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 2024/Semester: IV

Lab Report on:

AI Practical Lab

Submitted by:
Name: Kshitiz Shah
Roll: 1013

Submitted to:
Akash Deo

LAB 1

Programme to implement simple intelligent agent (Vacuum Cleaner Agent)

Theory

Intelligent agents are capable of perceiving their surroundings and making decisions based on that information in order to achieve their objectives. Suppose we consider a Vacuum Cleaner as an example of an intelligent agent. We can imagine a scenario where there are two rooms in the environment, and the vacuum robot can be located in either room.

Percepts: Location and Content → Room A/Room B, Dirty

Actions: Left, Right, Clean, No-Operation

Goal: To clean both rooms

Environment → Rooms A and B

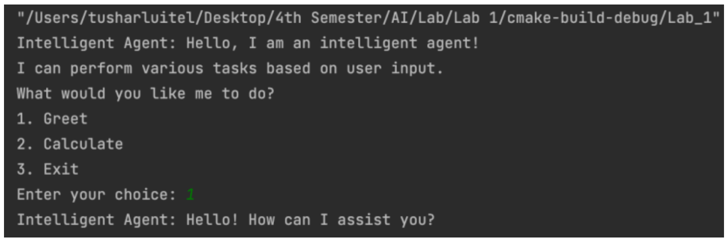
The pseudo-code for the agent is as follows:

```
Function Reflex-Vacuum Agent (location, status) return
isStatus = Dirty ten return Suck
else if location = A then return Right
else if location = B then return Left
```

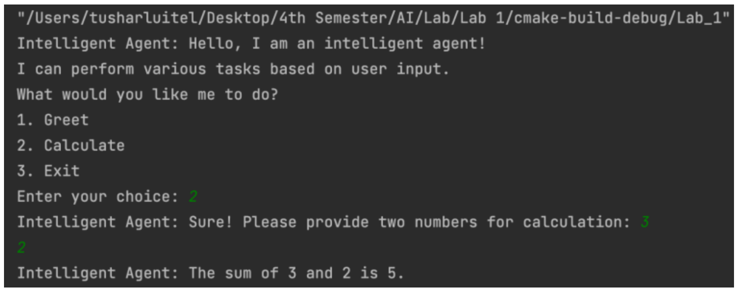
Code

```
#include <stdio.h>
void intelligentAgent() {
    printf("Intelligent Agent: Hello, I am an intelligent agent!\n");
    printf("I can perform various tasks based on user input.\n");
    printf("What would you like me to do?\n");
    printf("1. Greet\n");
    printf("2. Calculate\n");
    printf("3. Exit\n");

    int choice;
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Intelligent Agent: Hello! How can I assist you?\n");
            break;
        case 2:
            printf("Intelligent Agent: Sure! Please provide two numbers for calculation: ");
            int num1, num2;
            scanf("%d %d", &num1, &num2);
            int sum = num1 + num2;
            printf("Intelligent Agent: The sum of %d and %d is %d.\n", num1, num2, sum);
            break;
        case 3:
            printf("Intelligent Agent: Goodbye! Have a great day!\n");
            break;
        default:
            printf("Intelligent Agent: I'm sorry, I didn't understand your choice.\n");
            break;
    }
}

int main() {
    intelligentAgent();

    return 0;
}
```

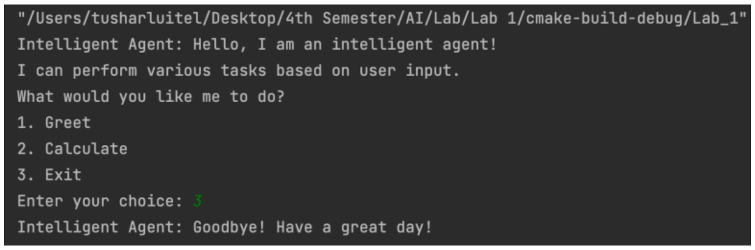
Output



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 1/cmake-build-debug/Lab_1"
Intelligent Agent: Hello, I am an intelligent agent!
I can perform various tasks based on user input.
What would you like me to do?
1. Greet
2. Calculate
3. Exit
Enter your choice: 1
Intelligent Agent: Hello! How can I assist you?
```



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 1/cmake-build-debug/Lab_1"
Intelligent Agent: Hello, I am an intelligent agent!
I can perform various tasks based on user input.
What would you like me to do?
1. Greet
2. Calculate
3. Exit
Enter your choice: 2
Intelligent Agent: Sure! Please provide two numbers for calculation: 3
2
Intelligent Agent: The sum of 3 and 2 is 5.
```



