VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM 590014

Laboratory Report on

Artificial Intelligence (22CS5PCAIN)

Submitted By

Ananya Aithal (1BM21CS259)

Under the Guidance of

Dr. Panimozhi N
Assistant Professor

Department of Computer Science and Engineering
B.M.S. College of Engineering
(Autonomous college under VTU)
P.O. Box No.: 1908, Bull Temple Road, Bangalore-560019

2023-202

CERTIFICATE

This is to certify that the Laboratory Report on Artificial Intelligence (22CS5PCAIN) has been successfully carried out by **Ananya Aithal (1BM21CS259)** during the academic year 2023-2024.

Signature of the Guide
Dr. Panimozhi N
Associate Professor, Dept. of CSE
BMSCE, Bangalore

Signature of the HOD
Dr. Jyothi S Nayak
Professor & Head,Dept. of CSE
BMSCE, Bangalore

DECLARATION

I, **Ananya Aithal (1BM21CS259)** students of 5^{th} Semester, B.E, Department of Computer Science and Engineering, B.M.S. College of Engineering, Bangalore, hereby declare that, this Laboratory Report has been carried out by me under the guidance of **Dr. Panimozhi N,** Assistant Professor, Department of CSE, B.M.S. College of Engineering, Bangalore during the academic semester Nov 2023 - Feb 2024.
We also declare that to the best of our knowledge and belief, the development reported here is not from part of any other report by any other students.

Signature

Ananya Aithal (1BM21CS259)

Table of Contents

Program 1: To implement a vacuum cleaner agent.

```
#Vacuum Cleaner Problem
#Clean is 0 and Dirty is 1

def vacuumcleaner(rooms,n):
        i = 0
        j = 0
        clean = 0
        while(clean < n):
            if(rooms[i][j] == 1):
                print("Cell",i,j,"is dirty")
                print("Performing suck...")
                rooms[i][j] = 0
                clean = clean + 1
                print("Cell",i,j,"is clean")

            elif(rooms[i][j] == 0):
                clean = clean + 1
                print("Cell",i,j,"is already clean")

            if(j==0):
                j+=1
                print("Moving Right....")
            elif(j==1 and i==0):
                i+=1
                j=0
                print("Moving Down......")


if __name__ == "__main__":
    n = 4
    clean = 0
    rows = 2
    cols = 2
    rooms = []
    print("Enter the room matrix with one entry in each line:")
    for i in range(rows):
        a =[]
        for j in range(cols):
```

```
            a.append(int(input()))
        rooms.append(a)

    vacuumcleaner(rooms,n)
```

Output:

```
C:\Users\Admin\Downloads\1BM21CS259\AI>python vacuum.py
Enter the room matrix with one entry in each line:
1
0
1
0
Cell 0 0 is dirty
Performing suck...
Cell 0 0 is clean
Moving Right....
Cell 0 1 is already clean
Moving Down......
Cell 1 0 is dirty
Performing suck...
Cell 1 0 is clean
Moving Right....
Cell 1 1 is already clean
```

