



TRIBHUVAN UNIVERSITY

**Institute of Engineering
Central Campus, Pulchowk
Pulchowk, Lalitpur**



**A Lab Report
of
Artificial Intelligence**

Submitted by:

Mamata Maharjan (077BCT043)
Group (B)

Submitted to:

Department of Electronics and Computer Engineering,
Pulchowk Campus

Submission Date: 2024/02/29

LAB No: 1

Introduction to Prolog

Prolog is a very important tool in programming artificial intelligence applications and in the development of expert systems. By allowing the programmer to model the logical relationships among objects and processes, complex problems are inherently easier to solve, and the resulting program is easier to maintain through its lifecycle. With Visual Prolog, applications such as customized knowledge bases, expert systems, natural language interfaces, and smart information management systems are easy to develop. Prolog is what is known as a declarative language. This means that given the necessary facts and rules, Prolog will use deductive reasoning to solve the programming problems. This is in contrast to traditional computer languages, such as C, BASIC and Pascal, which are procedural languages. We can also use prolog as any other programming languages in a procedural manner.

So prolog can be viewed as a tool to solve problems in the field of artificial intelligence or it can be very well used a general programming language. Prolog enforces a different problem solving paradigm complementary to traditional programming languages so it is believed that a student of computer should learn programming in prolog.

Data Types in Prolog

- Atoms and numbers
- Variables
- Structures

Atoms and Numbers:

Atoms can be constructed in three different ways

1. strings of letters, digits, and the underscore character ‘_’ starting with a lower case

letter.

for example: man, ram, comp_students, pc_ct_059.

2. strings of special characters

for example: <-----> :::::::::::

Care should be taken not to use the character combination that may have some built in meaning.

3. strings of characters enclosed in quotes

for example: 'Ram' 'Bird'

Numbers used in prolog are integers and real numbers.

Variables

Variables are strings of letters digits and underscore that start with an underscore or an upper-case letter. The scope of a variable is one clause only. So the same variable used in different clauses mean different thing.

For example: X, Ram, _weight etc.

Note here that Ram is a variable unlike the earlier use 'Ram' where it was a constant, an atom. An underscore '_' also known as anonymous variable is used in clauses when a variable need not be inferred to more than once.

Structures

Structures are objects that have different components. The components can be atoms or yet some other structures. A functor is used to construct a structure as follows.

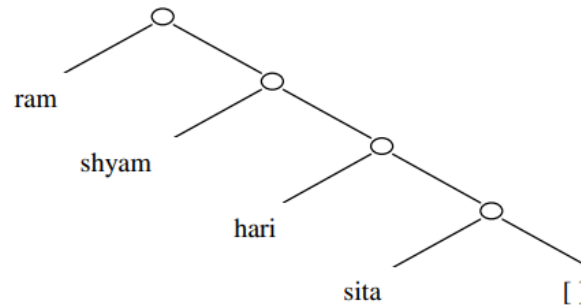
family(father, mother, children)

Here family is a structure that has father, mother and the children as its elements. The father and mother may be atoms while the children may be yet another structure or a list of atoms. List is a special built in structure in prolog.

A **list** is a built in structure in prolog. It can be thought of as a sequence of elements ordered linearly however it is internally represented as a binary tree.

For example: [ram,shyam,hari,sita]

The representation is as follows:



The list as such can be broken down into two parts, the **HEAD** and the **TAIL**. The head is the first element of the list and the tail is the remaining list. The above list can be broken down as:

[H| T]

Where H= ram and T= [shyam, hari,sita]

List is one of the most useful structure in prolog.

Writing Programs in Prolog

All prolog programs start from the goal. It then uses the facts and clauses to break down the goal to sub-goals and tries to prove the subgoals. A clause is said to have succeeded if there is a combination of facts and clause(s) that holds true. Prolog has built in backtracking mechanism i.e. whenever there are multiple paths, it chooses one tries to prove it, and comes back to the other choices whether the first one succeeds or fails.

Assignments

- 1) Write a program to find the hcf of two numbers.

Hint: use a predicate like hcf(no1,no2,result)

```
PREDICATES
HCF(integer, integer, integer)

CLAUSES
HCF(X, Y, Z):- Y>X, HCF(Y, X, Z).
HCF(X, Y, Z):- Y=0, Z=X.
HCF(X, Y, Z):- X=0, Z=Y.
HCF(X, Y, Z):- X>Y, R = X mod Y, HCF(Y, R, Z).

GOAL
HCF(18, 20, X).

OUTPUT:
X=2
```

Analysis and Discussion: The program follows Euclid's Algorithm for determining HCF of two numbers, where an error related to 'is' and '=' were encountered. Also the program didn't give any output when the fourth clause was placed at the top. After all the considerations and debugging, the program provided an output of X=2.

- 2) Write a program of your choice. Give some facts and use some rules to make a few deductions.

```
PREDICATES
PROFESSOR(String, String)
FRIEND(String, String)
GUIDE(String, String)
STUDENT(String, String)

CLAUSES
PROFESSOR("BISCUIT", "KILLUA").
PROFESSOR("BISCUIT", "GON").
PROFESSOR("WING", "ZUSHI").

FRIEND("GON", "KILLUA").
FRIEND("GON", "ZUSHI").
```

```

FRIEND("GON", "KURAPIKA").

STUDENT(A,B):- PROFESSOR(B,A).
STUDENT(A,C):- PROFESSOR(C,B),FRIEND(B,A).

GUIDE(A,C):- PROFESSOR(A,B),FRIEND(B,C).

GOAL
GUIDE(X, "KURAPIKA").

```

OUTPUT:

```

X=BISCUIT
1 Solution

```

Analysis and Discussion: The predicate professor and friend are here to assert facts where as predicate student and guide are here to define some rules. On our own evaluation Biscuit Krueger is indeed a guide of Kurapika Kurta in reference to a popular anime series HXH.

3) Write a program to add the content of an integer list and display it.

```

DOMAINS
INT_LIST=integer*

PREDICATES
ADD_LIST(INT_LIST, integer)

CLAUSES
ADD_LIST([],0).
ADD_LIST([H|T],A):-
    ADD_LIST(T,A1),A=H+A1.

GOAL
ADD_LIST([1,2,3,4],X).

```

OUTPUT:

```

X=10
1 Solution

```

Analysis and Discussion: The program adds up all the integers provided in the list and displayed the answer.

4) Write a program to find the length of a list.

```
DOMAINS
INT_LIST=integer*

PREDICATES
LENGTH(INT_LIST, integer)

CLAUSES
LENGTH([],0).
LENGTH([H|T],L):-
    LENGTH(T,L1),L=L1+1.

GOAL
LENGTH([1,2,3,4],X).
```

OUTPUT:

```
X=4
1 Solution
```

Analysis and Discussion: The program counts all the integers provided in the list and displays the answer.

5) Write a program to append two lists.

```
DOMAINS
LIST=integer*

PREDICATES
APPEND(LIST, LIST, LIST)

CLAUSES
APPEND([],LIST,LIST).
APPEND([H|L1], LIST,[H|L3]):-
    APPEND(L1, LIST, L3).

GOAL
APPEND([1,2,3,4], [5,6], X).
```

OUTPUT:

```
X=[1,2,3,4,5,6]
1 Solution
```

Analysis and Discussion: The program appends given two lists. First taking heads of first list followed by a total data of the second list.

6) Write a program which takes a list of integers and displays only 1s and 2s. (If the input is [1,2,4,5,2,4,5,1,1] the solution list should be [1,2,2,1,1].)

```
LIST=integer*

PREDICATES
SELECTIVEDISPLAY(LIST,LIST)
UPDATE(LIST,LIST,LIST)
REVERSE(LIST,LIST,LIST)

CLAUSES
SELECTIVEDISPLAY(A, B):-
    UPDATE(A, [],TEMP), REVERSE(TEMP,[],B).
    UPDATE([], A, A).

UPDATE([H|T], A, B):-
    H<3,H>0, UPDATE(T,[H|A], B).

UPDATE([H|T], A, B):-
    H<=0, UPDATE(T,A,B).

UPDATE([H|T], A, B):-
    H>=3, UPDATE(T,A,B).
    REVERSE([],A,A).

REVERSE([H|T], A, R):-
    REVERSE(T, [H|A], R).

GOAL
SELECTIVEDISPLAY([1,1,2,3,4,1,1],X).
```

OUTPUT:

```
X=[1,1,2,1,1]
1 Solution
```

Analysis and Discussion: The program skips the integers that are greater than 2 and less than 1 when inserting them into new list in reverse order which was at the end reordered in properly using REVERSE().

7) Write a program to delete a given element from the list.

```
DOMAINS
LIST=integer*

PREDICATES
DELETEDSELECTED(LIST, integer, LIST)
UPDATE(LIST,LIST,LIST, integer, integer)
REVERSE(LIST,LIST,LIST)

CLAUSES
DELETEDSELECTED(A, D, B):-
    UPDATE(A, [], TEMP, D, 0), REVERSE(TEMP,[],B).
    UPDATE([], A, A, _, _).

UPDATE([H|T], A, B, D, I):-
    D = I, NEWI=I+1, UPDATE(T, A, B, D, NEWI).

UPDATE([H|T], A, B, D, I):-
    D<>I, NEWI = I+1, UPDATE(T, [H|A], B, D, NEWI).
    REVERSE([], A, A).

REVERSE([H|T], A, R):-
    REVERSE(T, [H|A], R).

GOAL
DELETEDSELECTED([1,1,2,3,4,1,1], 2, B).
```

OUTPUT:

```
B=[1,1,3,4,1,1]
1 Solution
```

Analysis and Discussion: The program deletes the single element that lies in the index provided at the GOAL. The index of each head is compared to the desired index and when matched it is skipped from being added to the new list. Finally, the list was reversed in proper order and solution was displayed.

CONCLUSION: All the challenge assignments were successfully tested in visual prolog. These lab1 experiments introduced us to the basics of PROLOG and familiarized us in logical programming.

LAB No: 2

Constraint Programming

Constraint programming is a useful tool in formulating and solving problems that can be defined in terms of constraint among a set of variables. In fact real world problems are defined in terms of some variables that bear some constraints. Finding a set of variables that are within the constraints given (or observed) is a solution to that problem.

Let us consider a problem, that can be represented by some relations of the variables x , y and z . We have a domain D_x , D_y , D_z from where the variables can take a value. The constraint is given by a set C and may have a number of constraints C_1 , C_2 , C_3 etc each relating some or all of the variables x , y , z . Now a solution (or solutions) to the problem is a set of the problem is a set $dx \in D_x$, $dy \in D_y$, $dz \in D_z$ and all the constraints of set C are satisfied.

Crypto arithmetic problem

Crypto Arithmetic Problem is yet another constraint satisfaction problem. We have to assign numeric values (0 through 9) to the alphabets in the given words in such a way that the sum of the two words equals the third.

For example, **SEND+MORE=MONEY**

We have to assign values to the individual alphabets in such a way the arithmetic rules are followed, a trivial solution will be assign zeros to all but we have a constraint, no two alphabets should be assigned with the same number.

	C4	C3	C2	C1	
		S	E	N	D
+		M	O	R	E
<hr/>					
	M	O	N	E	Y

Now domain for alphabet is given by:

$S, E, N, D, M, O, R, Y \in \{0,1,2,3,4,5,6,7,8,9\}$.

The constraints are:

$D+E=Y+10C1$

$N+R+C1=E+10C2$

$E+O+C2=N+10C3$

$S+M+C3=O+10C4$

$M=C4$

$C1, C2, C3, C4 \in \{0,1\}$

And we have the constraint that no two alphabets should be assigned to the same number.

