**DEERWALK INSTITUTE OF TECHNOLOGY**

**Tribhuvan University Faculties of Computer Science**



**Bachelors of Science in**

**Computer Science and Information Technology (BSc. CSIT)**

**Course: Artificial Intelligence Class of 2026**

**Semester - 4**

|  |  |
| --- | --- |
| Submitted by: | Submitted to: |
| Sneha Shrestha | Surya Bahadur Basnet |
| Roll no: 1237 |  |

**Lab Report on: Artificial Intelligence**

# Table of Contents

|  |  |  |
| --- | --- | --- |
| S.no | Title | Signature |
| 1. | Program to implement simple intelligent agent (Vacuum Cleaner Agent) |  |
| 2. | Program to implement Depth First Search |  |
| 3. | Program to implement Breadth First Search |  |
| 4. | Program to implement Best First Search |  |
| 5. | Introduction to Prolog |  |
| 6. | Program to implement Naïve Baye’s Theorem |  |
| 7. | Program to implement Perceptron Learning Algorithm |  |
| 8. | Implement Back Propagation Algorithm in XOR Gate |  |
| 9. | Implement a Family Tree in Prolog |  |

**Lab 1**

Program to implement simple intelligent agent (Vacuum Cleaner Agent)

# Theory

In [artificial intelligence,](https://en.wikipedia.org/wiki/Artificial_intelligence) an intelligent agent (IA) is an agent acting in an [intelligent](https://en.wikipedia.org/wiki/Intelligent) manner; It [perceives](https://en.wikipedia.org/wiki/Machine_perception) its environment, takes actions [autonomously](https://en.wikipedia.org/wiki/Autonomous) in order to achieve goals, and may improve its performance with [learning](https://en.wikipedia.org/wiki/Machine_learning) or acquiring [knowledge.](https://en.wikipedia.org/wiki/Knowledge_representation)

Let us assume that a Vacuum Cleaner is an intelligence agent. Let the environment be just two rooms. The vacuum robot may be in either of the room. Let there be dirt in either one of the rooms, both of the rooms or none. The goal of our experiment is to ensure that both rooms are clean.

**Percepts**: Location and Content  Room 1/Room 2, Dirty

**Actions**: Left, Right, Clean, No-Operation **Goal**: To clean both rooms **Environment**: Rooms 1 and 2

**Pseudo-code**:

Function Reflex-Vacuum Agent (location, status) return is status = Dirty ten return Suck

else if location = A then return Right else if location = B then return Left

# Code:

#include <stdio.h>

typedef enum

{

DIRTY = 1,

CLEAN = 0

} Status; typedef enum

{

LEFT, RIGHT,

CLEAN\_ACTION, NO\_OP

} Action;

Action reflexVacuumAgent(char location, Status status)

{

if (status == DIRTY)

{

return CLEAN\_ACTION;

}

else if (location == 'A')

{

return RIGHT;

}

else if (location == 'B')

{

return LEFT;

}

else

{

return NO\_OP;

}

}

int main()

{

char location; Status status; Action action;

// Get initial location and status from the environment printf("Enter the current location (A or B): ");

scanf(" %c", &location);

printf("Enter the status of the current room (0 for clean, 1 for dirty): ");

scanf("%d", (int \*)&status);

// Calculate the next action based on the current location and status action = reflexVacuumAgent(location, status);

// Perform the action and print the result switch (action)

{

case LEFT:

printf("Action: Move Left\n"); break;

case RIGHT:

printf("Action: Move Right\n"); break;

case CLEAN\_ACTION:

printf("Action: Clean\n"); break;

case NO\_OP:

printf("Action: No-Operation\n"); break;

default:

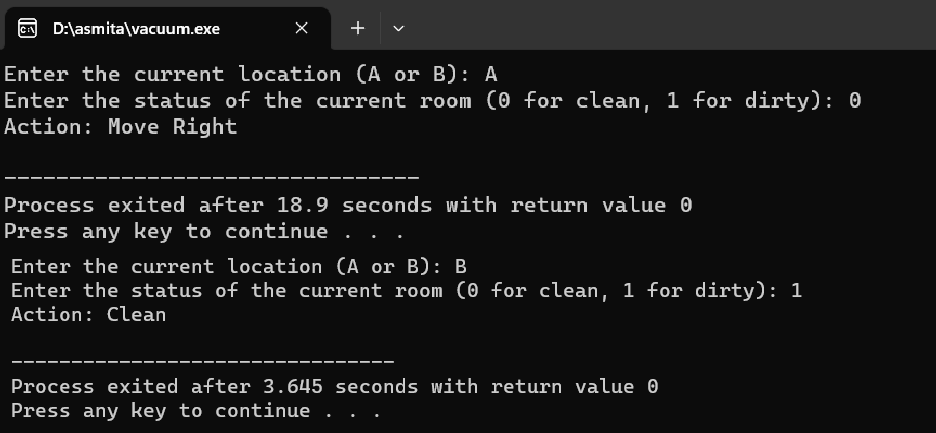
printf("Invalid action\n"); break;

}

return 0;

}

# Output:

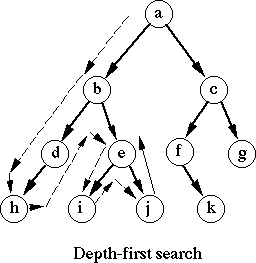


**Lab 2**

Program to implement Depth First Search

# Theory

Depth First Search is a searching algorithm in which used in graph traversal. In this method, each data in the node is searched from the top to bottom from left to right. The following diagram illustrates Depth First Search. Here **N** and **J** are the results which we must acquire.



# Code:

#include <stdio.h> #include <stdbool.h> #define MAX\_NODES 100 bool visited[MAX\_NODES];

int adjacencyMatrix[MAX\_NODES][MAX\_NODES]; int numNodes;

void initializeGraph()

{

int i, j;

for (i = 0; i < MAX\_NODES; i++)

{

visited[i] = false;

for (j = 0; j < MAX\_NODES; j++)

{

adjacencyMatrix[i][j] = 0;

}

}

}

void addEdge(int src, int dest)

{

adjacencyMatrix[src][dest] = 1;

adjacencyMatrix[dest][src] = 1;

}

void depthFirstSearch(int startNode)

{

printf("Visited node: %d\n", startNode); visited[startNode] = true;

int i;

for (i = 0; i < numNodes; i++)

{

if (adjacencyMatrix[startNode][i] && !visited[i])

{

depthFirstSearch(i);

}

}

}

int main()

{

int numEdges, i; int src, dest;

printf("Enter the number of nodes: "); scanf("%d", &numNodes); printf("Enter the number of edges: "); scanf("%d", &numEdges); initializeGraph();

printf("Enter the edges (source destination):\n"); for (i = 0; i < numEdges; i++)

{

scanf("%d %d", &src, &dest); addEdge(src, dest);

}

int startNode;

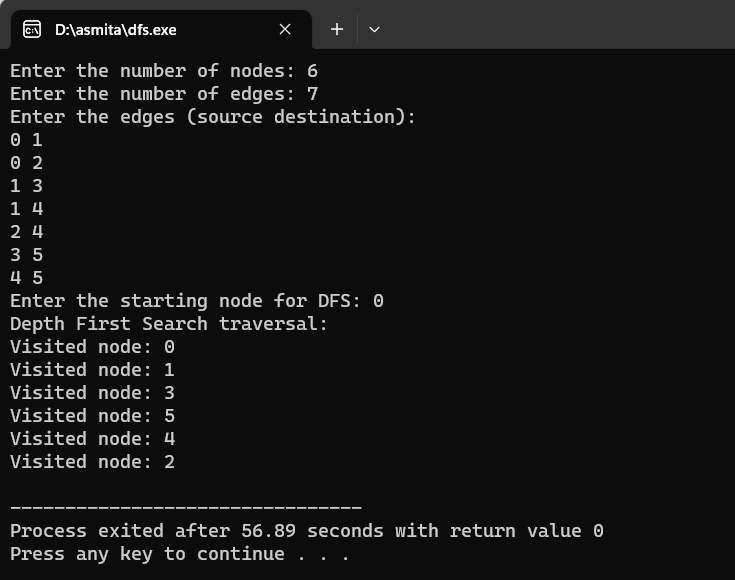
printf("Enter the starting node for DFS: "); scanf("%d", &startNode);

printf("Depth First Search traversal:\n"); depthFirstSearch(startNode);

return 0;

