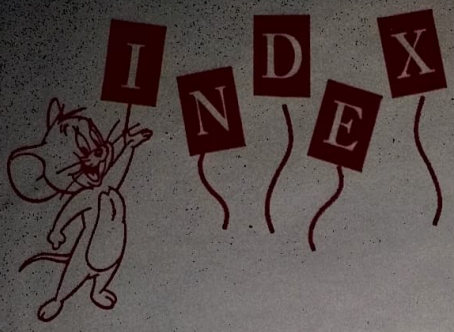


Veera's Student's Choice

Name : Blessy Abidha .B.S

STD : ..... SEC : ..... ROLL NO : .....

S.No.	Date	Title	Page No.	Teacher's Sign /
1.	9/8/24	Basic python programs using google colab.	9	
2.	16/8/24	Cardio vascular Risk Assesment	9	
3.		N-queens		
4.	13/9/24	DFS	10	
5.	20/9/24	DFS-Water Jug Problem	10	
6.	4/10/24	Minimax Algorithm	10	
7.	20/9/24	A* Algorithm	10	
10.	4/10/24	Implementing artificial neural networks for an application using python Regression.	10	
8.	4/10/24	Introduction to prolog	10	
9.	11/10/24	Prolog family Tree	10	
11.	18/10	Implementation of Decision Tree	10	
12.	25/10	K-Means clustering.	10	

completed

6/9/24 Ex: 8

N-Queens

```
def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(board, col):
    if col >= len(board):
        return True
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_n_queens(board, col+1):
                return True
            board[i][col] = 0
    return False

def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))

def solve():
    n = 8
    board = [[0 for i in range(n)] for i in range(n)]
    if not solve_n_queens(board, 0):
        print("solution does not exist")
        return False
```

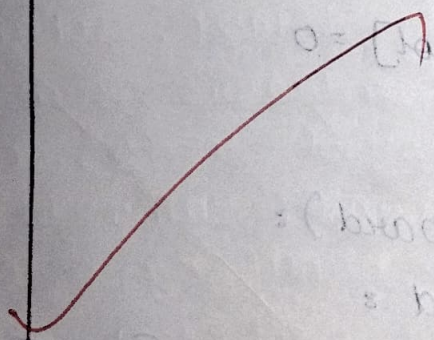
print boardlboard)
return True

solve()

o/p:

1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

True.



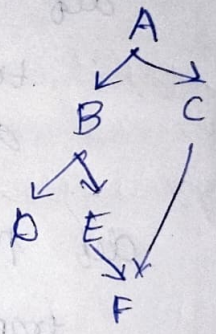
Result:

Thus the above 8 Queen problem is executed successfully.

13/9/24 Ex 4:

DFS

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbour in graph[start]:
        if neighbour not in visited:
            dfs(graph, neighbour, visited)

graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}
dfs(graph, 'A')
```



Output:

A B D E F C.

Result:

Thus DFS program is executed successfully.

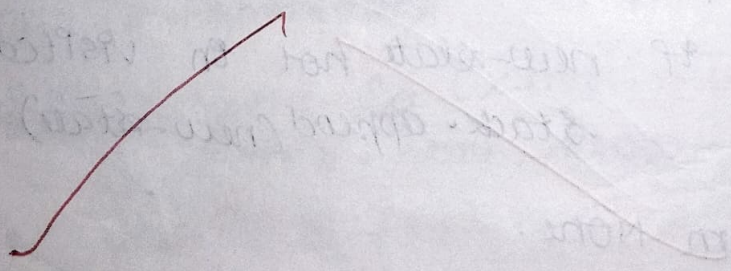
EX 5 :
Date : 20/9/24 Water Jug Problem Using DFS

```
# Helper functions for different actions
def fill_4_gallon(x, y, x_max, y_max):
    return (x_max, y)

def fill_3_gallon(x, y, x_max, y_max):
    return (x, y_max)

def empty_4_gallon(x, y, x_max, y_max):
    return (0, y)

def empty_3_gallon(x, y, x_max, y_max):
    return (x, 0)

def pour_4_to_3(x, y, x_max, y_max):
    transfer = min(x, y_max - y)
    return (x - transfer, y + transfer)

def pour_3_to_4(x, y, x_max, y_max):
    transfer = min(y, x_max - x)
    return (x + transfer, y - transfer)

# DFS Implementation
def dfs_water_jug(x_max, y_max, goal,
                  visited = None, start = (0,0)):
    if visited is None:
        visited = set()
    stack = [start]
    while stack:
        state = stack.pop()
        x, y = state
```