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 1/cmake-build-debug/Lab_1"
Intelligent Agent: Hello, I am an intelligent agent!
I can perform various tasks based on user input.
What would you like me to do?
1. Greet
2. Calculate
3. Exit
Enter your choice: 3
Intelligent Agent: Goodbye! Have a great day!
```

Conclusion

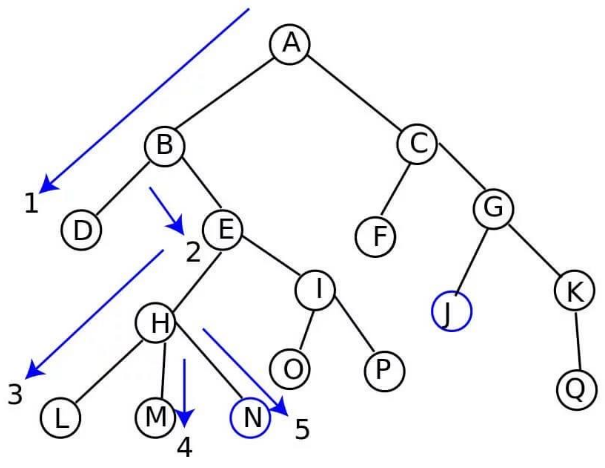
Thus, we learned how to implement Intelligent agents in the C programming Language

Lab 2

Programme 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.

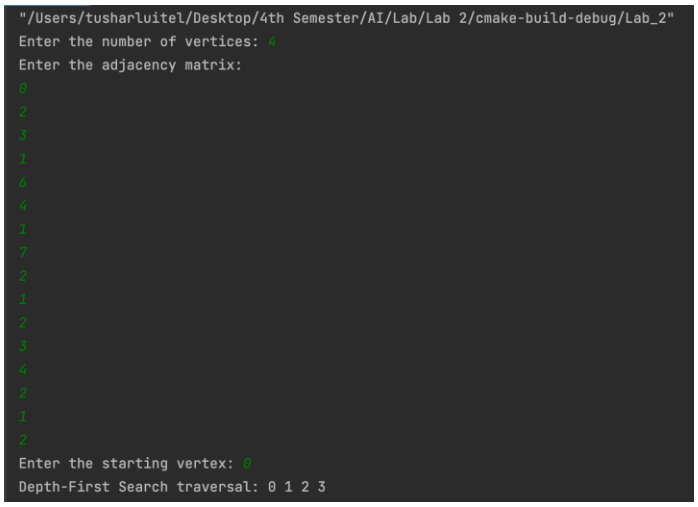


Courtesy of totheinnovation.com

Code

```
#include <stdio.h>

#define MAX_VERTICES 100

int visited[MAX_VERTICES];
void dfs(int graph[MAX_VERTICES][MAX_VERTICES], int v, int numVertices) {
    visited[v] = 1;
    printf("%d ", v);
    for (int i = 0; i < numVertices; i++) {
        if (graph[v][i] && !visited[i]) {
            dfs(graph, i, numVertices);
        }
    }
}

int main() {
    int numVertices, startVertex;
    int graph[MAX_VERTICES][MAX_VERTICES];

    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);
    for (int i = 0; i < numVertices; i++) {
        visited[i] = 0;
    }
    printf("Depth-First Search traversal: ");
    dfs(graph, startVertex, numVertices);

    return 0;
}
```

Output



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 2/cmake-build-debug/Lab_2"
Enter the number of vertices: 4
Enter the adjacency matrix:
0
2
3
1
6
4
1
7
2
1
2
3
4
2
1
2
Enter the starting vertex: 0
Depth-First Search traversal: 0 1 2 3
```