PROGRAM: 1

```
DOMAINS
int_list=integer*

PREDICATES
solution(int_list)
member(integer,int_list)

CLAUSES
solution([]).
solution([S,E,N,D,M,O,R,Y]):-
C4=1, %c4 must be 1 otherwise no need to mention M.
member(C1,[0,1]),
member(C2,[0,1]),
member(C3,[0,1]),
% C1, C2, C3 will have values 0 or 1
member(E,[0,1,2,3,4,5,6,7,8,9]),
member(N,[0,1,2,3,4,5,6,7,8,9]),
member(D,[0,1,2,3,4,5,6,7,8,9]),
member(M,[0,1,2,3,4,5,6,7,8,9]),
member(O,[0,1,2,3,4,5,6,7,8,9]),
member(R,[0,1,2,3,4,5,6,7,8,9]),
member(Y,[0,1,2,3,4,5,6,7,8,9]),
member(S,[0,1,2,3,4,5,6,7,8,9]),
% S,E,N, D, M, O, R, Y will have values between 0 and 9.
% The values of S,E,N, D, M, O, R, Y must not be equal.
```

```

S<>E, S<>N, S<>D, S<>M, S<>O, S<>R, S<>Y,
E<>N, E<>D, E<>M, E<>O, E<>R, E<>Y,
N<>D, N<>M, N<>O, N<>R, N<>Y,
D<>M, D<>O, D<>R, D<>Y,
M<>O, M<>R, M<>Y,
O<>R, O<>Y,
R<>Y,
% Computation for solution
D+E=Y+10*C1,
N+R+C1=E+10*C2,
E+O+C2=N+10*C3,
S+M+C3=O+10*C4,
M=C4.

member(X, [X|_]).
member(X, [_|Z]):-
    member(X,Z).

GOAL
solution([S,E,N,D,M,O,R,Y]).

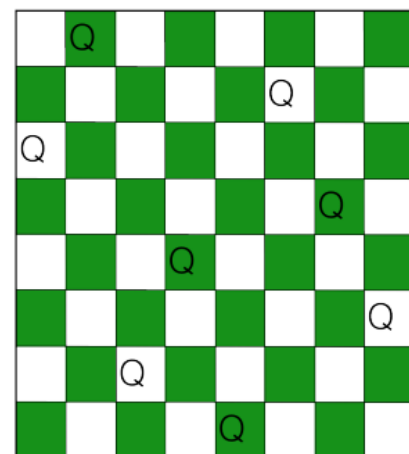
```

Eight Queens Problem

Eight queens problem is a constraint satisfaction problem. The task is to place eight queens in the 64 available squares in such a way that no queen attacks each other. So the problem can be formulated with variables $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ and $y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8$; the x s represent the rows and y s the column. Now a solution for this problem is to assign values for x and y such that the constraint is satisfied.

The problem can be formulated as $P = \{(x_1, y_1), (x_2, y_2) \dots (x_8, y_8)\}$ where (x_1, y_1) gives the position of the first queen and so on.

So it can be clearly seen that the domains for x_i and y_i are $D_x = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $D_y = \{1, 2, 3, 4, 5, 6, 7, 8\}$ Respectively.



The constraints are

- 1) No two queens should be in the same row. That is, $y_i \neq y_j$ for $i = 1$ to 8 ; $j = 1$ to 8 ; $i \neq j$
- 2) No two queens should be in the same columns. That is, $x_i \neq x_j$ for $i = 1$ to 8 ; $j = 1$ to 8 ; $i \neq j$
- 3) There should not be two queens placed on the same diagonal line.

That is, $(y_i - y_j) \neq \pm (x_i - x_j)$.

Now a solution to this problem is an instance of P wherein the above mentioned constraints are satisfied.

PROGRAM: 2

```
DOMAINS
cell = c(integer, integer)
list = cell*
int_list = integer*

PREDICATES
solution(list)
member(integer, int_list)
noattack(cell, list)

CLAUSES
solution([]).

solution([c(X,Y)|Others]):-
    solution(Others),
    member(Y,[1,2,3,4,5,6,7,8]),
    noattack(c(X,Y),Others).
    noattack(_,[]).

noattack(c(X,Y),[c(X1,Y1)|Others]):-
    Y<>Y1,
    Y1-Y<>X1-X,
    Y1-Y<>X-X1,
    noattack(c(X,Y),Others).
    member(X,[X|_]).

member(X,[_|Z]):-
    member(X,Z).

GOAL
solution([c(1,A),c(2,B),c(3,C),c(4,D),c(5,E),c(6,F),c(7,G),c(8,H)]).
```

Assignments

1) Make yourself a crypto arithmetic problem as above and find the solution.

```
C5 C4 C3 C2 C1
B A N A N A
+ G U A V A
-----
O R A N G E
```

```
%SOURCE CODE
DOMAINS
int_list=integer*

PREDICATES
solution(int_list)
member(integer,int_list)

CLAUSES
solution([]).

solution([B,A,N,G,V,U,O,R,E]):-
    C5 = 1,
    member(C1, [0,1]),
    member(C2, [0,1]),
    member(C3, [0,1]),
    member(C4, [0,1]),

    member(B,[0,1,2,3,4,5,6,7,8,9]),
    member(A,[0,1,2,3,4,5,6,7,8,9]),
    member(N,[0,1,2,3,4,5,6,7,8,9]),
    member(G,[0,1,2,3,4,5,6,7,8,9]),
    member(U,[0,1,2,3,4,5,6,7,8,9]),
    member(V,[0,1,2,3,4,5,6,7,8,9]),
    member(O,[0,1,2,3,4,5,6,7,8,9]),
    member(R,[0,1,2,3,4,5,6,7,8,9]),
    member(E,[0,1,2,3,4,5,6,7,8,9]),

    B<>N,B<>A,B<>G,B<>V,B<>U,B<>O,B<>R,B<>E,
    A<>N,A<>G,A<>U,A<>V,A<>O,A<>R,A<>E,
    N<>G,N<>V,N<>U,N<>O,N<>R,N<>E,
    G<>V,G<>U,G<>O,G<>R,G<>E,
    V<>U,V<>O,V<>R,V<>E,
```

```

U<>O,U<>R,U<>E,
O<>R,O<>E,
R<>E,

```

```

A + A = E + 10*C1,
N + V + C1 = G + 10*C2,
A + A + C2 = N + 10*C3,
N + U + C3 = A + 10*C4,
A + G + C4 = R + 10*C5,
B + C5 = 0.

```

```

member(X, [X|_]).

```

```

member(X, [_|Z]):-
    member(X,Z).

```

```

GOAL
solution([B,A,N,G,U,V,O,R,E]).

```

OUTPUT: -

```

B=2, A=4, N=9, G=6, U=7, V=5, O=3, R=1, E=8

```

Analysis and Discussion: The value of C5 is assigned 1 and the other carries C1, C2, C3 and C4 will have values 0 or 1. B, A, N, G, U, V, O, R, E will have values between 0 and 9, where none of them have the same value. The required equations for the computation in provided and GOAL is set to find out the values assigned to each alphabet of the problem.

2) Observe the result of the above program and discuss on the result. Test the goal by placing a few queens explicitly.

(Try goals like solution ([c(1,1),c(2,B),c(3,C),c(4,8),c(5,E),c(6,F),c(7,G),c(8,H)] etc.)

```

GOAL
solution([c(1,A),c(2,B),c(3,C),c(4,D),c(5,E),c(6,F),c(7,G),c(8,H)]).

```

OUTPUT:

```

A=5, B=7, C=2, D=6, E=3, F=1, G=4, H=8
92 Solutions

```

```

GOAL
solution ([c(1,1),c(2,B),c(3,C),c(4,8),c(5,E),c(6,F),c(7,G),c(8,H)]).

```

OUTPUT:

```
B=7, C=5, E=2, F=4, G=6, H=3  
1 Solution
```

GOAL

solution ([c(1,2),c(2,B),c(3,C),c(4,8),c(5,E),c(6,F),c(7,G),c(8,H)]).

OUTPUT:

```
B=7, C=5, E=1, F=4, G=6, H=3  
B=4, C=6, E=3, F=1, G=7, H=5  
2 Solutions
```

Analysis and Discussion: The Program 2 code provided the solutions for the eight queen problem. It was observed that the 8 queens could be arranged in 92 different arrangements among which a single solution was displayed as output. Similarly, some queens were placed explicitly and the program provided the positions of other queens to be placed in order to solve the eight queen problem.

3) Try to solve the above problem using C, C++.

```
//eightQueens.cpp  
#include <iostream>  
#include <vector>  
#include <cmath>  
  
struct cell {  
    int x, y;  
};  
  
class EightQueensSolver {  
private:  
    int boardSize;  
    std::vector<std::vector<cell>> solutions;  
  
public:  
    EightQueensSolver(int size) : boardSize(size) {}
```



```

bool noAttack(const std::vector<cell>& queens, const cell& candidate) {
    for (const auto& queen : queens) {
        if (queen.x == candidate.x || queen.y == candidate.y ||
            std::abs(candidate.y - queen.y) == std::abs(candidate.x -
queen.x)) {
            return false;
        }
    }
    return true;
}

void solve(int row, std::vector<cell>& queens) {
    if (row == boardSize) {
        solutions.push_back(queens);
        return;
    }

    for (int col = 0; col < boardSize; ++col) {
        cell candidate = {row, col};
        if (noAttack(queens, candidate)) {
            queens.push_back(candidate);
            solve(row + 1, queens);
            queens.pop_back(); // Backtrack
        }
    }
}

void solve() {
    std::vector<cell> queens;
    solve(0, queens);
}

void displaySolutions() const {
    for (const auto& solution : solutions) {
        for (const auto& queen : solution) {
            std::cout << "( " << queen.x << " , " << queen.y << " ) ";
        }
        std::cout << "\n";
    }
}

```

```

    }
};

int main() {
    EightQueensSolver solver(8);
    solver.solve();
    solver.displaySolutions();

    return 0;
}