```
    if state in visited:
        continue
    visited.add(state)
    print(f"Visiting state: {state}")
    # check if goal state is reached
    if x == goal_x:
        print(f"Goal reached: {state}")
        return state
    # Generate all possible next states based
    on current state
    next_states = [
        fill_4_gallon(x, y, x_max, y_max),
        fill_3_gallon(x, y, x_max, y_max),
        empty_4_gallon(x, y, x_max, y_max),
        empty_3_gallon(x, y, x_max, y_max),
        pour_4_to_3(x, y, x_max, y_max),
        pour_3_to_4(x, y, x_max, y_max)
    ]
    # Add all unvisited next states to the stack.
    for new_state in next_states:
        if new_state not in visited:
            stack.append(new_state)
return None.

# Parameters
x_max = 4  # Maximum capacity of 4-gallon jug
y_max = 3  # Maximum capacity of 3-gallon jug
goal_x = 2  # We want 2 gallons in 4 gallons jug.

dfs_water_jug(x_max, y_max, goal_x)
```

Output:

Visiting state : (0,0)
Visiting state : (0,3)
Visiting state : (3,0)
Visiting state : (3,3)
Visiting state : (4,2)
Visiting state : (4,0)
Visiting state : (1,3)
Visiting state : (1,0)
Visiting state : (0,1)
Visiting state : (4,1)
Visiting state : (2,3)
Goal Reached : (2,3)
(2,3)



Result:

Thus the water jug problem has been solved successfully using DFS.

Date: 20/9

A* Search Algorithm

Aim:

To Implement A* search algorithm using python

Code:

```
def astar (start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = { }
    parents = { }
    g[start_node] = 0
    parents [start_node] = start_node
    while len(open_set) > 0:
        n = None
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n]
                                + heuristic[n]:
            n = v
    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
    for (m, weight) in get_neighbours(n):
        if m not in open_set and m not in
                                closed_set:
            open_set.add(m)
            parents[m] = n
            g[m] = g[n] + weight
        else
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
```

```
            if m in closed_set:
                closed_set.remove(m)
                open_set.add(m)

        if n == None:
            print("Path does not exist!")
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
        path.append(start_node)
        path.reverse()
        print("Path : {}", format(path))
        return
        open_set.remove(n)
        closed_set.add(n)
        print("Path does not exist")
        return None
    def get_neighbours(v):
        if v in graph_nodes:
            return graph_nodes[v]
        else:
            return None
    def heuristic(n):
        H_dist = {
            'A' : 11,
            'B' : 6,
            'C' : 99,
            'D' : 1,
            'E' : 7,
            'G' : 0
        }
```

```
        return H-dist [n]
    Graph_nodes = {
        'A' : [('B', 2), ('E', 3)],
        'B' : [('C', 1), ('G', 9)]
        'C' : None,
        'E' : [('D', 6)],
        'D' : [('G', 1)],
    }
astar('A', 'G')
```

output :

path : ['A', 'E', 'D', 'G']

Result :

Thus the A* search algorithm using python is executed successfully.

Ex:11 Implementation of Decision Tree
Date: 8/10/24

Aim:

To implement a decision tree classification technique for gender classification using python.

Code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score,
    classification report, confusion matrix.
import matplotlib.pyplot as plt
from sklearn import tree.

# 1 Generating synthetic dataset
data = {
'Height': [150, 160, 170, 180, 155, 165, 175, 185, 140, 145],
'Weight': [50, 60, 70, 80, 55, 65, 75, 85, 45, 50],
'Age': [25, 30, 35, 40, 28, 32, 37, 42, 24, 29],
'Gender': ['Female', 'Female', 'Male', 'Male', 'Female',
           'Female', 'Male', 'Male', 'Female', 'Female']
    }
df = pd.DataFrame(data)
# 2 Encode the target variable
df['Gender'] = df['Gender'].map({'Female':0, 'Male':1})
# 3 Features and target variables
X = df[['Height', 'Weight', 'Age']]
y = df['Gender']
# 4 stratified split
X_train, X_test, y_train, y_test = train_test_split
    (X, y, test_size=0.3, random_state=42,
                                stratify=y)
# 5. Create a decision tree classifier
clf = DecisionTreeClassifier()
```

#6. Train the model
clf.fit(x_train, y_train)

#7. Make predictions
y_pred = clf.predict(x_test)

#8. Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred, zero_division=0)

#9. Display the output
print(f'Accuracy : {accuracy:.2f}')
print('Confusion Matrix :\n', conf_matrix)
print('Classification Report :\n', class_report)

#10. Visualizing the Decision tree
plt.figure(figure=[12, 8])
tree.plot_tree(clf, feature_names=x.columns, class_names=['Female', 'Male'], filled=True)
plt.title('Decision Tree for Gender Classification')
plt.show()

Output :
Enter height (in cm) for prediction : 169
Enter weight (in kg) for prediction : 61
Predicted gender for height 169.0 cm and weight 61.0 kg : Female.