Conclusion:

Thus, we understood how to implement Depth First Search graph traversal algorithm in the C programming language

Lab 3

Programme to implement Breadth First Search

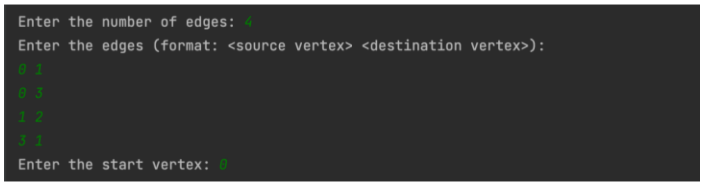
Theory

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.

Code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

// Create a new node
Node* createNode(int v) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Breadth-First Search function
void bfs(int startVertex, int visited[], Node* adjacencyList[]) {
    int queue[MAX_VERTICES];
    int front = 0, rear = -1;

    visited[startVertex] = 1;
    queue[++rear] = startVertex;

    while (front <= rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);


        Node* temp = adjacencyList[currentVertex];
        while (temp != NULL) {
            int connectedVertex = temp->vertex;
            if (!visited[connectedVertex]) {
                visited[connectedVertex] = 1;
                queue[++rear] = connectedVertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    int numVertices, numEdges;
    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    Node* adjacencyList[MAX_VERTICES];
    for (int i = 0; i < numVertices; i++) {
        adjacencyList[i] = NULL;
    }

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

    printf("Enter the edges (format: <source vertex> <destination vertex>):\n");
    for (int i = 0; i < numEdges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        Node* newNode = createNode(dest);
        newNode->next = adjacencyList[src];
        adjacencyList[src] = newNode;
    }

    int visited[MAX_VERTICES];
    for (int i = 0; i < numVertices; i++) {
        visited[i] = 0;
    }

    int startVertex;
    printf("Enter the start vertex: ");
    scanf("%d", &startVertex);

    printf("Breadth-First Search traversal: ");
    bfs(startVertex, visited, adjacencyList);

    return 0;
}
```

Output



```
Enter the number of edges: 4
Enter the edges (format: <source vertex> <destination vertex>):
0 1
0 3
1 2
3 1
Enter the start vertex: 0
```

Conclusion

Thu, we learned how to implement Breadth First Search tree traversal algorithm in C program

Lab 4

Programme to implement Best First Search

Theory

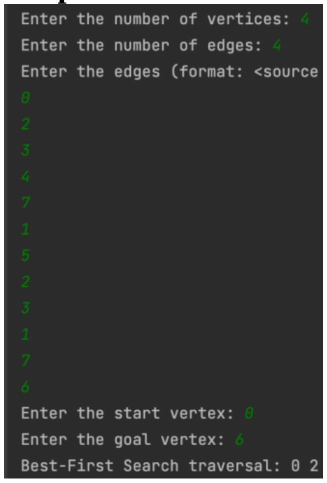
The Best-First Search (BFS) algorithm is a search algorithm that explores a graph or a tree based on an evaluation function. It evaluates each node based on a heuristic value and selects the most promising node to expand next.

Code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

typedef struct Node {
    int vertex;
    int heuristic;
    struct Node* next;
} Node;

// Create a new node
Node* createNode(int v, int h) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->heuristic = h;
    newNode->next = NULL;
    return newNode;
}

// Best-First Search function
void bestFirstSearch(int startVertex, int goalVertex, Node* adjacencyList[]) {
    int visited[MAX_VERTICES] = {0}; // Initialize visited array

    int queue[MAX_VERTICES];
    int front = 0, rear = -1;
    int found = 0; // Flag to check if the goal vertex is found

    visited[startVertex] = 1; // Mark the start vertex as visited
    queue[++rear] = startVertex; // Enqueue the start vertex

    while (front <= rear && !found) {
        int currentVertex = queue[front++]; // Dequeue a vertex from the queue
        printf("%d ", currentVertex); // Print the visited vertex
```

```
            if (currentVertex == goalVertex) {
                found = 1; // Goal vertex found, terminate the search
                break;
            }

            // Traverse all the adjacent vertices of the current vertex
            Node* temp = adjacencyList[currentVertex];
            while (temp != NULL) {
                int connectedVertex = temp->vertex;
                if (!visited[connectedVertex]) {
                    visited[connectedVertex] = 1; // Mark the adjacent vertex as visited
                    queue[++rear] = connectedVertex; // Enqueue the adjacent vertex
                }
                temp = temp->next;
            }
        }

        if (found) {
            printf("\nGoal vertex %d found!\n", goalVertex);
        } else {
            printf("\nGoal vertex %d not found!\n", goalVertex);
        }
    }

    int main() {
        int numVertices, numEdges;
        printf("Enter the number of vertices: ");
        scanf("%d", &numVertices);

        Node* adjacencyList[MAX_VERTICES];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = NULL; // Initialize the adjacency list
        }

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

        printf("Enter the edges (format: <source vertex> <destination vertex> <heuristic>):\n");
        for (int i = 0; i < numEdges; i++) {
            int src, dest, h;
            scanf("%d %d %d", &src, &dest, &h);
            Node* newNode = createNode(dest, h);
            newNode->next = adjacencyList[src];
            adjacencyList[src] = newNode;
        }

        int startVertex, goalVertex;
        printf("Enter the start vertex: ");
        scanf("%d", &startVertex);
        printf("Enter the goal vertex: ");
        scanf("%d", &goalVertex);

        printf("Best-First Search traversal: ");
        bestFirstSearch(startVertex, goalVertex, adjacencyList);
        return 0;
    }
```