```

OUTPUT:

```

( 0 , 0 ) ( 1 , 4 ) ( 2 , 7 ) ( 3 , 5 ) ( 4 , 2 ) ( 5 , 6 ) ( 6 , 1 ) ( 7 , 3 )
( 0 , 0 ) ( 1 , 5 ) ( 2 , 7 ) ( 3 , 2 ) ( 4 , 6 ) ( 5 , 3 ) ( 6 , 1 ) ( 7 , 4 )
( 0 , 0 ) ( 1 , 6 ) ( 2 , 3 ) ( 3 , 5 ) ( 4 , 7 ) ( 5 , 1 ) ( 6 , 4 ) ( 7 , 2 )
( 0 , 0 ) ( 1 , 6 ) ( 2 , 4 ) ( 3 , 7 ) ( 4 , 1 ) ( 5 , 3 ) ( 6 , 5 ) ( 7 , 2 )
( 0 , 1 ) ( 1 , 3 ) ( 2 , 5 ) ( 3 , 7 ) ( 4 , 2 ) ( 5 , 0 ) ( 6 , 6 ) ( 7 , 4 )
( 0 , 1 ) ( 1 , 4 ) ( 2 , 6 ) ( 3 , 0 ) ( 4 , 2 ) ( 5 , 7 ) ( 6 , 5 ) ( 7 , 3 )
( 0 , 1 ) ( 1 , 4 ) ( 2 , 6 ) ( 3 , 3 ) ( 4 , 0 ) ( 5 , 7 ) ( 6 , 5 ) ( 7 , 2 )
( 0 , 1 ) ( 1 , 5 ) ( 2 , 0 ) ( 3 , 6 ) ( 4 , 3 ) ( 5 , 7 ) ( 6 , 2 ) ( 7 , 4 )
( 0 , 1 ) ( 1 , 5 ) ( 2 , 7 ) ( 3 , 2 ) ( 4 , 0 ) ( 5 , 3 ) ( 6 , 6 ) ( 7 , 4 )
( 0 , 1 ) ( 1 , 6 ) ( 2 , 2 ) ( 3 , 5 ) ( 4 , 7 ) ( 5 , 4 ) ( 6 , 0 ) ( 7 , 3 )
( 0 , 1 ) ( 1 , 6 ) ( 2 , 4 ) ( 3 , 7 ) ( 4 , 0 ) ( 5 , 3 ) ( 6 , 5 ) ( 7 , 2 )
( 0 , 1 ) ( 1 , 7 ) ( 2 , 5 ) ( 3 , 0 ) ( 4 , 2 ) ( 5 , 4 ) ( 6 , 6 ) ( 7 , 3 )
( 0 , 2 ) ( 1 , 0 ) ( 2 , 6 ) ( 3 , 4 ) ( 4 , 7 ) ( 5 , 1 ) ( 6 , 3 ) ( 7 , 5 )
( 0 , 2 ) ( 1 , 4 ) ( 2 , 1 ) ( 3 , 7 ) ( 4 , 0 ) ( 5 , 6 ) ( 6 , 3 ) ( 7 , 5 )
( 0 , 2 ) ( 1 , 4 ) ( 2 , 1 ) ( 3 , 7 ) ( 4 , 5 ) ( 5 , 3 ) ( 6 , 6 ) ( 7 , 0 )
( 0 , 2 ) ( 1 , 4 ) ( 2 , 6 ) ( 3 , 0 ) ( 4 , 3 ) ( 5 , 1 ) ( 6 , 7 ) ( 7 , 5 )
( 0 , 2 ) ( 1 , 4 ) ( 2 , 7 ) ( 3 , 3 ) ( 4 , 0 ) ( 5 , 6 ) ( 6 , 1 ) ( 7 , 5 ) . . .
94 SOLUTIONS IN TOTAL

```

Analysis and Discussion: The above c++ program defines a class, ‘EightQueenSolver’, to solve the classic N-Queens problem using backtracking. The solutions are stored in vectors of vectors and the backtracking algorithm explores possible configurations and stores valid solutions. The ‘displaySolutions’ function outputs the found solutions. While the code provides solutions for multiple board sizes it lacks in optimization.

CONCLUSION: This lab focused on constraint programming in Prolog, demonstrating its practical use through examples like the Crypto Arithmetic Problem and the Eight Queens Problem. The exporting of Dll files were however not supported by the visual prolog 5.1 personal edition. Hence, We learned to express and solve problems by setting constraints on variables, showcasing Prolog's versatility.

LAB No: 3

First Order Predicate Logic (FOPL)

Introduction

First-order predicate logic (FOPL), also known as predicate calculus, has a rich history in knowledge representation, predating modern computers by decades. However, its practical application for representing and manipulating knowledge gained prominence in the early 1960s. In the field of Artificial Intelligence (AI), FOPL plays a crucial role. For AI students, understanding FOPL offers several advantages. Firstly, logic provides a formal and theoretically sound approach to reasoning. Secondly, FOPL's structure is flexible enough to accurately represent natural language, making it a valuable tool for expressing knowledge in AI systems. In simpler terms, FOPL serves as a powerful and flexible language for computers to understand and reason about information in a structured and systematic way.

For Example:

Ram loves all animals.

$$\forall x \text{Animals}(x) \Rightarrow \text{Loves}(\text{ram}, x)$$

Poppy is a dog.

Dog (Poppy)

Grandparent is a parent of one's parent

$$\forall x, y \text{Grandparent}(x, y) \Leftrightarrow \exists z \text{Parent}(x, z) \cap \text{Parent}(z, y)$$

Parent and child are inverse relation.

$$\forall x, y \text{Parent}(x, y) \Leftrightarrow \text{Child}(y, x)$$

Rules combine facts to increase knowledge of the system

$\text{son}(X, Y) :- \text{male}(X), \text{child}(X).$

X is a son of Y if X is male and X is a child of Y.

Conversion of Prolog into FOPL:

Prolog clauses can be directly translated into FOPL, except for a few exceptions like write, !, is, assert, retract, ...

The three simple rules for conversion are:

- ",", " corresponds to "&"
- ":-" corresponds to "<-"
- All variables are universally quantified.

Examples

1. grandfather(X,Y) :- father(X,Z),parent(Z,Y).

In FOPL:

$\forall x, \forall y, \forall z \text{ (grandfather}(x,y) <- (\text{father}(x,z) \& \text{parent}(z,y)))$

2. uncle(X,Y) :- parent(Z,Y),brother(X,Z).

In FOPL:

$\forall x, \forall y, \forall z \text{ (uncle}(x,y) <- (\text{parent}(z,y) \& \text{brother}(x,z)))$

3. member(X,[X|T]).

In FOPL:

$\forall x, \forall y, \forall t \text{ (member}(x,[x|t])$

4. member(X,[Y|T]) :- member(X,T).

In FOPL:

$\forall x, \forall y, \forall t \text{ (member}(x,[y|t]) <- \text{member}(x,t))$

5. a :- b,c,d,e.

In FOPL:

$a <- (b \& c \& d \& e)$

EXAMPLE 1: Monkey- Banana Problem

The Monkey-Banana Problem is a well-known challenge in AI.

Imagine a room with a monkey, a chair, and bananas hanging from the ceiling, just out of the monkey's reach. To solve this, the clever monkey can place the chair beneath the bananas, climb onto the chair, and then reach the bananas.

Now, let's use First Order Predicate Logic (FOPL) to represent this problem. In FOPL, we define essential objects like the monkey, the chair, and the bananas as well as their relationships. For instance, we might have predicates like "Monkey(x)," "Chair(y)," and "Bananas(z)." We can use logical statements to represent conditions such as "OnChair(x, y)" (the monkey is on the chair) and "Reachable(x, z)" (the bananas are reachable by the monkey). By expressing these relationships in FOPL, we can create a logical representation of the problem and use it to prove that the monkey can indeed reach the bananas. The program will encode these relationships and logic to demonstrate the solution step by step.

PROGRAM: 1

```
PREDICATES
in_room(symbol)
dexterous(symbol)
tall(symbol)
can_move(symbol,symbol,symbol)
can_reach(symbol,symbol)
get_on(symbol,symbol)
can_climb(symbol,symbol)
close(symbol,symbol)
under(symbol,symbol)

CLAUSES
in_room(bananas).
in_room(chair).
in_room(monkey).
dexterous(monkey).
tall(chair).
can_move(monkey,chair,bananas).
can_climb(monkey,chair).
can_reach(X,Y):- dexterous(X),close(X,Y).

close(X,Z):- get_on(X,Y), under(Y,Z), tall(Y).
```

```

get_on(X,Y):- can_climb(X,Y).
under(Y,Z):- in_room(X), in_room(Y), in_room(Z), can_move(X,Y,Z).

GOAL
can_reach(monkey,apple).

```

Example 2:

Write the following statements in FOPL form and by converting them into prolog program test the given goal.

1. Every American who sells weapons to hostile nations is a criminal.

$$\forall x(\text{American}(x) \wedge \text{sells}(x, \text{weapons}) \wedge \exists y (\text{hostile}(y) \wedge \text{sells}(x, y)) \Rightarrow \text{criminal}(x))$$

2. Every enemy of America is a hostile.

$$\exists x \text{ enemy}(\text{America}) \Rightarrow \text{hostile}(x)$$

3. Iraque has some missiles.

$$\exists m \text{ has_missiles}(\text{Iraque})$$

4. All missiles of Iraque were sold by George.

$$\forall m (\text{weapons}(m) \wedge \text{sold}(\text{George}, m)) \Rightarrow \text{has_missiles}(\text{Iraque})$$

5. George is an American.

$$\text{American}(\text{George})$$

6. Iraque is a country.

$$\text{Country}(\text{Iraque})$$

7. Iraque is the enemy of America.

$$\text{Enemy}(\text{America}, \text{Iraque})$$

8. Missiles are weapons.

$$\text{weapons}(\text{missiles})$$

PROGRAM: 2

PREDICATES

hostile(String)

enemy_of_america(String)

american(String)

criminal(String)

sells_missiles(String, String)

has_missile(String)

country(String)

CLAUSES

criminal(X):-

american(X), sells_missiles(X, Y),

hostile(Y).

enemy_of_america(X) :-

hostile(X).

enemy_of_america("Iraq").

hostile(X):-

country(X).

has_missile("Iraq").

sells_missiles("George", "Iraq").

american("George").

country("Iraq").

GOAL

criminal("George").

Assignments

1)PREMISES 1

1. Horses, cows, pigs are mammals.

$\text{mammal}(\text{Horse}) \wedge \text{mammal}(\text{cows}) \wedge \text{mammal}(\text{pigs})$

2. An offspring of a horse is a horse.

$\forall x, y (\text{offspring}(x, y) \wedge \text{Horse}(x) \Rightarrow \text{Horse}(y))$

3. Bluebeard is a horse.

$\text{Horse}(\text{Bluebeard})$

4. Bluebeard is Charlie's parent.

$\text{Horse}(\text{Bluebeard}) \wedge \text{parent}(\text{Bluebeard}, \text{Charlie})$

5. Offspring and parent are inverse relations.

$\forall x, y (\text{offspring}(x, y) \Leftrightarrow \text{parent}(y, x))$

6. Every mammal has a parent.

$\exists x (\text{mammal}(x) \wedge \forall y \text{parent}(y, x))$

QUERY

1. Is Charlie a horse?

$\text{Horse}(\text{Charlie}).$

```
%SOURCE CODE
PREDICATES
MAMMAL (STRING)
OFFSPRING (STRING, STRING)
HORSE (STRING)
COW (STRING)
PIG (STRING)
PARENT (STRING, STRING)

CLAUSES
MAMMAL (X) :- HORSE (X); COW (X); PIG (X).
PIG (X).
```



```

COW(X).

HORSE("Bluebeard").
HORSE(Y) :- OFFSPRING(X,Y) , HORSE(X).

PARENT("Bluebeard", "Charlie").
PARENT(_,X):-MAMMAL(X).

OFFSPRING(X, Y) :- PARENT(Y, X).

GOAL
HORSE("Charlie").

OUTPUT:-
yes

```

Analysis and Discussion: The premises were first converted into their respective predicate logics and implemented into prolog program. The query for if Charlie was a horse was replied as yes by this program, which is also valid since Charlie's parent bluebeard is a horse himself.