Program 2: To solve 8 Puzzle Problem

```
# Python3 program to print the path from root
# node to destination node for N*N-1 puzzle
# algorithm using Branch and Bound
# The solution assumes that instance of
# puzzle is solvable

# Importing copy for deepcopy function
import copy

# Importing the heap functions from python
# library for Priority Queue
from heapq import heappush, heappop

# This variable can be changed to change
# the program from 8 puzzle(n=3) to 15
# puzzle(n=4) to 24 puzzle(n=5)...
n = 3

# bottom, left, top, right
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

# A class for Priority Queue
class priorityQueue:

	# Constructor to initialize a
	# Priority Queue
	def __init__(self):
		self.heap = []

	# Inserts a new key 'k'
	def push(self, k):
		heappush(self.heap, k)

	# Method to remove minimum element
	# from Priority Queue
	def pop(self):
		return heappop(self.heap)
```

```python
    # Method to know if the Queue is empty
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure
class node:

    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):

        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

        # Stores the matrix
        self.mat = mat

        # Stores the position at which the
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Stores the number of misplaced tiles
        self.cost = cost

        # Stores the number of moves so far
        self.level = level

    # This method is defined so that the
    # priority queue is formed based on
    # the cost variable of the objects
    def __lt__(self, nxt):
        return self.cost < nxt.cost

# Function to calculate the number of
# misplaced tiles ie. number of non-blank
```

```
# tiles not in their goal position
def calculateCost(mat, final) -> int:

	count = 0
	for i in range(n):
		for j in range(n):
			if ((mat[i][j]) and
				(mat[i][j] != final[i][j])):
				count += 1

	return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
			level, parent, final) -> node:

	# Copy data from parent matrix to current matrix
	new_mat = copy.deepcopy(mat)

	# Move tile by 1 position
	x1 = empty_tile_pos[0]
	y1 = empty_tile_pos[1]
	x2 = new_empty_tile_pos[0]
	y2 = new_empty_tile_pos[1]
	new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

	# Set number of misplaced tiles
	cost = calculateCost(new_mat, final)

	new_node = node(parent, new_mat, new_empty_tile_pos,
					cost, level)
	return new_node

# Function to print the N x N matrix
def printMatrix(mat):

	for i in range(n):
		for j in range(n):
			print("%d " % (mat[i][j]), end = " ")

		print()
```

```
# Function to check if (x, y) is a valid
# matrix coordinate
def isSafe(x, y):

	return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node
def printPath(root):

	if root == None:
		return

	printPath(root.parent)
	printMatrix(root.mat)
	print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state.
def solve(initial, empty_tile_pos, final):

	# Create a priority queue to store live
	# nodes of search tree
	pq = priorityQueue()

	# Create the root node
	cost = calculateCost(initial, final)
	root = node(None, initial,
				empty_tile_pos, cost, 0)

	# Add root to list of live nodes
	pq.push(root)

	# Finds a live node with least cost,
	# add its children to list of live
	# nodes and finally deletes it from
	# the list.
	while not pq.empty():
```

```
        # Find a live node with least estimated
        # cost and delete it from the list of
        # live nodes
        minimum = pq.pop()

        # If minimum is the answer node
        if minimum.cost == 0:

            # Print the path from root to
            # destination;
            printPath(minimum)
            return

        # Generate all possible children
        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):

                # Create a child node
                child = newNode(minimum.mat,
                minimum.empty_tile_pos,
                new_tile_pos,
                minimum.level + 1,
                minimum, final,)
                # Add child to list of live nodes
                pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Solvable Final configuration
# Value 0 is used for empty space
```

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

# Blank tile coordinates in
# initial configuration
empty_tile_pos = [ 1, 2 ]

# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)
```

Output:

```
C:\Users\Admin\Downloads\1BM21CS259\AI>eightpuzzle.py
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4

1   2   3
5   8   6
7   0   4

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

Program 3: Tic Tac Toe using Min-Max strategy

```
import random

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

def is_winner(board, player):
    # Check rows, columns, and diagonals for a win
    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)) or all(board[i][2 - i] == player for i in range(3)):
        return True

    return False

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

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

def minimax(board, depth, maximizing_player):
    if is_winner(board, "X"):
        return -1
    elif is_winner(board, "O"):
        return 1
    elif is_board_full(board):
        return 0

    if maximizing_player:
        max_eval = float('-inf')
```

```
        for i, j in get_empty_cells(board):
            board[i][j] = "O"
            eval = minimax(board, depth + 1, False)
            board[i][j] = " "
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for i, j in get_empty_cells(board):
            board[i][j] = "X"
            eval = minimax(board, depth + 1, True)
            board[i][j] = " "
            min_eval = min(min_eval, eval)
        return min_eval

def best_move(board):
    best_val = float('-inf')
    best_move = None
    for i, j in get_empty_cells(board):
        board[i][j] = "O"
        move_val = minimax(board, 0, False)
        board[i][j] = " "
        if move_val > best_val:
            best_move = (i, j)
            best_val = move_val
    return best_move

def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player_turn = True

    print_board(board)

    while True:
        if player_turn:
            row = int(input("Enter the row (0, 1, or 2): "))
            col = int(input("Enter the column (0, 1, or 2): "))
            if board[row][col] == " ":
                board[row][col] = "X"
                player_turn = False
```

```
            else:
                print("Cell already taken. Try again.")
                continue
        else:
            print("Computer's turn:")
            move = best_move(board)
            board[move[0]][move[1]] = "O"
            player_turn = True

        print_board(board)

        if is_winner(board, "X"):
            print("Congratulations! You win!")
            break
        elif is_winner(board, "O"):
            print("Computer wins! Better luck next time.")
            break
        elif is_board_full(board):
            print("It's a tie!")
            break

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

Output:

```
C:\Users\DL-Users\Downloads>python dola.py
  |   |
---------
  |   |
---------
  |   |
---------
Enter the row (0, 1, or 2): 1
Enter the column (0, 1, or 2): 0
  |   |
---------
X |   |
---------
  |   |
---------
Computer's turn:
O |   |
---------
X |   |
---------
  |   |
---------
```

```
Enter the row (0, 1, or 2): 1
Enter the column (0, 1, or 2): 1
O |   |
---------
X | X |
---------
  |   |
---------
Computer's turn:
O |   |
---------
X | X | O
---------
  |   |
---------
```

```
Enter the row (0, 1, or 2): 2
Enter the column (0, 1, or 2): 0
O |   |
---------
X | X | O
---------
X |   |
---------
Computer's turn:
O |   | O
---------
X | X | O
---------
X |   |
---------
```

```
Enter the row (0, 1, or 2): 0
Enter the column (0, 1, or 2): 1
O | X | O
---------
X | X | O
---------
X |   |
---------
Computer's turn:
O | X | O
---------
X | X | O
---------
X |   | O
---------
Computer wins! Better luck next time.
```

```
C:\Users\DL-Users\Downloads>python dola.py
  |   |
---------
  |   |
---------
  |   |
---------
Enter the row (0, 1, or 2): 0
Enter the column (0, 1, or 2): 0
X |   |
---------
  |   |
---------
  |   |
---------
Computer's turn:
X |   |
---------
  | O |
---------
  |   |
---------
```

```
---------
Enter the row (0, 1, or 2): 2
Enter the column (0, 1, or 2): 2
X |   |
---------
  | O |
---------
  |   | X
---------
Computer's turn:
X | O |
---------
  | O |
---------
  |   | X
---------
```

```
Enter the row (0, 1, or 2): 2
Enter the column (0, 1, or 2): 0
X | O |
---------
  | O |
---------
X |   | X
---------
Computer's turn:
X | O |
---------
  | O |
---------
X | O | X
---------
Computer wins! Better luck next time.
```

Program 4: Iterative Deepening Depth First Search

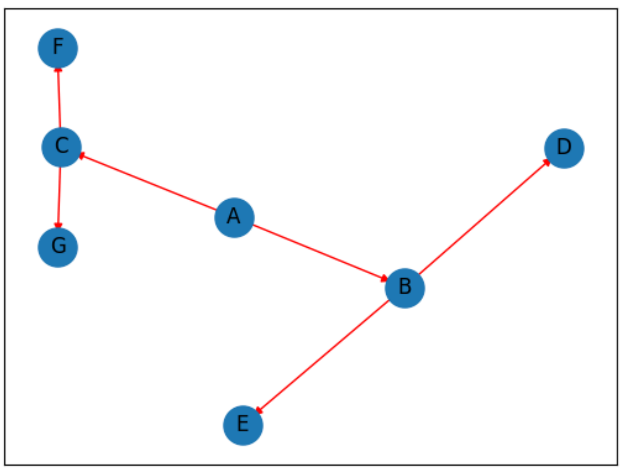
```
from collections import defaultdict
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
	def __init__(self,vertices):
		self.V = vertices
		self.graph = defaultdict(list)
	def addEdge(self,u,v):
		self.graph[u].append(v)
	def DLS(self,src,target,maxDepth):
		if src == target : return True
		if maxDepth <= 0 : return False
		for i in self.graph[src]:
				if(self.DLS(i,target,maxDepth-1)):
					return True
		return False
	def IDDFS(self,src, target, maxDepth):
		for i in range(maxDepth):
			if (self.DLS(src, target, i)):
				return True
		return False
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)
G = nx.DiGraph()
G.add_edges_from(
   [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G')])
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'), node_size = 500)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edges(G, pos, edge_color='r', arrows=True)
target = 6; maxDepth = 3; src = 0
target = int(input(("Enter the target:")))
```