Output



```
Enter the number of vertices: 4
Enter the number of edges: 4
Enter the edges (format: <source
0
2
3
4
7
1
5
2
3
1
7
6
Enter the start vertex: 0
Enter the goal vertex: 6
Best-First Search traversal: 0 2
```

Conclusion

Thus, we learned how to implement the Best First Search in the C programming language.

Lab 5

Introduction to Prolog

Theory

Prolog is a programming language for non-numeric computation. It is well suited for programmes 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.

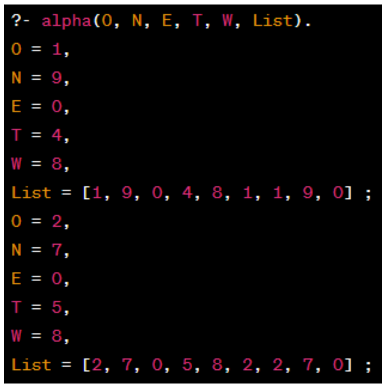
Prolog Programming is based on a few basic concepts like

- **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

```
alpha(O, N, E, T, W, [O, N, E, T, W, O, O, N, E]) :-
    between(1, 9, O),
    N \= 0,
    between(0, 9, E),
    T \= E, T \= N, T \= W,
    between(1, 9, W),
    W \= T, W \= E, W \= O, W \= N,
    ONE is O * 100 + N * 10 + E * 1,
    TWO is T * 100 + W * 10 + O * 1,
    TWO is ONE + ONE.
```

Output



```
?- alpha(O, N, E, T, W, List).
O = 1,
N = 9,
E = 0,
T = 4,
W = 8,
List = [1, 9, 0, 4, 8, 1, 1, 9, 0] ;
O = 2,
N = 7,
E = 0,
T = 5,
W = 8,
List = [2, 7, 0, 5, 8, 2, 2, 7, 0] ;
```

Conclusion

Thus, we learned the basic machinations of the Prolog Programming Language

Lab 6

Programme 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:

$P(A \mid B) = (P(B \mid A) * P(A)) / P(B)$

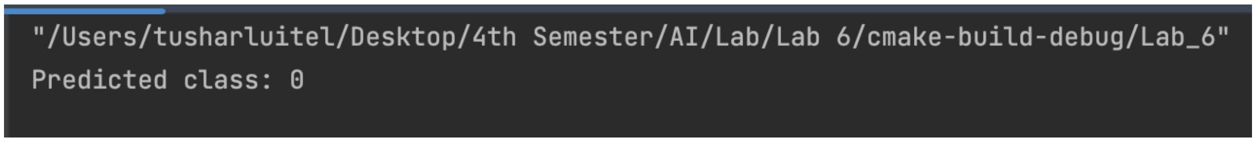
Where:

- $P(A \mid B)$ is the probability of event A occurring given that event B has occurred.
- $P(B \mid A)$ 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>
#define NUM_SAMPLES 5
#define NUM_FEATURES 4
#define NUM_CLASSES 3
// Training data
int trainingData[NUM_SAMPLES][NUM_FEATURES] = {
        {1, 1, 1, 0},
        {1, 0, 0, 0},
        {0, 1, 1, 0},
        {0, 1, 0, 1},
        {0, 0, 1, 1}
};
// Class labels
int classLabels[NUM_SAMPLES] = {1, 0, 1, 0, 1};
// Test data
int testData[NUM_FEATURES] = {1, 1, 0, 1};
// Function to calculate the probability of a feature given a class
double calculateFeatureProbability(int feature, int class) {
    int count = 0;
    for (int i = 0; i < NUM_SAMPLES; i++) {
        if (classLabels[i] == class && trainingData[i][feature] == testData[feature]) {
            count++;
        }
    }
    return (double)count / (double)(NUM_SAMPLES / 2);
}
// Function to calculate the probability of a class
double calculateClassProbability(int class) {
    int count = 0;
    for (int i = 0; i < NUM_SAMPLES; i++) {
        if (classLabels[i] == class) {
            count++;
        }
    }
    return (double)count / (double)NUM_SAMPLES;
}
// Function to classify the test data
int classify() {
    double classProbabilities[NUM_CLASSES];
    // Calculate the probability of each class
    for (int c = 0; c < NUM_CLASSES; c++) {
        double classProbability = calculateClassProbability(c);
        double featureProbabilities = 1.0;
        // Calculate the probability of each feature given the class
        for (int f = 0; f < NUM_FEATURES; f++) {
            double featureProbability = calculateFeatureProbability(f, c);
            featureProbabilities *= featureProbability;
        }
        classProbabilities[c] = classProbability * featureProbabilities;
    }

    int maxClass = 0;
    double maxProbability = classProbabilities[0];
    for (int c = 1; c < NUM_CLASSES; c++) {
        if (classProbabilities[c] > maxProbability) {
            maxClass = c;
            maxProbability = classProbabilities[c];
        }
    }
    return maxClass;
}
int main() {
    int predictedClass = classify();
    printf("Predicted class: %d\n", predictedClass);
    return 0;
}
```

Output



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 6/cmake-build-debug/Lab_6"
Predicted class: 0
```

Conclusion

Thus, we learned how to implement Naïve Baye’s Theorem in the C programming Language

Lab 7

Programme to implement Perceptron Learning Algorithm

Theory

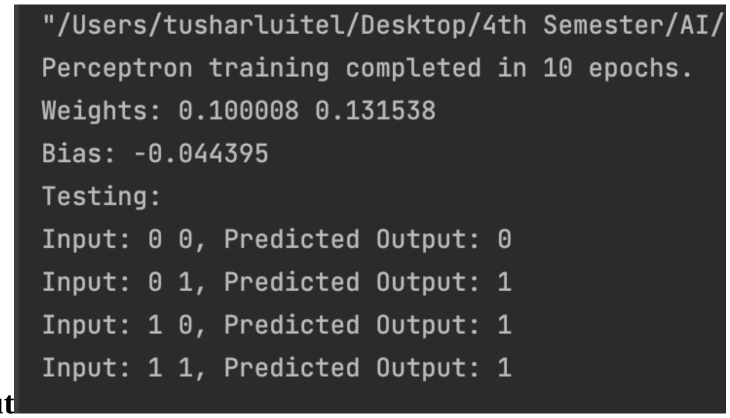
The Perceptron Learning Algorithm (PLA) is a simple supervised learning algorithm used for binary classification. It was introduced by Frank Rosenblatt in 1957 and is one of the earliest neural network algorithms.