2) PREMISES 2

1. All people who are not poor and are smart are happy.

$$\forall x (\text{people}(x) \wedge \neg \text{poor}(x) \wedge \text{smart}(x) \Rightarrow \text{happy}(x))$$

2. Those people who read are not stupid.

$$\exists x (\text{people}(x) \wedge \text{can_read}(x) \wedge \neg \text{stupid}(x))$$

3. John can read and is wealthy.

$$\text{people}(\text{John}) \wedge \text{can_read}(\text{John}) \wedge \text{wealthy}(\text{John})$$

4. Happy people have exciting lives.

$$\forall x,y (\text{people}(x) \wedge \text{happy}(x) \Rightarrow \text{Exciting_life}(y))$$

QUESTION

1. Can anyone be found with an exciting life?

$$\exists x (\text{people}(x) \wedge \exists y (\text{Exciting}(y)))$$

```

PREDICATES
PEOPLE(STRING)
SMART(STRING)
HAPPY(STRING)
CAN_READ(STRING)
WEALTHY(STRING)
EXCITING_LIFE(STRING)

CLAUSES
    PEOPLE("JOHN").
    WEALTHY("JOHN").
    HAPPY(X) :- PEOPLE(X) , WEALTHY(X) , SMART(X).

    CAN_READ("JOHN").
    SMART(X) :- PEOPLE(X), CAN_READ(X).

    EXCITING_LIFE(_):- PEOPLE(X),HAPPY(X).

GOAL
    PEOPLE(X),EXCITING_LIFE(_).

OUTPUT :
X=JOHN
1 Solution

```

Analysis and Discussion: The Program had two predicates ‘not stupid ‘ and ‘ not poor’ which was replaced by ‘smart’ and ‘wealthy’ respectively. Here the query was to find anyone who has an exciting life and for which the person had to be happy, from all the provided conditions it is clear that people who are wealthy and smart or can read are happy ultimately proving that John was happy and had an exciting life.

3) PREMISES 3

1. All pompeians are romans.

$$\forall x (\text{pompeians}(x) \Rightarrow \text{romans}(x))$$

2. all romans were either loyal to Caesar or hated him.

$$\forall x, (\text{romans}(x) \Rightarrow (\text{loyal}(\text{Caesar}, x) \vee \text{hated}(\text{Caesar}, x)))$$

3. everyone is loyal to someone.

$\forall x \exists y (\text{loyal}(y, x))$

4. people only try to assassinate rulers they are not loyal to.

$\forall x, y (\text{people}(x) \wedge \text{ruler}(y) \wedge \text{try_assassinate}(x, y) \Rightarrow \neg \text{loyal}(y, x))$

5. marcus tried to assassinate Caesar.

`try_assassinate(marcus, caesar).`

6. marcus was Pompeian.

`pompeian(marcus)`

QUESTION

1. did marcus hate Caesar?

`hated(Caesar, marcus)`

```
%SOURCE CODE
PREDICATES
POMPEIANS(STRING)
ROMANS(STRING)
LOYAL(STRING, STRING)
PEOPLE(STRING)
RULER(STRING)
TRY_ASSASSINATE(STRING, STRING)
HATED(STRING, STRING)

CLAUSES
POMPEIANS("MARCUS").
ROMANS(X) :- POMPEIANS(X).
ROMANS(X) :- LOYAL("CAESER", X) ; HATED("CAESER", X).
PEOPLE(X).
RULER(Y).
HATED(X, Y).
LOYAL(Y, X).
TRY_ASSASSINATE("MARCUS", "CAESER").
TRY_ASSASSINATE(X, Y) :- PEOPLE(X), RULER(Y), not(LOYAL(Y, X)).

GOAL
HATED("CAESER", "MARCUS").
```

OUTPUT :

yes

Analysis and Discussion: The converted predicate logics were implemented in the prolog and the query for if Marcus hated Caesar was evaluated to 'yes'. Verifying the result with computerless logical evaluation, Marcus was a pompeian and tried to assassinate the ruler 'Caesar' which lead to Marcus not being loyal to Caesar and being a Roman citizen his disloyalty to Ceaser concludes in him hating Caesar.

4) PREMISES 4

1. Bhogendra likes all kinds of food.

$$\forall x (\text{food}(x) \Rightarrow \text{likes}(\text{"Bhogendra"}, x))$$

2. Oranges are food. Chicken is food.

$$\text{food}(\text{Orange}) \wedge \text{food}(\text{Chicken})$$

3. Anything anyone eats and isn't killed by is food.

$$\forall x, y (\text{person}(y) \wedge \text{eats}(y, x) \wedge \neg \text{killed_by}(y, x) \Rightarrow \text{food}(x))$$

4. If a person likes a food means that person has eaten it.

$$\forall x, y (\text{person}(y) \wedge \text{food}(x) \wedge \text{likes}(y, x) \Rightarrow \text{eats}(y, x))$$

5. Jogendra eats peanuts and is still alive.

$$\text{person}(\text{Jogendra}) \wedge \text{eats}(\text{Jogendra}, \text{peanuts}) \wedge \text{alive}(\text{jogendra})$$

6. Shailendra eats everything Bhogendra eats.

$$\forall x (\text{food}(x) \wedge \text{eats}(\text{Bhogendra}, x) \Rightarrow \text{eats}(\text{Shailendra}, x))$$

QUESTION

Does Shailendra like chicken.

$$\text{likes}(\text{Shainlendra}, \text{Chicken}).$$

```

%SOURCE CODE
PREDICATES
FOOD(String)
LIKES(String, String)
PERSON(String)
KILLED_BY(String, String)
EATS(String, String)
ALIVE(String)

CLAUSES
ALIVE("JHOGENDRA").
KILLED_BY("JHOGENDRA",Y) :- not(ALIVE("JHOGENDRA")).
FOOD(CHICKEN).
FOOD(ORANGE).

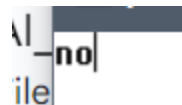
LIKES("BHOGEN德拉", X):- FOOD(X).
LIKES(Y,X) :- PERSON(Y), FOOD(X), EATS(Y,X).
PERSON("JOGENDRA").

EATS("JOGENDRA", "PEANUTS").
EATS("SHAIENDRA",X) :- FOOD(X), EATS("BHOGEN德拉", X).
EATS(Y,X) :- PERSON(Y), FOOD(X), not(KILLED_BY(Y,X)).

GOAL
LIKES("SAIENDRA", "CHICKEN").

```

OUTPUT:



Analysis and Discussion: The converted predicate logics were implemented in the prolog and the query for if Shailendra liked Chicken was evaluated as no by the program.

5) PREMISES 5

1. Dave and Fred are members of a dancing club.

$\text{member}(\text{Dave}) \wedge \text{member}(\text{fred})$

2. no member can both waltz and jive.

$\forall x (\text{member}(x) \Rightarrow \sim (\text{walts}(x) \vee \text{jive}(x)))$

3. Fred's dad can't waltz

$\neg \text{waltz}(\text{"dad"})$

4. Dave can do whatever fred can't do.

$\forall x (\neg(\text{jive}(x)) \Rightarrow \text{jive}(\text{"dave"}))$

5. If a child can do something, then their parents can do it also.

$\forall x,y (\text{child}(x, y) \Rightarrow (\text{walts}(x) \vee \text{jive}(x)) \Rightarrow (\text{walts}(y) \vee \text{jive}(y)))$

PROVE that there is a member of the dancing club who can't jive

$\exists x (\text{member}(x) \wedge \neg \text{jive}(x))$

```
%SOURCE CODE
PREDICATES
MEMBER(String)
CAN(String, String)
CANNOT(String, String)
PARENT(String, String)

CLAUSES
MEMBER("DAVE").
MEMBER("FRED").

PARENT("DAD", "FRED").

CANNOT("DAD", "WALTZ").
CANNOT(X,Y) :- PARENT(Z,X), CANNOT(Z,Y).
CANNOT(X,"WALTZ"):- MEMBER(X), CAN(X,"JIVE").
CANNOT(X,"JIVE"):- MEMBER(X), CAN(X, "WALTZ").

CAN("DAVE",Y):- CANNOT("FRED",Y).
```

```
GOAL
MEMBER(X) , CANNOT(X,"JIVE").
OUTPUT:
```

```
X=DAVE
1 Solution
```

Analysis and Discussion: The converted predicate logics were implemented in the prolog and the query for if there are any member that cannot jive which was evaluated to be Fred. As per the computerless logical evaluation parents can do what ever their child can do, since freds dad cannot do waltz and all member can do either jive or walts indicating fred cannot do walts and can jive. Thus, Dave being able to do whatever fred cannot do, Dave can watz but not jive

CONCLUSION: This lab focused on first order predicate logic programming in Prolog, demonstrating its practical use through examples like the charlie being a horse or dave not being able to Jive and other various relation findings. Hence, We learned to express and solve problems by setting first order predicate logic, showcasing Prolog's versatility.

LAB No: 4

Backtracking

Backtracking in Prolog

In Prolog, there's a feature called backtracking, which is like a built-in trial-and-error mechanism. When we ask Prolog to find a solution to a problem, it tries different possibilities until it finds one that works. This is helpful because it saves us from having to explicitly handle backtracking ourselves. However, sometimes backtracking can make our programs less efficient. For example, if we only need one solution to a problem, but Prolog keeps trying to find more solutions, it wastes time and resources. Also, if we have rules (or clauses) that are mutually exclusive (meaning only one of them can be true at a time), then once Prolog proves one of them to be true, it's pointless to keep trying the others. To deal with these situations, Prolog has a feature called "cut" (written as "!"). We can use the cut to tell Prolog to stop backtracking and not consider any other possibilities. This can make our programs faster and more efficient.

However, using the cut can also make our code less clear and harder to understand. It can change the order in which Prolog considers rules, which might affect the results we get. So, while the cut can be useful for improving performance, we need to be careful when using it to avoid making our code harder to maintain.

There are two main uses of the cut:

1. When you know in advance that certain possibilities will never give rise to meaningful solutions, it's a waste of time and storage space to look for alternate solutions. If you use a cut in this situation, your resulting program will run quicker and use less memory. This is called a green cut.
2. When the logic of a program demands the cut, to prevent consideration of alternate subgoals. This is a red cut

Let's consider an examples that show how you can use the cut in your programs.


Example Rule : `r1 :- a, b, !, c.`

This is a way of telling Visual Prolog that you are satisfied with the first solution it finds to the subgoals `a` and `b`. Although Visual Prolog is able to find multiple solutions to the call to `c` through backtracking, it is not allowed to backtrack across the cut to find an alternate solution to the calls `a` or `b`. It is also not allowed to backtrack to another clause that defines the predicate `r1`.

Example Program

```
%SOURCE CODE
PREDICATES
buy_car(symbol,symbol)
nondeterm car(symbol,symbol,integer)
colors(symbol,symbol)
CLAUSES
buy_car(Model,Color):-
car(Model,Color,Price),
colors(Color,sexy),!,
Price < 25000.
car(maserati,green,25000).
car(corvette,black,24000).
car(corvette,red,26000).
car(porsche,red,24000).
colors(red,sexy).
colors(black,mean).
colors(green,preppy).
GOAL
buy_car(corvette, Y).
```

OUTPUT:

 [Inactive C
No Solution

In this example, the goal is to find a Corvette with a sexy color and a price that's ostensibly affordable. The cut in the `buy_car` rule means that, since there is only one Corvette with a sexy color in the known facts, if its price is too high there's no need to search for another car.