```
maxDepth = int(input(("Enter the max depth:")))
src = int(input(("Enter the source:")))
if g.IDDFS(src, target, maxDepth) == True:
        print ("Target is reachable from source " + str(src) + " within max depth")
else :
        print ("Target is NOT reachable from source " + str(src) + " within max depth")
plt.show()
```

Output

```
C:\Users\Admin\Downloads\1BM21CS259\AI>d.py
C:\Users\Admin\AppData\Local\Programs\Python\Python312\Lib\site-packages\networkx\drawing\nx_pylab.py:437: UserWarning:
No data for colormapping provided via 'c'. Parameters 'cmap' will be ignored
  node_collection = ax.scatter(
Enter the target:4
Enter the max depth:3
Enter the source:0
Target is reachable from source 0 within max depth
```



Program 5: A* Search

```
import heapq

def astar(graph, heuristic, start, goal):
    came_from = {node: None for node in graph}

    open_set = [(0, start)]  # Priority queue with (f, node)
    closed_set = set()
    g_values = {node: float('inf') for node in graph}
    g_values[start] = 0
    f_values = {node: float('inf') for node in graph}
    f_values[start] = heuristic[start]

    while open_set:
        current_f, current_node = heapq.heappop(open_set)

        if current_node == goal:
            path = reconstruct_path(came_from, start, goal)
            return path

        closed_set.add(current_node)

        for neighbor, cost in graph[current_node]:
            if neighbor in closed_set:
                continue

            tentative_g = g_values[current_node] + cost

            if tentative_g < g_values[neighbor]:
                g_values[neighbor] = tentative_g
                f_values[neighbor] = tentative_g + heuristic[neighbor]
                heapq.heappush(open_set, (f_values[neighbor], neighbor))

    return None  # No path found

def reconstruct_path(came_from, start, goal):
    path = [goal]
    while goal != start:
        goal = came_from[goal]
```

```
        if goal is None:
            break  # To handle cases where there is no valid path
        path.append(goal)
    return path[::-1]


# Example usage:
n = int(input("Enter the number of nodes: "))
m = int(input("Enter the number of edges: "))

graph = {i: [] for i in range(n)}
heuristic = {}

for _ in range(m):
    src, dest, cost = map(int, input("Enter edge (source destination cost): ").split())
    graph[src].append((dest, cost))

for i in range(n):
    heuristic[i] = int(input(f"Enter heuristic value for node {i}: "))

s = int(input("Enter source node: "))
d = int(input("Enter destination node: "))

path = astar(graph, heuristic, s, d)

if path:
    print(f"Shortest path from {s} to {d}: {path}")
else:
    print(f"No path found from {s} to {d}.")
```

Output

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
Enter the number of nodes: 3
Enter the number of edges: 2
Enter edge (source destination cost): 0 1 2
Enter edge (source destination cost): 1 2 3
Enter heuristic value for node 0: 2
Enter heuristic value for node 1: 3
Enter heuristic value for node 2: 6
Enter source node: 0
Enter destination node: 2
Shortest path from 0 to 2: [2]
PS C:\Users\Admin\Desktop\1BM22CS096>
```