}

# Output:

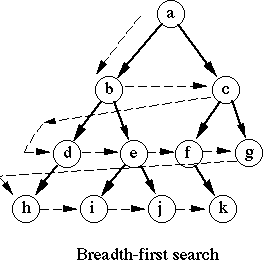


**Lab 3**

Program to implement Breadth First Search

# Theory

Breadth First Search is a graph traversal searching algorithm. In this method, each node in the tree is searched from left to right from starting from the starting node to the nodes in the consecutive levels. The following diagram depicts Breadth First Search.



# Code

#include <stdio.h> #include <stdbool.h> #define MAX\_NODES 100

bool visited[MAX\_NODES];

int adjacencyMatrix[MAX\_NODES][MAX\_NODES]; int numNodes;

void initializeGraph()

{

int i, j;

for (i = 0; i < MAX\_NODES; i++)

{

visited[i] = false;

for (j = 0; j < MAX\_NODES; j++)

{

adjacencyMatrix[i][j] = 0;

}

}

}

void addEdge(int src, int dest)

{

adjacencyMatrix[src][dest] = 1;

adjacencyMatrix[dest][src] = 1;

}

void breadthFirstSearch(int startNode)

{

int queue[MAX\_NODES]; int front = 0, rear = 0;

int current, i;

printf("Visited node: %d\n", startNode); visited[startNode] = true; queue[rear++] = startNode;

while (front < rear)

{

current = queue[front++];

for (i = 0; i < numNodes; i++)

{

if (adjacencyMatrix[current][i] && !visited[i])

{

printf("Visited node: %d\n", i); visited[i] = true; queue[rear++] = i;

}

}

}

}

int main()

{

int numEdges, i; int src, dest;

printf("Enter the number of nodes: "); scanf("%d", &numNodes); printf("Enter the number of edges: "); scanf("%d", &numEdges); initializeGraph();

printf("Enter the edges (source destination):\n"); for (i = 0; i < numEdges; i++)

{

scanf("%d %d", &src, &dest); addEdge(src, dest);

}

int startNode;

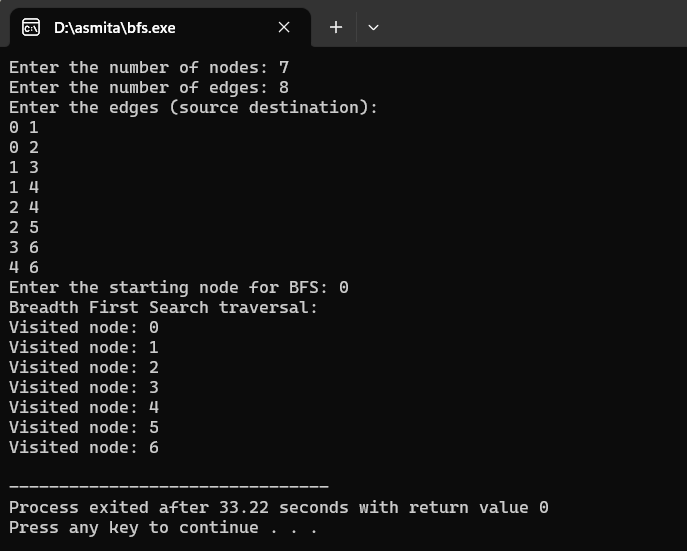
printf("Enter the starting node for BFS: "); scanf("%d", &startNode);

printf("Breadth First Search traversal:\n"); breadthFirstSearch(startNode);

return 0;

}

**Output**



# Lab 4

Program to implement Best First Search

# Theory

The Best-First Search (BFS) algorithm is a search algorithm that explores a graph or a tree based on an evaluation function. It evaluates each node based on a heuristic value and selects the most promising node to expand next. Best First Search falls under the category of Heuristic Search or Informed Search.