Given the goal

`buy_car(corvette, Y)`

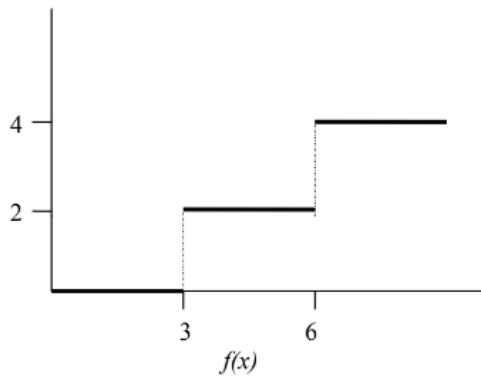
1. Visual Prolog calls `car`, the first subgoal to the `buy_car` predicate.
2. It makes a test on the first car, the Maserati, which fails.
3. It then tests the next car clauses and finds a match, binding the variable `Color` with the value `black`.
4. It proceeds to the next call and tests to see whether the car chosen has a sexy color. `Black` is not a sexy color in the program, so the test fails.
5. Visual Prolog backtracks to the call to `car` and once again looks for a Corvette to meet the criteria.
6. It finds a match and again tests the color. This time the color is sexy, and Visual Prolog proceeds to the next subgoal in the rule: the cut. The cut immediately succeeds and effectively "freezes into place" the variable bindings previously made in this clause.
7. Visual Prolog now proceeds to the next (and final) subgoal in the rule: the comparison `Price < 25000`.
8. This test fails, and Visual Prolog attempts to backtrack in order to find another car to test. Since the cut prevents backtracking, there is no other way to solve the final subgoal, and the goal terminates in failure

Consider the function as shown in the figure below. The relation between `X` and `Y` can be specified by the following three rules.

Rule 1: if `X < 3` then `Y = 0`

Rule 2: if `3 ≤ X < 6` then `Y = 2`

Rule 3: if `6 ≤ X` then `Y = 4`



A double step function

Assignment1: Program this function and modify the program using cut and observe the difference between the two modules. Comment on the difference

<p>PREDICATES f(integer,integer)</p> <p>CLAUSES f(X,0):- X<3. f(X,2):- 3<=X,X<6. f(X,4):- 6<X.</p> <p>GOAL f(2,X).</p> <p>OUTPUT: [inactive] X=0 1 Solution</p>	<p>PREDICATES f(integer,integer)</p> <p>CLAUSES f(X,0):- X<3. f(X,2):- 3<=X,! ,X<6. f(X,4):- 6<X.</p> <p>GOAL f(7,X).</p> <p>OUTPUT: No Solution</p>
---	--

Analysis and Discussion: The program for the aforementioned function worked as expected providing all the correct Y coordinates for the given X value. When the cut was applied to the program at the second clause inbetween two conditions then for the x input that was greater than 3 couldn't obtaine any closure. The cut feature of prolog limited the program to go any further after greater than 3 condition was checked which resulted in no solution.

Assignment2: Define the relation $\text{min}(X,Y,Z)$ where Z returns the smaller of the two given numbers X and Y . Do it with and without the use of cut and comment on the result.

<pre> PREDICATES min(integer,integer,integer) CLAUSES min(X,Y,Z):- X<Y,Z = X;Y<X, Z=Y. GOAL min(6,4,X). OUTPUT: X=4 1 Solution </pre>	<pre> PREDICATES min(integer,integer,integer) CLAUSES min(X,Y,Z):- X<Y,Z = X;!!Y<X, Z=Y. GOAL min(6,4,X). OUTPUT: X= 1 Solution </pre>
---	--

Analysis and Discussion: The program finds the minimum between X and Y then imprints the value that is smaller to Z and presents it as output. When the cut feature was used in the program as condition AND cut didnot display any changes but when the cut feature was used as OR function then the program returned empty output value.

Structure revisited:

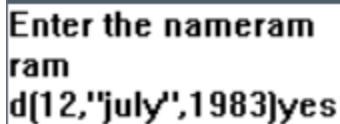
In prolog we can use structures to define data types of our requirement. For example if we want to use date as an structure we can define date as a structure in the domains section as follows $\text{date}=\text{d}(\text{integer},\text{symbol},\text{integer})$ We can then on use date as a data type to contain the date.

<pre> DOMAINS date=d(integer,symbol,integer) PREDICATES inquire display(symbol) date_of_birth(symbol,date) CLAUSES </pre>

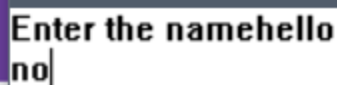
```

date_of_birth(ram,d(12,july,1983)).
date_of_birth(shyam,d(15,august,1976)).
date_of_birth(hari,d(26,may,1994)).
date_of_birth(sita,d(29,september,1991)).
display(X):-
date_of_birth(X,Y),
write(X),nl, write(Y).
inquire:- write("Enter the name"),
    readln(X), display(X).
GOAL
inquire.
OUTPUT:

```



Enter the name ram
ram
d(12,'july',1983)yes



Enter the name hello
no

Analysis and Discussion: Here the goal so proceeds as to ask a name from the user and to display the date of birth of the person with that name. With a little modification we can write goals which can find out persons with age below or above certain value, persons born in a month etc as in a relational database.

So the facts of the prolog can be thought of as a database. In fact we use structures to define certain relations and for all purposes of integrity this can be used similar to a table in a relational database. We call it the prolog's internal database. We can update this database during the execution of the program by using the following keywords.

assert(C) – this keyword can be used to assert a data in the facts base as **asserta(C)** and **assertz(C)** can be used to control the position of insertion, the two asserts at the beginning and the end respectively. **retract(C)** –deletes a clause that matches C.

Assignment 3:

```
DOMAINS
date=d(integer,symbol,integer)
works=w(symbol,integer)
FACTS
person(symbol,symbol,date,works).
PREDICATES
start
load_name
evalans(integer)
display
search
dispname(symbol)
delete
CLAUSES
person(shyam,sharma,d(12,august,1976),w(ntv,18000)).
person(ram,sharma,d(12,august,1976),w(ntv,18000)).
person(ram,singh,d(13,may,2001),w(utl,12000)).
start:-
write("*****MENU*****"),nl,
write("Press 1 to add new data"),nl,
write("Press 2 to show existing data"),nl,
write("Press 3 to search"),nl,
write("Press 4 to delete"),nl,
write("Press 0 to exit"),nl,
write("*****MENU*****"),nl,
readint(X),
```

```

evalans(X).
evalans(1):-load_name,start.
evalans(2):-display,evalans(2).
evalans(3):-search,start.
evalans(4):-delete,start.
evalans(0):-write("Thank You").
delete :-
    write("Enter the name of the person to delete: "), nl,
    readln(N),
    retract(person(N,_,d(,_,_),w(,_,_))),
    write("Person deleted successfully."), nl.
search:-
write("Enter the name \n"),nl,
readln(N),
person(N,_,d(,_,_),w(,_,_)),
dispname(N).
dispname(N):-
person(N,C,d(D,M,Y),w(O,S)),
write("Name:",N," ",C),nl,
write("Date of Birth:",D,"th"," ",M," ",Y),nl,
write("Organisation:",O),nl,
write("Salary:",S),nl,nl.
display:-
retract(person(N,X,d(D,M,Y),w(O,S))),
write("Name:",N," ",X),nl,
write("Date of Birth:",D,"th"," ",M," ",Y),nl,
write("Organisation:",O),nl,
write("Salary:",S),nl,nl.

```

```
load_name:-
write("Enter the name \n"),
readln(N),
write("Enter the surname \n"),
readln(S),
write("Date of Birth \n Day:"),
readint(D),nl,
write("Month:"),
readln(M),nl,
write("Year:"),
readint(Y),nl,
write("Enter the organisation:"),
readln(O),
write("Enter the salary:"),
readint(S1),nl,nl,
asserta(person(N,S,d(D,M,Y),w(O,S1))).
GOAL
Start.
OUTPUT:
```


*****MENU*****

Press 1 to add new data
Press 2 to show existing data
Press 3 to search
Press 4 to delete
Press 0 to exit

*****MENU*****

1

Enter the name

momo

Enter the surname

momo

Date of Birth

Day:1

Month:may

Year:2021

Enter the organisation:hello world

Enter the salary:12122121

*****MENU*****

Press 1 to add new data
Press 2 to show existing data
Press 3 to search
Press 4 to delete
Press 0 to exit

*****MENU*****

2

Name:momo momo

Date of Birth:1th may 2021

Organisation:hello world

Salary:12122121

Name:shyam sharma

Date of Birth:12th august 1976

Organisation:ntv

Salary:18000

Name:ram sharma

Date of Birth:12th august 1976

Organisation:ntv

Salary:18000

Name:ram singh

Date of Birth:13th may 2001

Organisation:utl

Salary:12000

no|

*****MENU*****

3

Enter the name

shyam

Name:shyam sharma

Date of Birth:12th august 1976

Organisation:ntv

Salary:18000

*****MENU*****

Press 1 to add new data

Press 2 to show existing data

Press 3 to search

Press 4 to delete

Press 0 to exit

*****MENU*****

4

Enter the name of the person to delete:

shyam

Person deleted successfully.

*****MENU*****

Press 1 to add new data

Press 2 to show existing data

Press 3 to search

Press 4 to delete

Press 0 to exit

*****MENU*****

2

Name:ram sharma

Date of Birth:12th august 1976

Organisation:ntv

Salary:18000

Name:ram singh

Date of Birth:13th may 2001

Organisation:utl

Salary:12000

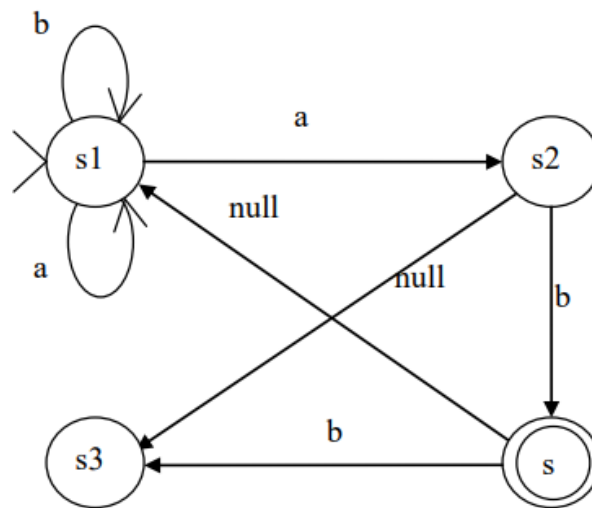
no

Analysis and Discussion: The provided Prolog program functions as a rudimentary database management system, allowing users to input, display, search for, and delete records of individuals' personal and employment information. The system's interface presents users with a clear menu to choose desired operations, facilitating ease of use. However, the program lacks robust error handling mechanisms, potentially leading to unexpected behavior if users provide

invalid inputs. Additionally, while the output is structured, it could benefit from formatting enhancements for improved readability. Despite these limitations, the program effectively demonstrates basic database management functionalities and serves as a foundation for further refinement and expansion to enhance its usability and reliability.

Simple application:

Let us now see how the features of prolog can be used to simulate a non-deterministic automata



which would have been a cumbersome task using other programming languages.

A nondeterministic finite automaton is an abstract machine that reads a string of symbols as input and decides whether to accept or to reject the input string. An automaton has a number of states and it is always in one of the states. The automata can change from one state to another upon encountering some input symbols. In a non deterministic automata the transitions can be non deterministic meaning that the transition may take place with a NULL character or the same character may result in different situations. A non deterministic automaton decides which of the possible moves to execute, and it chooses a move that leads to the acceptance of the string if such a move is available. Let us simulate the given automata.

Assignment 4. Check the automaton with various input strings and with various initial states. (The initial state need not necessarily be s1.) Observe the result and comment on how the simulation works.

