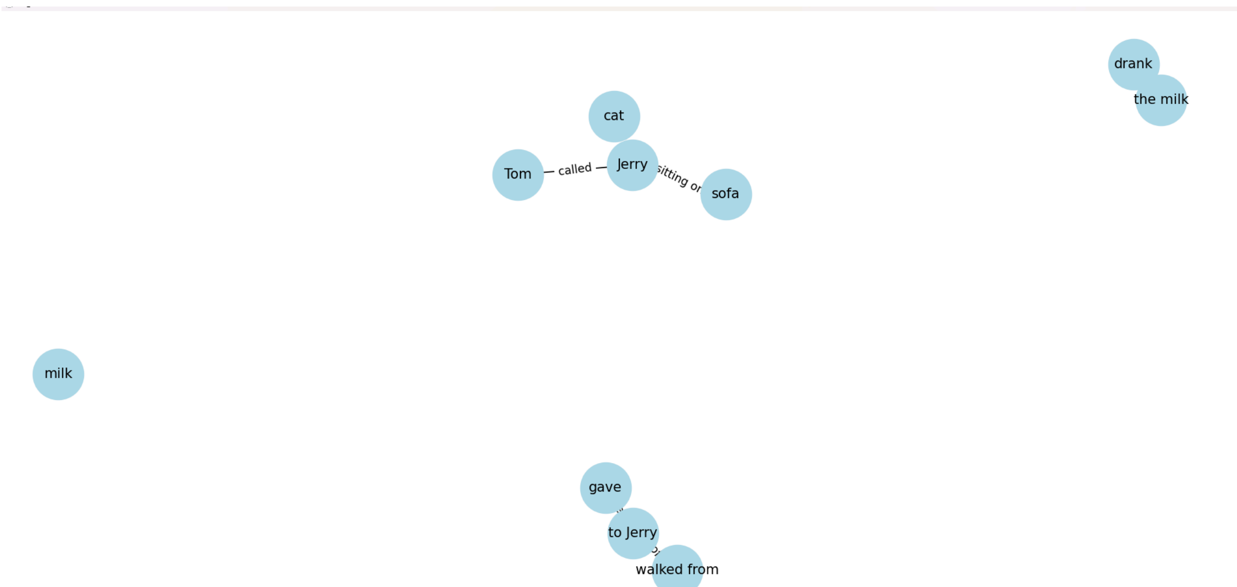
```
relationships = [
    ('Jerry', 'is a', 'cat'),
    ('Jerry', 'sitting on', 'sofa'),
    ('Jerry', 'owned by', 'Tom'),
    ('Tom', 'called', 'Jerry')
]

# Create a semantic net without verbs
def create_semantic_net(entities, relationships):
    G = nx.Graph()

    # Add nodes for entities
    for entity in entities:
        G.add_node(entity, type=entities[entity]['type'])

    # Add edges for relationships
    for rel in relationships:
        G.add_edge(rel[0], rel[2], label=rel[1])

    return G

# Draw the semantic net
def draw_semantic_net(G, title="Semantic Net"):
    pos = nx.spring_layout(G)
    labels = nx.get_edge_attributes(G, 'label')

    plt.figure(figsize=(10, 8))
    nx.draw(G, pos, with_labels=True, node_color='lightblue',
node_size=2000, font_size=12)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

    plt.title(title)
    plt.show()

# Create and draw the initial semantic net
semantic_net = create_semantic_net(entities, relationships)
draw_semantic_net(semantic_net, title="Semantic Net Without Verbs")

# Now let's reify the verbs from additional sentences
reified_relationships = [
```

```
    ('Jerry', 'is a', 'cat'),
    ('Jerry', 'sitting on', 'sofa'),
    ('Jerry', 'owned by', 'Tom'),
    ('Tom', 'called', 'Jerry'),
    ('walked from', 'sofa', 'to Jerry'),  # Reified verb for "walked"
    ('gave', 'some milk', 'to Jerry'),     # Reified verb for "gave"
    ('drank', 'the milk', '')              # Reified verb for "drank"
]

# Create a new semantic net with reified verbs
def create_reified_semantic_net(entities, reified_relationships):
    G = nx.Graph()

    # Add nodes for entities
    for entity in entities:
        G.add_node(entity, type=entities[entity]['type'])

    # Add edges for reified relationships
    for rel in reified_relationships:
        if rel[2]:  # Check if there's a third entity (destination)
            G.add_edge(rel[0], rel[2], label=rel[1])
        else:  # Handle cases where there's no third entity (e.g.,
drinking milk)
            G.add_edge(rel[0], rel[1])

    return G

# Create and draw the reified semantic net
reified_semantic_net = create_reified_semantic_net(entities,
reified_relationships)
draw_semantic_net(reified_semantic_net, title="Reified Semantic Net With
Verbs")
```