# Code

#include <bits/stdc++.h> using namespace std; typedef pair<int, int> pi; vector<vector<pi>> graph;

void addedge(int x, int y, int cost)

{

graph[x].push\_back(make\_pair(cost, y)); graph[y].push\_back(make\_pair(cost, x));

}

void best\_first\_search(int actual\_Src, int target, int n)

{

vector<bool> visited(n, false);

// MIN HEAP priority queue

priority\_queue<pi, vector<pi>, greater<pi>> pq;

// sorting in pq gets done by first value of pair pq.push(make\_pair(0, actual\_Src));

int s = actual\_Src; visited[s] = true; while (!pq.empty())

{

int x = pq.top().second;

// Displaying the path having lowest cost cout << x << " ";

pq.pop();

if (x == target) break;

for (int i = 0; i < graph[x].size(); i++)

{

if (!visited[graph[x][i].second])

{

visited[graph[x][i].second] = true; pq.push(make\_pair(graph[x][i].first, graph[x][i].second));

}

}

}

}

// Driver code to test above methods int main()

{

// No. of Nodes int v = 14; graph.resize(v);

// The nodes shown in above example(by alphabets) are

// implemented using integers addedge(x,y,cost); addedge(0, 1, 3);

addedge(0, 2, 6);

addedge(0, 3, 5);

addedge(1, 4, 9);

addedge(1, 5, 8);

addedge(2, 6, 12);

addedge(2, 7, 14);

addedge(3, 8, 7);

addedge(8, 9, 5);

addedge(8, 10, 6);

addedge(9, 11, 1);

addedge(9, 12, 10);

addedge(9, 13, 2);

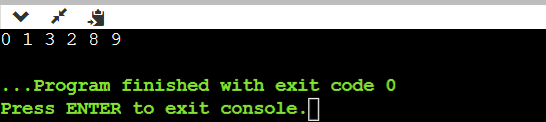
int source = 0; int target = 9;

best\_first\_search(source, target, v);

return 0;

}

# Output



**Lab 5**

Introduction to Prolog

# Theory

Prolog is a programming language for non-numeric computation. It is well suited for programs involving objects and relationship between objects. The user develops the problem using facts and rules and define a goal. Prolog then attempts to prove this goal by using formal reasoning of the given data.

Prolog Programming is based on a few basic concepts like

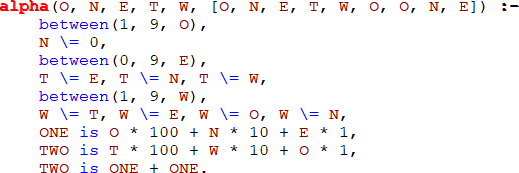
**Predicate:** They are statements that defines relationship between two or more objects

**Facts:** They are statements that are true about a particular object

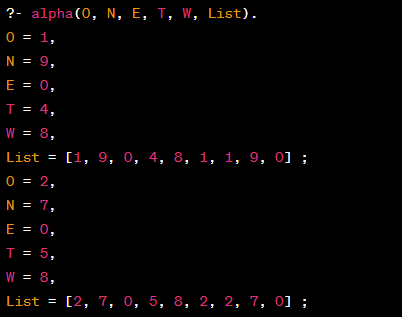
**Rules:** They are statements that defines relationship between two or more predicates

**Query:** They are questions that is to be passed to the prolog system

**Code**



**Output**



**Lab 6**

Program to implement Naïve Baye’s Theorem

# Theory

Naive Bayes is a classification algorithm based on Bayes' theorem. Bayes' theorem is a fundamental concept in probability theory that calculates the conditional probability of an event based on prior knowledge or evidence. The theorem is named after the Reverend Thomas Bayes, who introduced it in the 18th century.