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>
#include <stdlib.h>
#include <stdbool.h>

#define INPUT_SIZE 2
#define LEARNING_RATE 0.1
#define MAX_EPOCHS 100

// Training data for OR gate
int input[4][2] = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
int output[4] = {0, 1, 1, 1};

// Perceptron weights
double weights[INPUT_SIZE];
double bias;

// Function to initialize weights and bias
void initialize() {
    for (int i = 0; i < INPUT_SIZE; i++) {
        weights[i] = (double)rand() / RAND_MAX;
    }
    bias = (double)rand() / RAND_MAX;
}

// Activation function (Step function)
int activate(double net) {
    if (net >= 0) {
        return 1;
    } else {
        return 0;
    }
}
```

```
// Perceptron learning algorithm
void train() {
    initialize();
    int epochs = 0;
    bool error;

    do {
        error = false;

        for (int i = 0; i < 4; i++) {
            double net = bias;
            for (int j = 0; j < INPUT_SIZE; j++) {
                net += weights[j] * input[i][j];
            }
            int predictedOutput = activate(net);

            if (predictedOutput != output[i]) {
                error = true;

                for (int j = 0; j < INPUT_SIZE; j++) {
                    weights[j] += LEARNING_RATE * (output[i] - predictedOutput) * input[i][j];
                }
                bias += LEARNING_RATE * (output[i] - predictedOutput);
            }
        }

        epochs++;
    } while (error && epochs < MAX_EPOCHS);

    if (error) {
        printf("Perceptron training failed to converge within the maximum number of epochs.\n");
    } else {
        printf("Perceptron training completed in %d epochs.\n", epochs);
        printf("Weights: ");
        for (int i = 0; i < INPUT_SIZE; i++) {
            printf("%f ", weights[i]);
        }
        printf("\nBias: %f\n", bias);
    }
}

// Function to test the trained perceptron
void test(int x, int y) {
    double net = bias + weights[0] * x + weights[1] * y;
    int predictedOutput = activate(net);
    printf("Input: %d %d, Predicted Output: %d\n", x, y, predictedOutput);
}

int main() {
    train();

    printf("Testing:\n");
    test(0, 0);
    test(0, 1);
    test(1, 0);
    test(1, 1);
    return 0;
}
```