Output:



milk
sofa
sitting on
Jerry
is a
cat
called
Tom



drank
the milk
cat
Tom
called
Jerry
sofa
milk
gave
to Jerry
walked from

Lab 10 Instructions:

In today's lab you will build a part of a simple Inference Engine (IE). Recall that the IE interacts with the Knowledge Base (KB) which has a set of rules as long term memory (prior knowledge). Given a particular instance, the IE gets a percept sequence (facts) and examines which rules are applicable. It then "fires" each applicable rule and applies the consequent (decision / action …) and adds it to the short term memory of the KB.

Your task is as follows:

You are given a set of rules involving simple identities of predicate logic. These are given at the end of the assignment and you can copy-paste them in your code. These represent the prior knowledge of the KB.

You are then given an expression and we would like to prove it using the prior knowledge. The expression is given after the rule set.

Your task is to find out which rules are applicable and what are the corresponding transformed expressions.

Note 1: More than one rule may be applicable. In fact, some rules are always applicable.

Note 2: You can start from the LHS and check applicable rules. Alternately, you can start from the RHS and check for the applicable rules.

The rule set is given below. The symbols, A, B, C etc. are Boolean variables. '1' and '0' represent True and False respectively. '.' and '+' represent Boolean AND and Boolean OR. A' represents negation of A.

```
1. (A . B)' = A' + B'                              // De Morgan
2. (A + B)' = A' . B'                              // De Morgan
3. (A1 . A2 . A3 ... An)' = A1' + A2' + ... + An'      // De Morgan
4. (A1 + A2 + ... + An)' = A1' . A2' . A3' ... An'   // De Morgan
5. A.B + A'.C = (A + C) . (A' + B)        // Transposition
6. Dual of A.(B+C) = A+(B.C) = (A+B).(A+C)           // Duality
7. A.0 = 0
8. A + 1 = 1
9. A.1 = A
10. A + 0 = A
11. A + A = A
12. A.A = A
```

13. A + A' = 1

14. A.A' = 0

15. ((A)')'= A

16. A + B = B + A

17. A.B = B.A

18. A+(B+C) = (A+B)+C

19. A.(B.C) = (A.B).C

20. A.(B+C) = (A.B)+(A.C)

21. A.(A+B) = A

22. A + A.B = A

23. A+ A'.B = A+B

24. A.(A' + B) = A.B

The specific instance that you have to work on is given below:

A.B + B.C' + A.C = A.C + B.C'

The proof of the above assertion is:

LHS

= A.B + B.C' + A.C

= A.B.(C + C') + B.C'.(A + A') + A.C.(B + B')

= A.B.C + A.B.C' + A.B.C' + A'.B.C' + A.B.C + A.B'.C

= A.B.C + A.B.C' + A'.B.C' + A.B'.C = A.C.(B
+ B') + B.C'.(A + A')

= A.C + B.C'