Bayes' theorem can be stated as follows:

P(A | B) = (P(B | A) \* P(A)) / P(B)

Where:

* P(A | B) is the probability of event A occurring given that event B has occurred.
* P(B | A) is the probability of event B occurring given that event A has occurred.
* P(A) is the prior probability of event A.
* P(B) is the prior probability of event B.

# Code

#include <stdio.h>

double calculateBayes(double probA, double probBGivenA, double probB)

{

// Calculate P(A|B) using Bayes' theorem

double probAGivenB = (probBGivenA \* probA) / probB; return probAGivenB;

}

int main()

{

double probA = 0.3; // Probability of event A

double probBGivenA = 0.6; // Probability of event B given A double probB = 0.4; // Probability of event B

double result = calculateBayes(probA, probBGivenA, probB);

printf("Probability of event A is: %lf\n", probA);

printf("Probability of event B given event A: %lf\n", probBGivenA); printf("Probability of event B: %lf\n", probB);

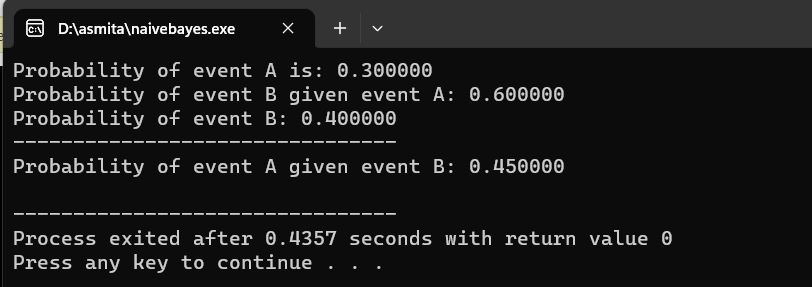
printf(" \n");

printf("Probability of event A given event B: %lf\n", result);

return 0;

}

# Output



**Lab 7**

Program to implement Perceptron Learning Algorithm

# Theory

The Perceptron Learning Algorithm (PLA) is a simple supervised learning algorithm used for binary classification. The Perceptron is a single-layer neural network that takes a set of input features and produces a binary output. It is based on the concept of a biological neuron, where inputs are weighted, summed, and passed through an activation function to produce an output.

# Code

#include <stdio.h>

#define NUM\_SAMPLES 4

#define NUM\_FEATURES 3

#define LEARNING\_RATE 0.1

#define MAX\_ITERATIONS 10

int trainSamples[NUM\_SAMPLES][NUM\_FEATURES] = {{1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1,

1}};

int targetOutputs[NUM\_SAMPLES] = {-1, -1, -1, 1}; int weights[NUM\_FEATURES] = {0};

int predict(int \*sample)

{

int sum = 0;

for (int i = 0; i < NUM\_FEATURES; i++)

{

sum += sample[i] \* weights[i];

}

return (sum >= 0) ? 1 : -1;

}

void updateWeights(int \*sample, int targetOutput, int predictedOutput)

{

for (int i = 0; i < NUM\_FEATURES; i++)

{

weights[i] += LEARNING\_RATE \* (targetOutput - predictedOutput) \* sample[i];

}

}

void train()

{

int iteration = 0, errors = 1;

while (errors != 0 && iteration < MAX\_ITERATIONS)

{

errors = 0;

for (int i = 0; i < NUM\_SAMPLES; i++)

{

int predictedOutput = predict(trainSamples[i]); if (predictedOutput != targetOutputs[i])

{

updateWeights(trainSamples[i], targetOutputs[i], predictedOutput); errors++;

}

}

iteration++;

}

}

int main()

{

train();

printf("Learned Weights: ");

for (int i = 0; i < NUM\_FEATURES; i++)

{

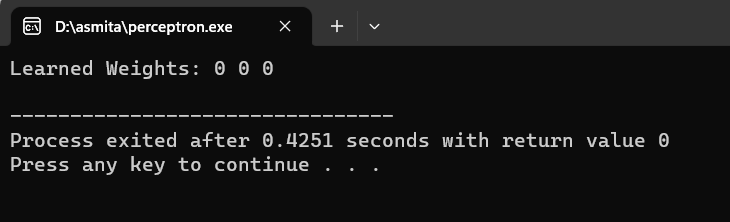
printf("%d ", weights[i]);

}

printf("\n"); return 0;

}

**Output**



# Lab 8

Implement Back Propagation Algorithm in XOR Gate

# Theory

The backpropagation algorithm is a common technique used to train artificial neural networks. It involves propagating the errors backward through the network, adjusting the weights and biases to minimize the error between the predicted output and the desired output.