Use the following goals

```

accepts(s1,[a,a,b]).
accepts(s1,[a,b,b]).
accepts(S,[b,a,b]).
accepts(s1,[X,Y,Z]).
accepts(s2,[b]).
accepts(s1,[_,_,_][a,b])

```

```

DOMAINS
symbol_list=symbol*
PREDICATES
transition(symbol, symbol, symbol)
null_transition(symbol, symbol)
final(symbol)
accepts(symbol, symbol_list)
CLAUSES
final(s).
transition(s1, a, s1).
transition(s1, a, s2).
transition(s1, b, s1).
transition(s2, b, s).
transition(s, b, s3).
null_transition(s2, s3).
null_transition(s, s1).
accepts(STATE, []):-
final(STATE).
accepts(STATE, [H|T]):-
transition(STATE,H, NEW_STATE),

```

```

accepts(NEW_STATE, T).
accepts(STATE, X):-
null_transition(STATE, NEW_STATE),
accepts(NEW_STATE, X).
GOAL
% accepts(s1,[a,a,b]).
% accepts(s1,[a,b,b]).
% accepts(S,[b,a,b]).
% accepts(s1,[X,Y,Z]).
% accepts(s2,[b]).
accepts(s1,[_,_,_,_][a,b])).
OUTPUT:

```

Analysis and Discussion: The provided Prolog program effectively models a finite automaton with states and transitions, allowing for the determination of whether a given symbol list is accepted by the automaton. Through the defined predicates for transitions, null transitions, and final states, the program accurately evaluates the acceptance of symbol lists according to the defined rules. By testing various input symbol lists against the defined goals, the program demonstrates its functionality and provides clear outputs, thereby achieving the intended purpose of modeling and simulating a finite automaton.

Conclusion:

In conclusion, these experiments provided insights into Prolog programming. The first program showed the impact of the cut operator on program flow, emphasizing its careful usage. The second program, a basic database system, demonstrated effective data management but lacked

robust error handling. Lastly, the third program, modeling a finite automaton, showcased Prolog's capability in simulating complex systems. These experiments highlight Prolog's versatility but also underscore the importance of thoughtful design and error handling in programming.

LAB No: 5

Artificial Neural Network

Introduction

An Artificial Neural Network (ANN) is a computer system that's inspired by how our brains work. Just like our brains have many interconnected neurons, an ANN has many interconnected processing elements called neurons. These neurons work together to solve different problems, like recognizing patterns or classifying data. One cool thing about ANNs is that they can learn from examples, just like we do. When we teach an ANN to do something, like recognize images of cats, it learns by looking at lots of examples of cats and adjusting its connections between neurons accordingly. So, in simple terms, an ANN is like a computer brain made up of lots of little parts (neurons) that work together to learn and solve problems. It learns by looking at examples and adjusting how its parts are connected, similar to how our brains learn by making connections between neurons.

Why Use Artificial Neural Networks?

Artificial Neural Networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained Artificial Neural Network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.

Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organization: An ANN can create its own organization or representation of the information it receives during learning time.

3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

A Simple Neuron

An artificial neuron is a computational unit designed to process information. It features multiple input channels and a single output. The neuron operates in two distinct modes: training and inference.

During the training phase, the neuron adjusts its internal parameters, such as weights and biases, based on input-output pairs provided by a learning algorithm. This adjustment process enables the neuron to learn associations between input patterns and desired output responses.

In the inference mode (using mode), the neuron applies the learned parameters to new input patterns. If the input pattern matches one of the learned patterns, the neuron produces the corresponding output. However, if the input pattern is novel, the neuron utilizes a predefined firing rule or activation function to determine its output response. This rule typically involves a mathematical operation on the weighted sum of inputs, often followed by a non-linear transformation to introduce complexity and expressive power to the neuron's behavior.

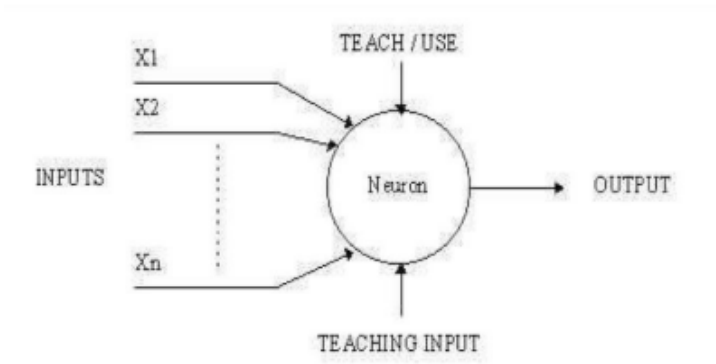


Figure 1.1: *A simple neuron*

An implementation of a simple neuron in python:

```
# implementing a simple neuron in python

import numpy as np

# a Neuron class
class Neuron:

    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs) # Iniitalizing with random weights
        self.bias = np.random.rand() # Initializing bias randomly

    def activate(self, inputs):
        print(self.weights, self.bias)
        summation = np.dot(inputs, self.weights) + self.bias
        print("summation: ", summation)
        return 1 if summation > 0 else 0

# A neuron with 3 inputs
neuron = Neuron(3)

inputs = np.array([0.5, 0.3, 0.1])

output = neuron.activate(inputs)

print("OUTPUT: ", output)
```


OUTPUT:

[0.10952803 0.6820824 0.90629053] 0.6151883456613486

summation: 0.9652061335604942

OUTPUT: 1

Network Layers:

The commonest type of Artificial Neural Network consists of three groups, or layers, of units: a layer of "input" units is connected to a layer of "hidden" units, which is connected to a layer of "output" units.

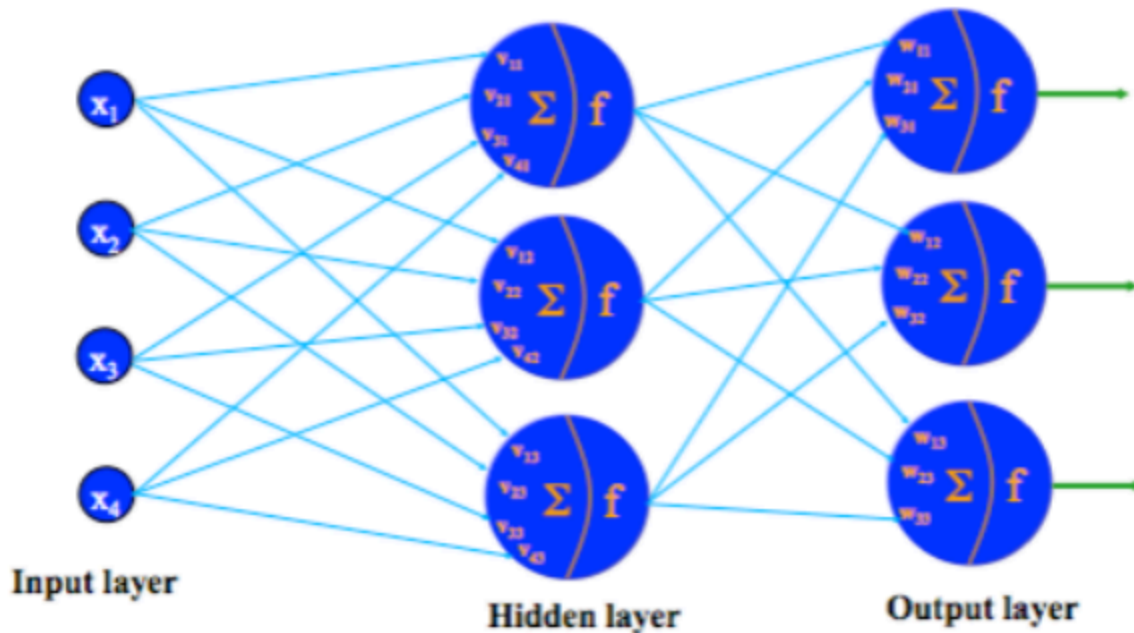


Figure 1.2: *A Simple Feed Forward Network*

In a neural network, the input units receive and represent the initial information fed into the system. The activity of each hidden unit is influenced by both the input units' activity and the weights assigned to the connections between them. Similarly, the output units' behavior is determined by the activity of the hidden units and the weights on the connections between the hidden and output units.

This type of network is intriguing because the hidden units have the ability to create their own interpretations or representations of the input data. The weights between the input and hidden units dictate when each hidden unit becomes active. By adjusting these weights, a hidden unit can effectively choose what aspect of the input it focuses on or represents.

Neural networks can be categorized into single-layer and multi-layer architectures. In a single-layer network, all units are connected to each other, offering considerable computational flexibility. Conversely, multi-layer networks feature hierarchical structures where units are organized into layers. In such networks, units within each layer are often numbered sequentially, reflecting their position within the network, rather than having a global numbering scheme.

Perceptrons:

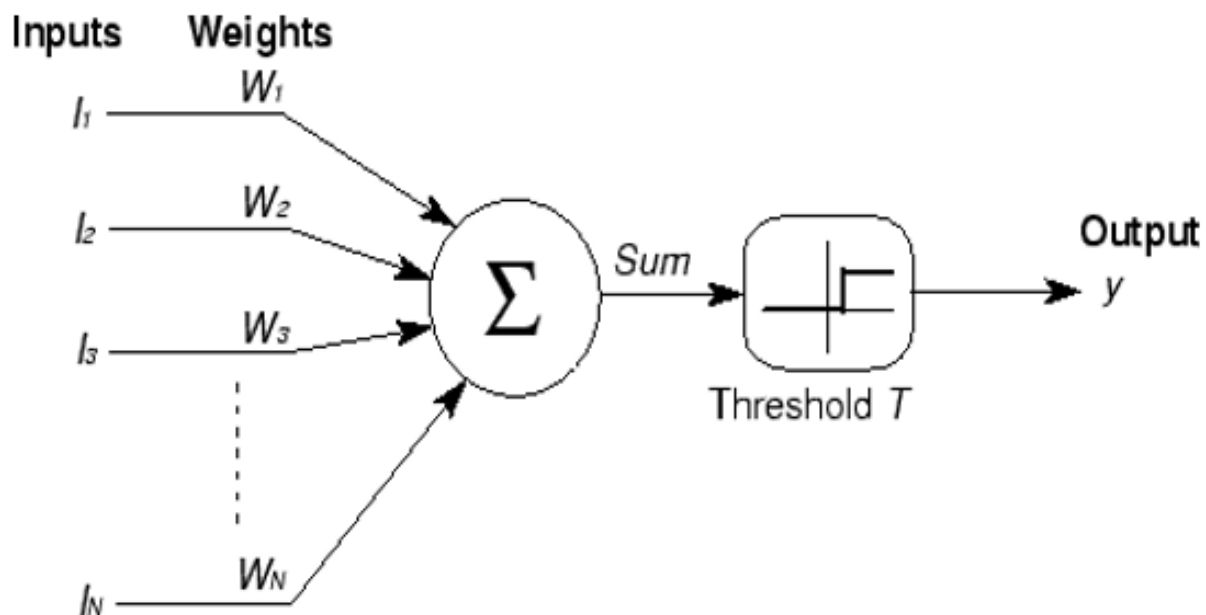


Fig: A simplifies model of a real neuron

In the 1960s, one of the most influential works in the field of neural networks was centered around a concept known as "perceptrons," a term coined by Frank Rosenblatt. The perceptron, depicted in Figure 1.3, is essentially a simplified model of a neuron, known as an MCP model, which includes weighted inputs along with some fixed pre-processing.