```
"/Users/tusharluitel/Desktop/4th Semester/AI/
Perceptron training completed in 10 epochs.
Weights: 0.100008 0.131538
Bias: -0.044395
Testing:
Input: 0 0, Predicted Output: 0
Input: 0 1, Predicted Output: 1
Input: 1 0, Predicted Output: 1
Input: 1 1, Predicted Output: 1
```

Output

Conclusion

Thus, we learned to implement Perceptron Learning Algorithm in C Programming Language

Lab 8

Implement Back Propagation Algorithm in XOR Gate

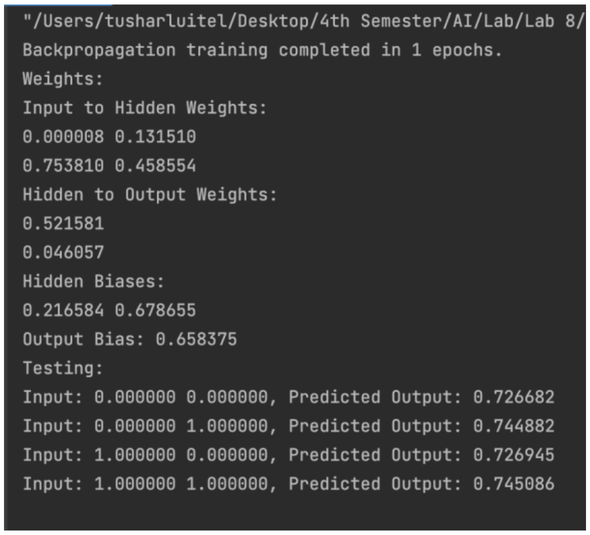
Theory

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_EPOCHS 10000
#define THRESHOLD 0.5

double sigmoid(double x) {
    return 1.0 / (1.0 + exp(-x));
}

double sigmoidDerivative(double x) {
    return sigmoid(x) * (1.0 - sigmoid(x));
}

double inputHiddenWeights[INPUT_SIZE][HIDDEN_SIZE];
double hiddenOutputWeights[HIDDEN_SIZE][OUTPUT_SIZE];
double hiddenBiases[HIDDEN_SIZE];
double outputBias;

double forwardPropagation(double input[]) {
    double hiddenOutputs[HIDDEN_SIZE];
    double output = 0.0;

    for (int i = 0; i < HIDDEN_SIZE; i++) {
        double net = hiddenBiases[i];
        for (int j = 0; j < INPUT_SIZE; j++) {
            net += input[j] * inputHiddenWeights[j][i];
        }
        hiddenOutputs[i] = sigmoid(net);
    }

    for (int i = 0; i < OUTPUT_SIZE; i++) {
        double net = outputBias;
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            net += hiddenOutputs[j] * hiddenOutputWeights[j][i];
        }
        output = sigmoid(net);
    }
    return output;
}

void backPropagation(double inputs[][INPUT_SIZE], double targets[]) {
    for (int i = 0; i < INPUT_SIZE; i++) {
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            inputHiddenWeights[i][j] = (double)rand() / RAND_MAX;
        }
    }

    for (int i = 0; i < HIDDEN_SIZE; i++) {
        for (int j = 0; j < OUTPUT_SIZE; j++) {
            hiddenOutputWeights[i][j] = (double)rand() / RAND_MAX;
        }
        hiddenBiases[i] = (double)rand() / RAND_MAX;
    }

    outputBias = (double)rand() / RAND_MAX;

    int epochs = 0;
    double error;

    do {
        error = 0.0;

        // Perform forward propagation and backpropagation for each training sample
        for (int sample = 0; sample < 4; sample++) {
            double input[INPUT_SIZE];
            for (int i = 0; i < INPUT_SIZE; i++) {
                input[i] = inputs[sample][i];
            }

            double target = targets[sample];

            // Forward propagation
            double output = forwardPropagation(input);

            // Backpropagation
            double outputError = target - output;
            error += outputError * outputError;

            // Update output layer weights and bias
            for (int i = 0; i < HIDDEN_SIZE; i++) {
                double deltaOutputWeight = LEARNING_RATE * outputError * hiddenOutputWeights[i][0] *
sigmoidDerivative(output);
                hiddenOutputWeights[i][0] += deltaOutputWeight;
            }

            // Update hidden layer weights and biases
            for (int i = 0; i < HIDDEN_SIZE; i++) {
                double hiddenError = outputError * hiddenOutputWeights[i][0] * sigmoidDerivative(output);
                for (int j = 0; j < INPUT_SIZE; j++) {
                    double deltaHiddenWeight = LEARNING_RATE * hiddenError * inputHiddenWeights[j][i] *
sigmoidDerivative(output);
                    inputHiddenWeights[j][i] += deltaHiddenWeight;
                }
                double deltaHiddenBias = LEARNING_RATE * hiddenError * sigmoidDerivative(output);
                hiddenBiases[i] += deltaHiddenBias;
            }

            // Update output bias
            double deltaOutputBias = LEARNING_RATE * outputError * sigmoidDerivative(output);
            outputBias += deltaOutputBias;
        }

        error /= 4;  // Mean squared error

        epochs++;
    } while (error > THRESHOLD && epochs < MAX_EPOCHS);

    if (epochs == MAX_EPOCHS) {
        printf("Backpropagation failed to converge within the maximum number of epochs.\n");
    } else {
        printf("Backpropagation training completed in %d epochs.\n", epochs);
        printf("Weights:\n");

        printf("Input to Hidden Weights:\n");
        for (int i = 0; i < INPUT_SIZE; i++) {
            for (int j = 0; j < HIDDEN_SIZE; j++) {
                printf("%f ", inputHiddenWeights[i][j]);
            }
            printf("\n");
        }

        printf("Hidden to Output Weights:\n");
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            for (int j = 0; j < OUTPUT_SIZE; j++) {
                printf("%f ", hiddenOutputWeights[i][j]);
            }
            printf("\n");
        }

        printf("Hidden Biases:\n");
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            printf("%f ", hiddenBiases[i]);
        }
        printf("\n");

        printf("Output Bias: %f\n", outputBias);
    }
}
int main() {
    // Training data for XOR gate
    double inputs[4][INPUT_SIZE] = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
    double targets[4] = {0, 1, 1, 0};

    backPropagation(inputs, targets);

    // Test the trained network
    printf("Testing:\n");
    for (int sample = 0; sample < 4; sample++) {
        double input[INPUT_SIZE];
        for (int i = 0; i < INPUT_SIZE; i++) {
            input[i] = inputs[sample][i];
        }
        double output = forwardPropagation(input);
        printf("Input: %f %f, Predicted Output: %f\n", input[0], input[1], output);
    }

    return 0;
}
```