Program 6: Simulated Annealing Algorithm

```
import math
import random
def euclidean_distance(point1, point2):
    """Calculate Euclidean distance between two points."""
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
def total_distance(tour, distances):
    """Calculate the total distance of a tour."""
    total = 0
    for i in range(len(tour) - 1):
        total += distances[tour[i]][tour[i + 1]]
    total += distances[tour[-1]][tour[0]]  # Return to the starting point
    return total
def generate_neighbor(tour):
    """Generate a neighboring tour by swapping two random cities."""
    tour_copy = tour.copy()
    i, j = random.sample(range(len(tour)), 2)
    tour_copy[i], tour_copy[j] = tour_copy[j], tour_copy[i]
    return tour_copy
def simulated_annealing_tsp(distances, initial_tour, initial_temperature, cooling_rate,
max_iterations):
    current_tour = initial_tour
    best_tour = current_tour
    for iteration in range(max_iterations):
        temperature = initial_temperature * (cooling_rate**iteration)
        neighbor_tour = generate_neighbor(current_tour)
        current_distance = total_distance(current_tour, distances)
        neighbor_distance = total_distance(neighbor_tour, distances)
        if neighbor_distance < current_distance or random.uniform(0, 1) <
math.exp((current_distance - neighbor_distance) / temperature):
            current_tour = neighbor_tour
        if total_distance(neighbor_tour, distances) < total_distance(best_tour, distances):
            best_tour = neighbor_tour
    return best_tour
# Example Usage:
# Define cities and their coordinates
cities = {
    'A': (0, 0),
    'B': (1, 2),
```

```
    'C': (3, 1),
    'D': (5, 2),
    'E': (6, 0)
}
# Calculate distances between cities
distances = {city1: {city2: euclidean_distance(cities[city1], cities[city2]) for city2 in cities} for city1 in cities}
# Initial tour (can be random or a predefined order)
initial_tour = list(cities.keys())
# Input parameters from the user
initial_temperature = float(input("Enter the initial temperature: "))
cooling_rate = float(input("Enter the cooling rate: "))
max_iterations = int(input("Enter the maximum number of iterations: "))
# Run simulated annealing for TSP
best_tour = simulated_annealing_tsp(distances, initial_tour, initial_temperature, cooling_rate, max_iterations)
# Print the best tour and its total distance
print("Best Tour:", best_tour)
print("Total Distance:", total_distance(best_tour, distances))
```

Output

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
Enter the initial temperature: 1000
Enter the cooling rate: 0.95
Enter the maximum number of iterations: 1000
Best Tour: ['E', 'D', 'B', 'A', 'C']
Total Distance: 14.79669127533634
PS C:\Users\Admin\Desktop\1BM22CS096>
```

Program 6: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

```
//Truth Table Approach

from itertools import product

def is_entailed(knowledge_base, query):
    symbols = set()
    symbols.update(*(clause.free_symbols for clause in knowledge_base))
    symbols.update(query.free_symbols)

    truth_table = list(product([False, True], repeat=len(symbols)))

    for assignment in truth_table:
        assignment_dict = dict(zip(symbols, assignment))

        kb_eval = all(clause.subs(assignment_dict) for clause in knowledge_base)
        query_eval = query.subs(assignment_dict)

        if kb_eval and not query_eval:
            return False

    return True

if __name__ == "__main__":
    from sympy import symbols, Implies, Not

    # Define symbols
    R, W, G = symbols('R W G')

    # Knowledge base
    knowledge_base = [
        Implies(R, W),
        Implies(W, G)
    ]

    # Query
    query = Implies(R, G)
```

```
    # Check if the query is entailed by the knowledge base using truth table
    entailed = is_entailed(knowledge_base, query)

    # Output result
   print("knowledge_base:")
    print(knowledge_base)
    print("query:")
    print(query)

if entailed:
        print("The query is entailed by the knowledge base.")
    else:
        print("The query is not entailed by the knowledge base.")
```

Output

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
knowledge_base:
[Implies(R, W), Implies(W, G)]
query:
Implies(W, R)
The query is not entailed by the knowledge base.
PS C:\Users\Admin\Desktop\1BM22CS096>
```

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
knowledge_base:
[Implies(R, W), Implies(W, G)]
query:
Implies(R, G)
The query is entailed by the knowledge base.
PS C:\Users\Admin\Desktop\1BM22CS096>
```

Program 7: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

```
from sympy import symbols, Or, Not, Implies, satisfiable

def resolution(kb, query):
    symbols = set()
    symbols.update(*(clause.free_symbols for clause in kb))
    symbols.update(query.free_symbols)