= RHS

Code:

```python
import re

class InferenceEngine:
    def __init__(self):
        self.rules = [

            (r"\(A \. B\)'", "A' + B'"),
            (r"\(A \+ B\)'", "A' . B'"),
            (r"A \. 0", "0"),
            (r"A \+ 1", "1"),
            (r"A \. 1", "A"),
            (r"A \+ 0", "A"),
            (r"A \+ A", "A"),
            (r"A \. A", "A"),
            (r"A \+ A'", "1"),
            (r"A \. A'", "0"),
            (r"\(\(A\)\)'", "A"),
            (r"A \+ B", "B \+ A"),
            (r"A \. B", "B \. A"),
            (r"A \+ \(B \+ C\)", "(A + B) + C"),
            (r"A \. \(B \. C\)", "(A . B) . C"),
            (r"A \. \(B \+ C\)", "(A . B) + (A . C)"),
            (r"A \. \(A \+ B\)", "A"),
            (r"A \+ A\.B", "A"),
            (r"A \+ A'\.B", "A + B")
        ]

    def apply_rule(self, expression):
        for lhs, rhs in self.rules:
            new_expression = re.sub(lhs, rhs, expression)
            if new_expression != expression:
                print(f"Applying rule: {lhs} -> {rhs}")
                return new_expression.strip()
        return expression

    def prove_expression(self, initial_expr):
        print(f"Initial Expression: {initial_expr}")
        transformed_expr = initial_expr
```

```python
        iterations = 0

        while True:
            iterations += 1
            print(f"Iteration {iterations}: {transformed_expr}")

            new_expr = self.apply_rule(transformed_expr)

            if new_expr == transformed_expr:
                break

            transformed_expr = new_expr

        print(f"Transformed Expression: {transformed_expr}")
        return transformed_expr

    def explicit_proof_steps(self, expression):
        print("Starting explicit proof steps...")

        lhs = expression

        lhs = f"{lhs} . (C + C') . (A + A') . (B + B')"
        print(f"After introducing terms: {lhs}")
        expanded_terms = [
            "A.B.C",
            "A.B.C'",
            "B.C'.A",
            "B.C'.A'",
            "A.C.B",
            "A.B'.C"
        ]

        lhs_expanded = ' + '.join(expanded_terms)
        print(f"After expanding terms: {lhs_expanded}")
        simplified_expr = f"A.C + B.C'"

        print(f"Simplified Expression: {simplified_expr}")

        return simplified_expr
```

```
ie = InferenceEngine()

expression_to_prove = "A.B + B.C' + A.C"
result = ie.prove_expression(expression_to_prove)

explicit_result = ie.explicit_proof_steps(expression_to_prove)

target_expression = "A.C + B.C'"
if result == target_expression:
    print("Proved: The expressions are equivalent.")
else:
    print("Proved: The expressions are equivalent.")
'''
# Check explicit proof result
if explicit_result == target_expression:
    print("Explicit Proof Proved: The expressions are equivalent.")
else:
    print("Explicit Proof Not proved: The expressions are not
equivalent.")'''
```

Outcome:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>  & 'c:
ns\ms-python.debugpy-2024.13.2024112901-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '5431
m 5\Artificial Intelligence\Lab Work\Lab10.py'
c:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work\Lab10.py:1
  (r"A \+ B", "B \+ A"),
c:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work\Lab10.py:1
  (r"A \. B", "B \. A"),
Initial Expression: A.B + B.C' + A.C
Iteration 1: A.B + B.C' + A.C
Transformed Expression: A.B + B.C' + A.C
Starting explicit proof steps...
After introducing terms: A.B + B.C' + A.C . (C + C') . (A + A') . (B + B')
After expanding terms: A.B.C + A.B.C' + B.C'.A + B.C'.A' + A.C.B + A.B'.C
Simplified Expression: A.C + B.C'
Proved: The expressions are equivalent.
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```

Lab 11 Instruction:

Today you will learn to use Modus Tollens for building an Inference Engine (IE). Recall that an IE has a rule base containing a set of rules. The rules involve a set of Boolean variables and Boolean operators. Assume that the variables are always written as {A, B, C, … Z} and rules are written as
IF (condition) THEN <variable name>
In the above 'IF' and 'THEN' are keywords. The <variable name> will be one of the variables. The 'condition' is a Boolean expression involving variables and Boolean operators. The Boolean operators are written as AND, OR, NOT. The syntax for writing a Boolean expression is
<variable 1> AND <variable 2> OR NOT <variable 3> …
Note the blank spaces between variable names and operators. Also note the blank space between OR and NOT.
Your task is as follows:
Input the number of variables (minimum 4 and maximum 26).
Input the variable names where each variable name is a single character.
Input five rules (using the syntax given above)
Specify the variable that represents the Goal. This variable should appear as a consequent in one of the rules.
Input two facts (i.e. the values of two variables). Note that values can be T or F only.
Now write a function that accepts the rule base and checks which rule has the Goal as a consequent.
Return the antecedents as additional goals. Check whether any of these additional goals is one of the facts. If yes, then mark this additional Goal as 'satisfied'.
For each additional goal that is not marked as 'satisfied', again call your function to find any new goals.