Output



```
"/Users/tusharluitel/Desktop/4th Semester/AI/Lab/Lab 8/
Backpropagation training completed in 1 epochs.
Weights:
Input to Hidden Weights:
0.000008 0.131510
0.753810 0.458554
Hidden to Output Weights:
0.521581
0.046057
Hidden Biases:
0.216584 0.678655
Output Bias: 0.658375
Testing:
Input: 0.000000 0.000000, Predicted Output: 0.726682
Input: 0.000000 1.000000, Predicted Output: 0.744882
Input: 1.000000 0.000000, Predicted Output: 0.726945
Input: 1.000000 1.000000, Predicted Output: 0.745086
```

Conclusion

Thus, we learned to implement Back Propagation Algorithm for XOR Gates in the C programming language.

Lab 9

Implement a Family Tree in Prolog

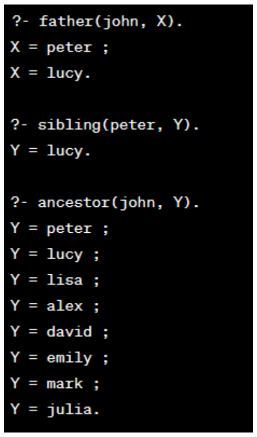
Theory

Prolog is a programming language for non-numeric computation. It is well suited for programmes 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.

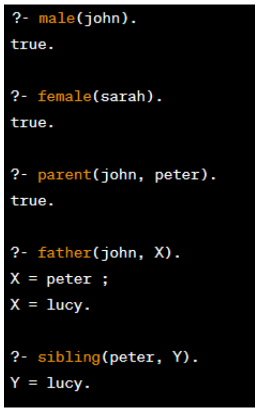
Code

```
/* Facts */
male(john).
male(peter).
male(michael).
male(alex).
male(david).
male(mark).
female(sarah).
female(lucy).
female(lisa).
female(julia).
female(emily).
/* Parent relationship */
parent(john, peter).
parent(john, lucy).
parent(sarah, peter).
parent(sarah, lucy).
parent(peter, lisa).
parent(peter, alex).
parent(peter, david).
parent(lucy, mark).
parent(lucy, julia).
parent(lisa, emily).
/* Rules */
father(X, Y) :- male(X), parent(X, Y).
mother(X, Y) :- female(X), parent(X, Y).
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
grandfather(X, Y) :- male(X), grandparent(X, Y).
grandmother(X, Y) :- female(X), grandparent(X, Y).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Output



```
?- father(john, X).
X = peter ;
X = lucy.

?- sibling(peter, Y).
Y = lucy.

?- ancestor(john, Y).
Y = peter ;
Y = lucy ;
Y = lisa ;
Y = alex ;
Y = david ;
Y = emily ;
Y = mark ;
Y = julia.
```



```
?- male(john).
true.

?- female(sarah).
true.

?- parent(john, peter).
true.

?- father(john, X).
X = peter ;
X = lucy.

?- sibling(peter, Y).
Y = lucy.
```

Conclusion

Thus, we finally practiced to use family tree in Prolog.