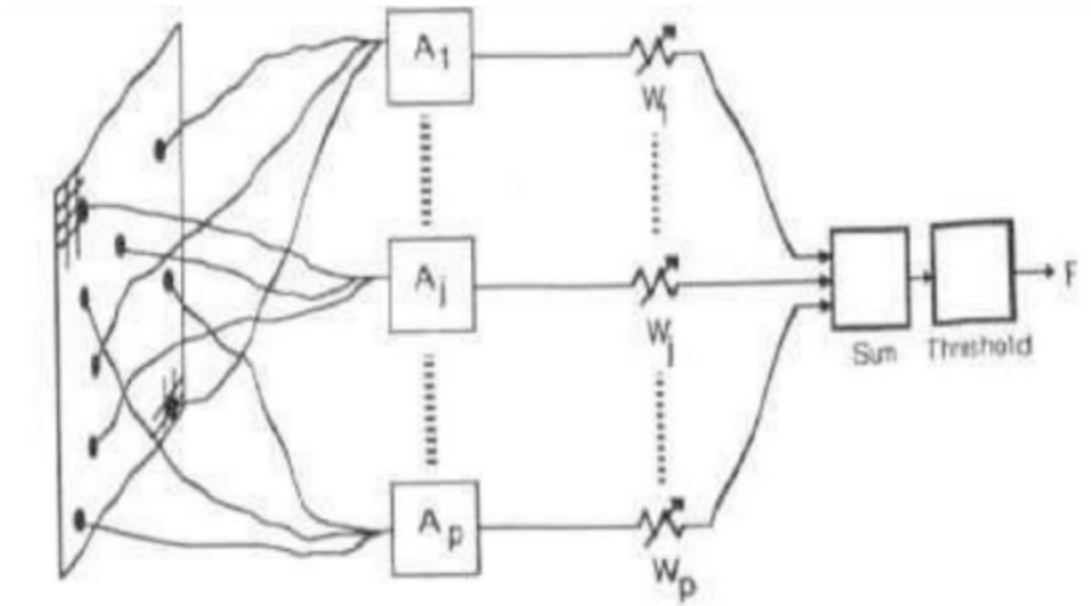


Figure 1.3: A Perceptron

Within the perceptron model, units labeled A_1 , A_2 , A_j , and A_p are referred to as association units. These units are responsible for extracting specific, localized features from input images. In essence, perceptrons aim to emulate the fundamental principles of the mammalian visual system. While initially utilized primarily for pattern recognition tasks, perceptrons possessed capabilities that extended beyond this domain. Despite their simple structure, they demonstrated potential for more complex tasks and contributed significantly to the development of neural network theory and applications.

Transfer Functions:

The behavior of an ANN depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- For linear (or ramp) the output activity is proportional to the total weighted output.
- For threshold units, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

- For sigmoid units, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations.

To make an Artificial Neural Network that performs some specific task, we must choose how the units are connected to one another and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

Algorithm for Adaline Learning :

Step 0: Initialize Weights and Learning Rate

- Initialize the weights w_1, w_2, \dots, w_n and bias term b to small random values.
- Select a learning rate α (alpha).

Step 1: Iterate Over Training Data

- For each input vector s with target output t :
 - Set the inputs to s .

Step 2: Compute Neuron Inputs

- Compute the net input y_{in} for the neuron:

$$y_{in} = b + \sum (x_i * w_i)$$

- Where x_i are the input features and w_i are the corresponding weights.

Step 3: Update Weights and Bias

- Use the Delta Rule to update the bias and weights:

$$w_i := w_i + \alpha * (t - y_{in}) * x_i$$

$$b := b + \alpha * (t - y_{in})$$

- Where α is the learning rate, t is the target output, and y_{in} is the computed net input.

Step 4: Check for Convergence

- Stop if the largest weight change across all the training samples is less than a specified tolerance.
- Otherwise, cycle through the training set again.

Backpropagation Algorithm:

Step 0: Initialize Weights

- Initialize the weights to small random values.

Step 1: Forward Pass

- Feed the training sample through the network and determine the final output.

Step 2: Compute Output Unit Errors

- Compute the error for each output unit:

$$\delta_k = (t_k - y_k) * f'(y_{ink})$$

- Where t_k is the target output, y_k is the actual output, and $f'(y_{ink})$ is the derivative of the activation function with respect to the net input of unit k .

Step 3: Backpropagate Errors to Hidden Units

- Propagate the delta terms (errors) back through the weights of the hidden units:

$$\delta_j = (\sum (\delta_k * w_{jk})) * f'(y_{inj})$$

- Where δ_j is the error for hidden unit j, w_{jk} is the weight between hidden unit j and output unit k, and $f'(y_{inj})$ is the derivative of the activation function with respect to the net input of hidden unit j.

Step 4: Update Weights for Output Units

- Calculate the weight correction term for each output unit:

$$\Delta w_{jk} = \alpha * \delta_k * z_j$$

- Where α is the learning rate, δ_k is the error for output unit k, and z_j is the output of hidden unit j.

Step 5: Update Weights for Hidden Units

- Calculate the weight correction term for each hidden unit:

$$\Delta w_{ij} = \alpha * \delta_j * x_i$$

- Where δ_j is the error for hidden unit j, x_i is the input to hidden unit j, and α is the learning rate.

Step 6: Update Weights

- Update the weights using the calculated weight correction terms:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \Delta w_{ij}$$

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$

Step 7: Test for Stopping

- Test for stopping criteria such as reaching a maximum number of epochs or small changes in weights.

Implementation of Logic Functions

Assignment (1): Design a McCulloch-Pitts neural network which behaves as AND function using Adaline learning. Consider unipolar case. Perform analysis by varying NN parameters.

Assignment (2): Similarly develop a McCulloch-Pitts neural net for OR, NAND and NOR gate and draw neural nets.

```
# A McCulloch-Pitts neural network which behaves as AND function using Adaline Learning. Considering a Unipolar case
```

```
import numpy as np
```

```
# Adaline Learning
class Adaline:
```

```
    def __init__(self, num_inputs, learning_rate = 0.1 ):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand()
        self.learning_rate = learning_rate
```

```
    def activate(self, inputs):
        net_inputs = np.dot(inputs, self.weights) + self.bias
        return 1 if net_inputs > 0 else 0
```

```
    def train(self, inputs, target):
        output = self.activate(inputs)
        error = target - output
        self.weights += self.learning_rate * error * inputs
        self.bias += self.learning_rate * error
```

```
    def getUpdates(self):
```

```
        print("Weights : ", self.weights, "\n bias : ", self.bias, "\n learning rate: ", self.learning_rate, "\n ")
```

```
def andData():
    # Inputs and Target Outputs of a AND function
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([0, 0, 0, 1])

    return inputs, targets
```

```
def orData():
    # Inputs and Target Outputs of a AND function
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```

    targets = np.array([0, 1, 1, 1])

    return inputs, targets

def nandData():
    # Inputs and Target Outputs of a AND function
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([1, 1, 1, 0])

    return inputs, targets

def norData():
    # Inputs and Target Outputs of a AND function
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([1, 0, 0, 0])

    return inputs, targets

function_map = {
    '1': 'andData',
    '2': 'orData',
    '3': 'nandData',
    '4': 'norData'
}

def main():
    ch = input("1. AND \n 2. OR \n 3. NAND \n 4. NOR \n:")
    function = globals().get(function_map.get(ch))
    if function:
        inputs, targets = function()
    else:
        print("Function not found")
        exit()

    # A varing range of Neural Network parameters
    learning_rates = [0.01, 0.1, 0.5]
    epochs_list = [100, 500, 1000]

    for learning_rate in learning_rates:

        for epochs in epochs_list:

            Anding = Adaline(num_inputs=2, learning_rate=learning_rate)

            print("\n \n Before Training: ")
            Anding.getUpdates()

            # Training
            for epoch in range(epochs):
                for input_data, target in zip(inputs, targets):
                    Anding.train(input_data, target)

```



```

        print(f"After Training with {epochs} epoch cycle ")
        Anding.getUpdates()

    # Testing
    for input_data, target in zip(inputs, targets):
        output = Anding.activate(input_data)
        print("Input:", input_data, "Target:", target, "Output:", output)

if __name__ == "__main__":
    main()

```

OUTPUT:

PS C:\AI\NN> python -u "c:\AI\NN\Adaline.py"

1. AND
2. OR
3. NAND
4. NOR

:1

Before Training:
Weights : [0.04986217 0.58262814]
bias : 0.19124795911343961
learning rate: 0.01

After Training with 100 epoch cycle
Weights : [0.01986217 0.25262814]
bias : -0.2687520408865605
learning rate: 0.01

Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1

Before Training:
Weights : [0.79065075 0.86727757]
bias : 0.7071295283791379
learning rate: 0.01

After Training with 500 epoch cycle
Weights : [0.37065075 0.40727757]
bias : -0.4128704716208628
learning rate: 0.01

Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1

```
Before Training:
Weights : [0.89749906 0.28552379]
bias : 0.3631094522263879
learning rate: 0.01

After Training with 1000 epoch cycle
Weights : [0.41749906 0.10552379]
bias : -0.4268905477736125
learning rate: 0.01

Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1
```

```
Before Training:
Weights : [0.15314443 0.60683926]
bias : 0.004956079703986926
learning rate: 0.1

After Training with 100 epoch cycle
Weights : [0.15314443 0.30683926]
bias : -0.3950439202960131
learning rate: 0.1
```

```
Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1
```

```
Before Training:
Weights : [0.34234651 0.41398174]
bias : 0.9527450231962218
learning rate: 0.1

After Training with 500 epoch cycle
Weights : [0.14234651 0.01398174]
bias : -0.14725497680377808
learning rate: 0.1
```

```
Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1
```

```
Before Training:
Weights : [0.9728707 0.14462691]
bias : 0.598046954547507
learning rate: 0.1
```

After Training with 1000 epoch cycle

Weights : [0.3728707 0.04462691]

bias : -0.401953045452493

learning rate: 0.1

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 0 Output: 0

Input: [1 0] Target: 0 Output: 0

Input: [1 1] Target: 1 Output: 1

Before Training:

Weights : [0.11572757 0.03060736]

bias : 0.5405108804352526

learning rate: 0.5

After Training with 100 epoch cycle

Weights : [1.11572757 0.53060736]

bias : -1.4594891195647475

learning rate: 0.5

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 0 Output: 0

Input: [1 0] Target: 0 Output: 0

Input: [1 1] Target: 1 Output: 1

Before Training:

Weights : [0.19203589 0.75189121]

bias : 0.40642881198399383

learning rate: 0.5

After Training with 500 epoch cycle

Weights : [1.19203589 0.75189121]

bias : -1.5935711880160062

learning rate: 0.5

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 0 Output: 0

Input: [1 0] Target: 0 Output: 0

Input: [1 1] Target: 1 Output: 1

Before Training:

Weights : [0.00865052 0.96637072]

bias : 0.20841435403622566

learning rate: 0.5

After Training with 1000 epoch cycle

Weights : [0.50865052 0.46637072]

bias : -0.7915856459637743

learning rate: 0.5

```
Input: [0 0] Target: 0 Output: 0
Input: [0 1] Target: 0 Output: 0
Input: [1 0] Target: 0 Output: 0
Input: [1 1] Target: 1 Output: 1
```

The neuron provided correct output when acting as all aforementioned logic functions

Analysis and Discussion: The McCulloch-Pitts neural network implemented using Adaline learning, configured to behave as an AND function in a unipolar case, was tested with varying neural network parameters such as learning rates and epochs. The network demonstrated the capability to learn and accurately predict the outputs for the AND function. Analysis of the training process revealed that with increased epochs and appropriate learning rates, the network's weights and biases converged to values that enabled accurate predictions for the given input patterns. Furthermore, the network's performance was evaluated using other logical functions such as OR, NAND, and NOR, demonstrating its flexibility and effectiveness in learning different types of patterns. The results indicate the versatility and reliability of the Adaline learning algorithm in training McCulloch-Pitts neural networks to perform various logical functions.