# Code

#include <stdio.h> #include <stdlib.h> #include <math.h> #define INPUT\_SIZE 2

#define HIDDEN\_SIZE 2

#define OUTPUT\_SIZE 1

#define LEARNING\_RATE 0.1

#define MAX\_ITERATIONS 10000

double sigmoid(double x)

{

return 1 / (1 + exp(-x));

}

double derivativeSigmoid(double x)

{

return x \* (1 - x);

}

void train(double inputs[][INPUT\_SIZE], double labels[], double hiddenWeights[][HIDDEN\_SIZE], double outputWeights[],

double hiddenBiases[], double outputBias[])

{

double hiddenLayer[HIDDEN\_SIZE]; double outputLayer[OUTPUT\_SIZE]; double hiddenLayerDelta[HIDDEN\_SIZE]; double outputLayerDelta[OUTPUT\_SIZE];

for (int iteration = 0; iteration < MAX\_ITERATIONS; iteration++)

{

for (int sample = 0; sample < 4; sample++)

{

// Forward propagation

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

double sum = 0;

for (int j = 0; j < INPUT\_SIZE; j++)

{

sum += inputs[sample][j] \* hiddenWeights[j][i];

}

hiddenLayer[i] = sigmoid(sum + hiddenBiases[i]);

}

double sum = 0;

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

sum += hiddenLayer[i] \* outputWeights[i];

}

outputLayer[0] = sigmoid(sum + outputBias[0]);

// Backpropagation

outputLayerDelta[0] = (labels[sample] - outputLayer[0]) \* derivativeSigmoid(outputLayer[0]); for (int i = 0; i < HIDDEN\_SIZE; i++)

{

hiddenLayerDelta[i] = outputLayerDelta[0] \* outputWeights[i] \* derivativeSigmoid(hiddenLayer[i]);

}

// Update weights and biases

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

for (int j = 0; j < INPUT\_SIZE; j++)

{

hiddenWeights[j][i] += LEARNING\_RATE \* inputs[sample][j] \* hiddenLayerDelta[i];

}

}

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

outputWeights[i] += LEARNING\_RATE \* hiddenLayer[i] \* outputLayerDelta[0];

}

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

hiddenBiases[i] += LEARNING\_RATE \* hiddenLayerDelta[i];

}

outputBias[0] += LEARNING\_RATE \* outputLayerDelta[0];

}

}

}

void test(double inputs[][INPUT\_SIZE], double hiddenWeights[][HIDDEN\_SIZE], double outputWeights[],

double hiddenBiases[], double outputBias[])

{

printf("Testing XOR gate:\n"); for (int i = 0; i < 4; i++)

{

double hiddenLayer[HIDDEN\_SIZE]; double outputLayer[OUTPUT\_SIZE]; for (int j = 0; j < HIDDEN\_SIZE; j++)

{

double sum = 0;

for (int k = 0; k < INPUT\_SIZE; k++)

{

sum += inputs[i][k] \* hiddenWeights[k][j];

}

hiddenLayer[j] = sigmoid(sum + hiddenBiases[j]);

}

double sum = 0;

for (int j = 0; j < HIDDEN\_SIZE; j++)

{

sum += hiddenLayer[j] \* outputWeights[j];

}

outputLayer[0] = sigmoid(sum + outputBias[0]);

printf("Input: %d %d, Output: %.2lf\n", (int)inputs[i][0], (int)inputs[i][1], outputLayer[0]);

}

}

int main()