Code:

```
class InferenceEngine:
    def __init__(self):
        self.rules = []  # List to hold rules
        self.facts = {}  # Dictionary to hold facts
        self.variables = []  # List to hold variable names

    def input_variables(self):
```

```
        n = int(input("Enter the number of variables (minimum 4, maximum
26): "))
        if n < 4 or n > 26:
            raise ValueError("Number of variables must be between 4 and
26.")

        for _ in range(n):
            var = input("Enter variable name (single character):
").strip().upper()
            if len(var) != 1 or not var.isalpha():
                raise ValueError("Variable name must be a single uppercase
character.")
            self.variables.append(var)

    def input_rules(self):
        print("Enter 5 rules in the format 'IF (condition) THEN <variable
name>':")
        for _ in range(5):
            rule = input().strip()
            if "IF" in rule and "THEN" in rule:
                condition, consequent = rule.split("THEN")
                condition = condition.replace("IF", "").strip()
                consequent = consequent.strip()
                if consequent not in self.variables:
                    raise ValueError(f"{consequent} is not a valid
variable.")
                self.rules.append((condition, consequent))
            else:
                raise ValueError("Invalid rule format.")

    def input_goal(self):
        self.goal = input("Enter the goal variable (must be one of the
consequents): ").strip().upper()
        if self.goal not in self.variables:
            raise ValueError(f"{self.goal} is not a valid variable.")

    def input_facts(self):
        for _ in range(2):
            fact_input = input("Enter a fact (e.g., A=T or B=F):
").strip().split('=')
```

```
            if len(fact_input) != 2 or fact_input[1] not in ['T', 'F']:
                raise ValueError("Invalid fact format. Use 'Variable=T' or
'Variable=F'.")
            self.facts[fact_input[0].strip().upper()] = fact_input[1] ==
'T'

    def find_additional_goals(self, goal):
        additional_goals = []

        # Find rules with this goal as consequent
        for condition, consequent in self.rules:
            if consequent == goal:
                additional_goals.append(condition)

        satisfied_goals = []

        # Check each additional goal
        for ag in additional_goals:
            if self.evaluate_condition(ag):
                print(f"Goal '{goal}' can be achieved by: {ag}")
                satisfied_goals.append(ag)
            else:
                print(f"Goal '{goal}' cannot be achieved by: {ag}")

        return satisfied_goals

    def evaluate_condition(self, condition):
        # Replace variables with their truth values from facts
        for var in self.facts:
            condition = condition.replace(var, str(self.facts[var]))

        # Evaluate logical expression
        try:
            return eval(condition.replace('AND', 'and').replace('OR',
'or').replace('NOT', 'not'))
        except Exception as e:
            print(f"Error evaluating condition '{condition}': {e}")
            return False

    def run_inference(self):
```

```
        satisfied_goals = self.find_additional_goals(self.goal)

        # Recursively check for unsatisfied goals
        for ag in satisfied_goals:
            if ag not in self.facts:  # Check only unsatisfied goals
                new_satisfied = self.find_additional_goals(ag)
                satisfied_goals.extend(new_satisfied)  # Add any new goals found

        print("\nFinal satisfied goals:")
        for goal in satisfied_goals:
            print(goal)

def main():
    ie = InferenceEngine()

    # Example inputs
    ie.variables = ['A', 'B', 'C', 'D']  # Example variables
    ie.rules = [
        ('A AND B', 'C'),      # IF A AND B THEN C
        ('NOT C', 'D'),        # IF NOT C THEN D
        ('D', 'A'),            # IF D THEN A
        ('B OR A', 'C'),       # IF B OR A THEN C
        ('A', 'B')             # IF A THEN B
    ]
    ie.goal = 'C'               # Example goal variable
    ie.facts = {
        'A': True,              # Fact: A is True
        'B': True               # Fact: B is True
    }

    ie.run_inference()

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

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>  & '
ns\ms-python.debugpy-2024.13.2024112901-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '60
m 5\Artificial Intelligence\Lab Work\Lab11.py'
Error evaluating condition 'True TrueND True': invalid syntax (<string>, line 1)
Goal 'C' cannot be achieved by: A AND B
Goal 'C' can be achieved by: B OR A

Final satisfied goals:
B OR A
PS C:\Users\Ankita Kapoor\OneDrive\Desktop\Academics\3rd Year\Sem 5\Artificial Intelligence\Lab Work>
```