    # Convert to CNF
    cnf_kb = And(*kb)
    cnf_query = And(Not(query))

    # Negate the query and add it to the knowledge base
    extended_kb = And(cnf_kb, cnf_query)

    # Check for satisfiability (contradiction)
    return not satisfiable(extended_kb)

if __name__ == "__main__":
    # Define symbols
    P, Q, R, S = symbols('P Q R S')

    # Knowledge base
    kb = [
        Or(P, Q),
        Or(Not(P), R),
        Or(Not(Q), S),
        Or(Not(R), Not(S))
    ]

    # Query
    query = Not(P)

    # Check if the query is proven by the knowledge base using resolution
    proven = resolution(kb, query)
    print("knowledge_base:")
    print(kb)
    print("query:")
```

```
print(query)

# Output result
if proven:
    print("The query is proven by the knowledge base using resolution.")
else:
    print("The query is not proven by the knowledge base using resolution.")
```

Output

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
knowledge_base:
[P | Q, R | ~P, S | ~Q, ~R | ~S]
query:
~P
The query is not proven by the knowledge base using resolution.
```

```
PS C:\Users\Admin\Desktop\1BM22CS096> python -u "c:\Users\Admin\Desktop\1BM22CS096\m.py"
knowledge_base:
[P | Q, R | ~P, S | ~Q, ~R | ~S]
query:
P | S
The query is proven by the knowledge base using resolution.
PS C:\Users\Admin\Desktop\1BM22CS096>
```

Program 9: Implement unification in first order logic.

```
def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def unify(x, y, theta):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x.isalpha():
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.isalpha():
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return None
        else:
            for xi, yi in zip(x, y):
                theta = unify(xi, yi, theta)
            return theta
    else:
        return None

if __name__ == "__main__":
    # Example usage
    expr1 = ["P", "x", "y"]
    expr2 = ["P", "a", "b"]
    substitution = unify(expr1, expr2, {})

    if substitution is not None:
        print("Unification successful. Substitution:")
        for key, value in substitution.items():
            print(f"{key} = {value}")
```

```
    else:
        print("Unification failed.")
```

Output

```
if __name__ == "__main__":
    expr1 = ["P", "x", "y"]
    expr2 = ["P", "a", "b"]
    substitution = unify(expr1, expr2, {})
```

```
Unification successful. Substitution:
x = a
y = b
```

Program 10: Convert a given first order logic statement into Conjunctive Normal Form (CNF).

```
from sympy import symbols, Implies, Not, Or, And, to_cnf, simplify_logic

def convert_to_cnf(statement):
    # Define symbols for variables
    symbols_list = list(set(symbols(statement)))
    vars = symbols(symbols_list)

    # Parse the given statement
    parsed_statement = eval(statement)

    # Convert to CNF
    cnf = to_cnf(parsed_statement)

    # Simplify the CNF
    simplified_cnf = simplify_logic(cnf)

    return simplified_cnf

if __name__ == "__main__":
    # Example usage
    input_statement = "P(x) => Q(x) | R(x)"
    cnf_result = convert_to_cnf(input_statement)

    print("Original Statement:", input_statement)
    print("CNF Form:", cnf_result)
```

Output

```
Original Statement: P(x) => Q(x) | R(x)
CNF Form: (~P(x) | Q(x) | R(x))
```

Program 11: Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

```
from prover9 import Prover9

def forward_reasoning(knowledge_base, query):
    prover = Prover9()

    # Add statements to the knowledge base
    for statement in knowledge_base:
        prover.add_statement(statement)

    # Add the negation of the query to the knowledge base
    prover.add_statement(f"~({query})")

    # Attempt to prove the contradiction
    result = prover.prove()

    return not result

if __name__ == "__main__":
    # Example knowledge base
    knowledge_base = ["P(x) => Q(x)", "Q(a)"]

    # Example query
    query = "P(a)"

    # Perform forward reasoning
    result = forward_reasoning(knowledge_base, query)

    if result:
        print(f"The query '{query}' can be inferred from the knowledge base.")
    else:
        print(f"The query '{query}' cannot be inferred from the knowledge base.")
```

Output