{

double inputs[4][INPUT\_SIZE] = {

{0, 0},

{0, 1},

{1, 0},

{1, 1}};

double labels[4] = {0, 1, 1, 0};

double hiddenWeights[INPUT\_SIZE][HIDDEN\_SIZE]; double outputWeights[HIDDEN\_SIZE];

double hiddenBiases[HIDDEN\_SIZE]; double outputBias[OUTPUT\_SIZE];

// Initialize weights and biases with random values between -1 and 1 for (int i = 0; i < INPUT\_SIZE; i++)

{

for (int j = 0; j < HIDDEN\_SIZE; j++)

{

hiddenWeights[i][j] = (double)rand() / RAND\_MAX \* 2 - 1;

}

}

for (int i = 0; i < HIDDEN\_SIZE; i++)

{

outputWeights[i] = (double)rand() / RAND\_MAX \* 2 - 1; hiddenBiases[i] = (double)rand() / RAND\_MAX \* 2 - 1;

}

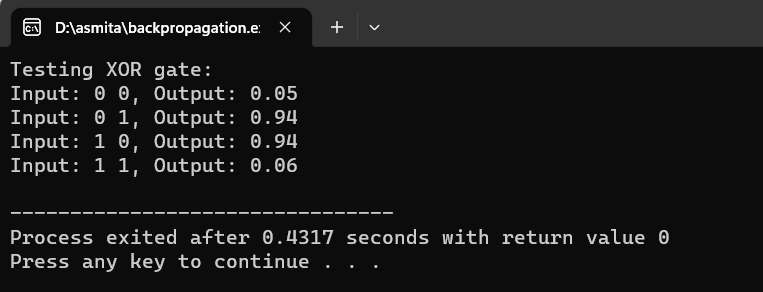
outputBias[0] = (double)rand() / RAND\_MAX \* 2 - 1;

train(inputs, labels, hiddenWeights, outputWeights, hiddenBiases, outputBias); test(inputs, hiddenWeights, outputWeights, hiddenBiases, outputBias);

return 0;

}

# Output



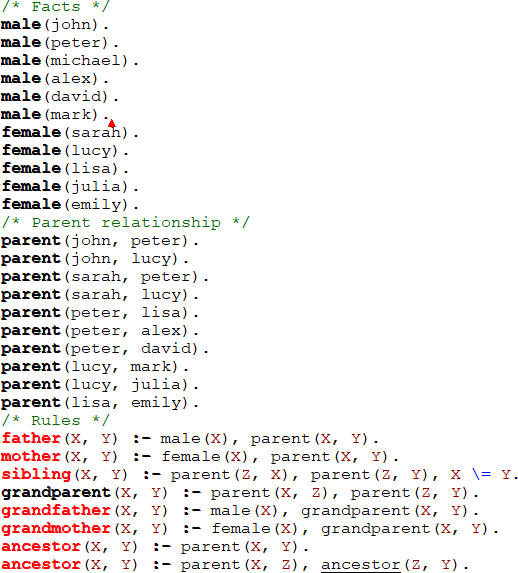
**Lab 9**

Implement a Family Tree in Prolog

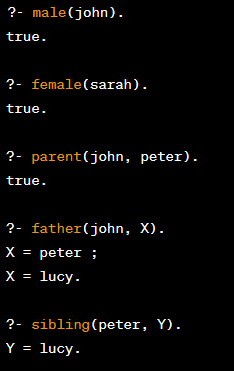
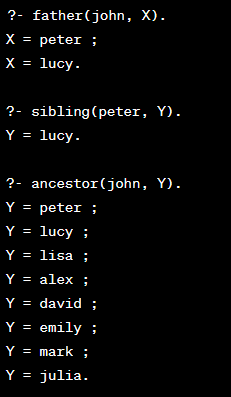
# Theory

Prolog is a programming language for non-numeric computation. It is well suited for programs involving objects and relationship between objects. The user develops the problem using facts and rules and define a goal. Prolog then attempts to prove this goal by using formal reasoning of the given data.

**Code**



# Output



**Conclusion:**

At the end of this lab, we successfully implemented different AI concepts and learning algorithms using the C programming language and Prolog. We conducted extensive testing by providing various inputs and verifying the corresponding outputs to ensure their functionality.