Assignment (3): Perform test for bipolar model as well.

For the similar training model above input parameters and activation function was altered as follows

```
...
def activate(self, inputs):
    net_inputs = np.dot(inputs, self.weights) + self.bias

    return 1 if net_inputs > 0 else -1

...
def main():
    ch = input("1. AND \n 2. OR \n 3. NAND \n 4. NOR \n:")

    inputs = np.array([[ -1, -1], [ -1, 1], [ 1, -1], [ 1, 1]])

    if(ch == '1'):
        targets = np.array([ -1, -1, -1, 1])
    elif(ch == '2'):
        targets = np.array([ -1, 1, 1, 1])
    elif(ch == '3'):
```

```

        targets = np.array([1, 1, 1, -1])
    elif(ch == '4'):
        targets = np.array([1, -1, -1, -1])
    else:
        print("Error!")
        exit()

    learning_rates = [0.01, 0.1, 0.5]
    epochs_list = [100, 500, 1000]
    ...

```

Analysis and Discussion: The McCulloch-Pitts neural network was adapted for a bipolar case and utilized Adaline learning to emulate logical functions, such as AND, OR, NAND, and NOR. Various neural network parameters, including learning rates and epochs, were explored to understand their impact on the training process and subsequent performance. Through systematic testing, the network successfully learned the specified logical functions, accurately predicting outputs for given input patterns. Analysis of the training process revealed that the network converged to appropriate weights and biases, facilitating accurate predictions. Furthermore, the network's versatility and effectiveness in handling various pattern recognition tasks were showcased by its ability to learn different logical functions. Overall, the results demonstrate the robustness and reliability of the Adaline learning algorithm in training McCulloch-Pitts neural networks for bipolar cases.

Assignment 4: Implement McCulloch-Pitts neural network model for XOR and give all the formula you used in the implementation. Draw the MLPs used for the implementation of above functions.

```

# Implmention of XOR logic function using 2 layers

import numpy as np

class Adaline:

    def __init__(self, num_inputs, learning_rate = 0.1 ):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand()
        self.learning_rate = learning_rate

    def activate(self, inputs):
        net_inputs = np.dot(inputs, self.weights) + self.bias
        return 1 if net_inputs > 0 else 0

```

```

def train(self, inputs, target):
    output = self.activate(inputs)
    error = target - output
    self.weights += self.learning_rate * error * inputs
    self.bias += self.learning_rate * error

def getUpdates(self):

    print("Weights : ", self.weights, "\n bias : ", self.bias,
"\n learning rate: ", self.learning_rate, "\n ")

def train_perceptron(perceptron, inputs, targets):

    print("Before Training : ",)
    perceptron.getUpdates()

    # Training the perceptron model
    epochs = 100
    for each in range(epochs):
        for input_data, target in zip(inputs, targets):
            perceptron.train(input_data, target)

    print("After Training : ")
    perceptron.getUpdates()

    output_list = []
    # Testing the trained model
    for input_data, target in zip(inputs, targets):
        output = perceptron.activate(input_data)
        output_list.append(output)
        print("Input:", input_data, "Target: ", target, "Output: ",
output)

    return output_list

def main():

    # creating a perceptron model with 2 inputs units
    perceptron = Adaline(num_inputs= 2, learning_rate= 0.1)

```

```

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

targets = [0, 0, 1, 0]
z1 = train_perceptron(perceptron, inputs, targets)

targets = [0, 1, 0, 0]
z2 = train_perceptron(perceptron, inputs, targets)

# Train for OR
targets = [0, 1, 1, 1]
temp = train_perceptron(perceptron, inputs, targets)

# now that the weights and biases are updated for OR function,
sending the intermediate inputs for ORring
intermediate = np.array( [[z1,z2] for z1, z2 in zip(z1,z2)])
targets = [0, 1, 1, 0]

print("XOR OUTPUT:")
for input_data, target, actual_input in zip(intermediate,
targets, inputs):
    output = perceptron.activate(input_data)
    print("Input:", actual_input, "Intermediate: ", input_data,
"Target: ", target, "Output: ", output)

```

```

if __name__ == "__main__":
    main()

```

OUTPUT:

Before Training :

Weights : [0.79983341 0.47475105]

bias : 0.9798415870485035

learning rate: 0.1

After Training :

Weights : [0.29983341 -0.32524895]

bias : -0.22015841295149632

learning rate: 0.1

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 0 Output: 0

Input: [1 0] Target: 1 Output: 1

Input: [1 1] Target: 0 Output: 0

Before Training :

Weights : [0.29983341 -0.32524895]

bias : -0.22015841295149632

learning rate: 0.1

After Training :

Weights : [-0.20016659 0.17475105]

bias : -0.020158412951496313

learning rate: 0.1

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 1 Output: 1

Input: [1 0] Target: 0 Output: 0

Input: [1 1] Target: 0 Output: 0

Before Training :

Weights : [-0.20016659 0.17475105]

bias : -0.020158412951496313

learning rate: 0.1

After Training :

Weights : [0.09983341 0.17475105]

bias : -0.020158412951496313

learning rate: 0.1

Input: [0 0] Target: 0 Output: 0

Input: [0 1] Target: 1 Output: 1

Input: [1 0] Target: 1 Output: 1

Input: [1 1] Target: 1 Output: 1

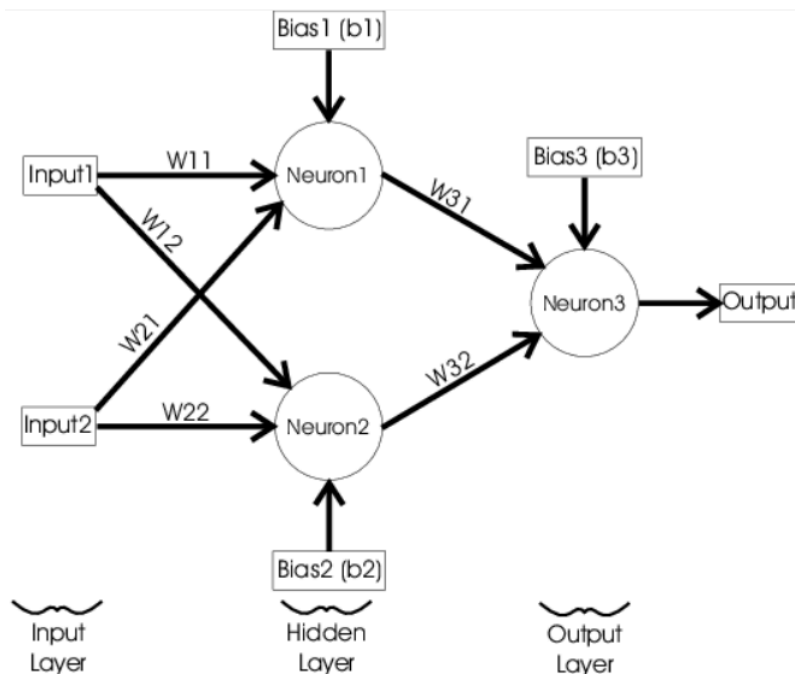
XOR OUTPUT:

Input: [0 0] Intermediate: [0 0] Target: 0 Output: 0

Input: [0 1] Intermediate: [0 1] Target: 1 Output: 1

Input: [1 0] Intermediate: [1 0] Target: 1 Output: 1

Input: [1 1] Intermediate: [0 0] Target: 0 Output: 0



Analysis and Discussion:

The provided code trains a perceptron initially for the logical AND operations $z1 = x1 \bar{x}2$ and $z2 = \bar{x}1 x2$, followed by training for the logical OR operation. It then utilizes the intermediate data of $z1$ and $z2$ to compute the XOR output. By training the perceptron for basic logical operations and utilizing intermediate results, the code showcases the versatility of perceptrons in handling different tasks. However, perceptrons have limitations when dealing with complex data patterns or nonlinear relationships, necessitating more advanced neural network architectures for such tasks.

Assignment (5): Implement MLP model for XOR by using backpropagation algorithm

```
import numpy as np
#np.random.seed(0)

def sigmoid (x):
    return 1/(1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])

epochs = 10000
lr = 0.1
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

#Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
hidden_bias = np.random.uniform(size=(1,hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))

print("Initial hidden weights: ",end="")
print(*hidden_weights)
print("Initial hidden biases: ",end="")
print(*hidden_bias)
print("Initial output weights: ",end="")
print(*output_weights)
print("Initial output biases: ",end="")
print(*output_bias)

#Training algorithm
for _ in range(epochs):
```

```

#Forward Propagation
hidden_layer_activation = np.dot(inputs,hidden_weights)
hidden_layer_activation += hidden_bias
hidden_layer_output = sigmoid(hidden_layer_activation)

output_layer_activation = np.dot(hidden_layer_output,output_weights)
output_layer_activation += output_bias
predicted_output = sigmoid(output_layer_activation)

#Backpropagation
error = expected_output - predicted_output
d_predicted_output = error * sigmoid_derivative(predicted_output)

error_hidden_layer = d_predicted_output.dot(output_weights.T)
d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

#Updating Weights and Biases
output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
hidden_weights += inputs.T.dot(d_hidden_layer) * lr
hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("Final hidden weights: ",end=")
print(*hidden_weights)
print("Final hidden bias: ",end=")
print(*hidden_bias)
print("Final output weights: ",end=")
print(*output_weights)
print("Final output bias: ",end=")
print(*output_bias)

print("\nOutput from neural network after 10,000 epochs: ",end=")
print(*predicted_output)

OUTPUT:
Initial hidden weights: [0.72089572 0.71470676] [0.83976815 0.12917138]
Initial hidden biases: [0.06008692 0.84752141]
Initial output weights: [0.97351059] [0.63951291]
Initial output biases: [0.62845166]Final hidden weights: [6.26598166 3.59835464] [5.87296901 3.5380068 ]
Final hidden bias: [-2.49594017 -5.44838159]
Final output weights: [7.47055059] [-8.18860971]
Final output bias: [-3.32078904]

Output from neural network after 10,000 epochs: [0.05803393] [0.94517017] [0.94627922] [0.05944358]

```

Analysis and Discussion:

The backpropagation algorithm was employed to train a multi-layer perceptron (MLP) for solving XOR. In Python, an MLP was initialized with random weights and biases, followed by iterative forward and backward passes to compute gradients and update parameters. Error

convergence was monitored, and the trained model was tested on XOR data to evaluate the effectiveness of backpropagation in learning the XOR function. Through this process, the algorithm demonstrated its ability to adjust network parameters to minimize error and make accurate predictions, showcasing its significance in training neural networks for challenging tasks. The output was obtained from 10,000 epochs and learning rate = 0.1, Hence, the neural network has converged to the expected output:

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

CONCLUSION:

In conclusion, throughout the lab assignments, various neural network models were successfully explored and implemented to perform logical operations such as AND, OR, NAND, and NOR gates using the McCulloch-Pitts neural network framework. Adaline learning was employed in the unipolar case, allowing for an analysis of the network's behavior by varying parameters and providing insights into its performance. Additionally, exploration extended to include bipolar models, ensuring a comprehensive understanding of different scenarios. Furthermore, a McCulloch-Pitts neural network model specifically for the XOR function was implemented, with the formulas utilized in the implementation detailed and the network structures visually represented. Lastly, an MLP model for XOR using the backpropagation algorithm was implemented, showcasing the versatility and effectiveness of this approach in solving complex problems. Through these assignments, valuable insights into neural network architectures and their applications in solving logical operations were gained, laying a solid foundation for further exploration and study in this field.