Result :

Thus the decision Tree classification technique for gender classification using python was implemented sucessfully.

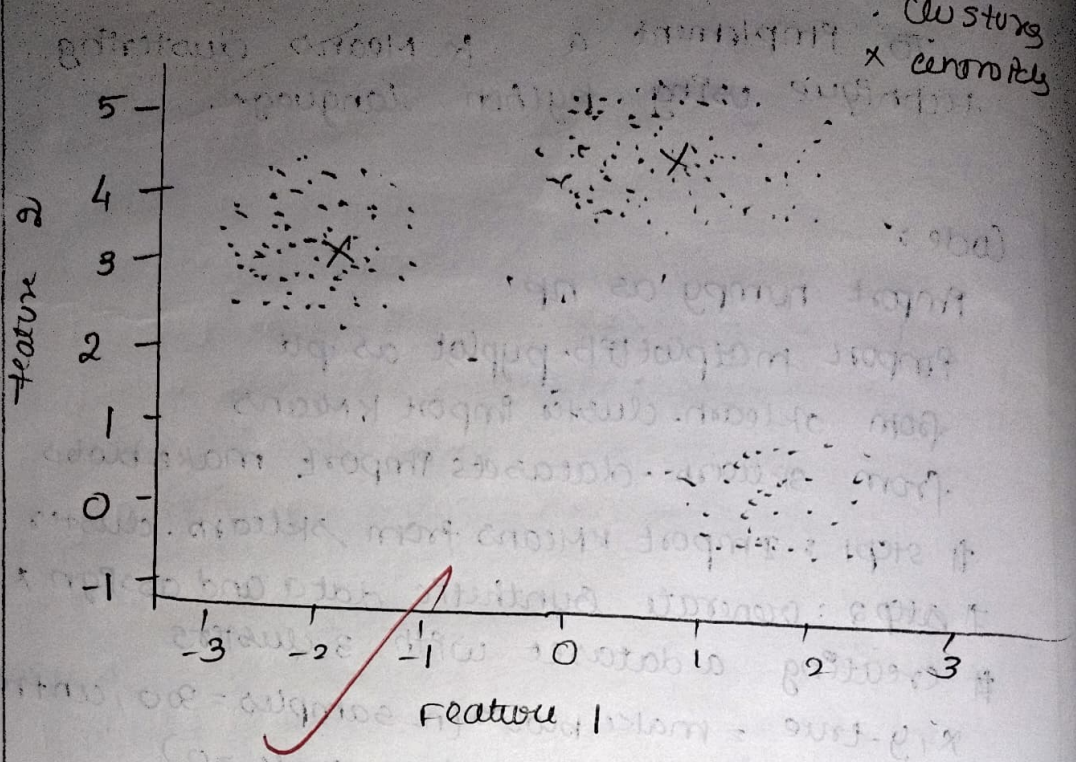
EX : 12
Date : 25/10 K-Means clustering
Aim :

To implement a K-Means clustering technique using python language.

Code :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
# step1 : Import KMeans from sklearn.cluster
# step2 : Generate synthetic data and assign X and y
# creating a dataset with 3 clusters
X, y_true = make_blobs(n_samples=300, centres=3,
  cluster_std=0.60, random_state=0)
# step 3 : Call the function KMeans()
# Fit K-Means with the chosen number of clusters(K=3)
k=3
kmeans = KMeans(n_clusters=k, random_state=0)
y_kmeans = kmeans.fit_predict(X)
# step 4 : Perform scatter operation and display
the output
plt.figure(figsize=(8,6))
plt.scatter(X[:,0], X[:,1], c=y_kmeans, s=30,
  cmap='viridis', label='clusters')
centers = kmeans.cluster_centers
plt.scatter(centers[:,0], centers[:,1], c='red',
  s=200, alpha=0.75, marker='X', label='centroids')
plt.title('K-means clustering Results')
plt.xlabel('Feature 1')
plt.ylabel('Feature 0')
plt.legend()
plt.show()
```

Output :



Result :

Thus the K-Means clustering Algorithm is implemented successfully.

EX :10
Date : 4/10/ Artificial Neural Networks

Aim :

To implementing artificial neural networks for an application in Regression using python

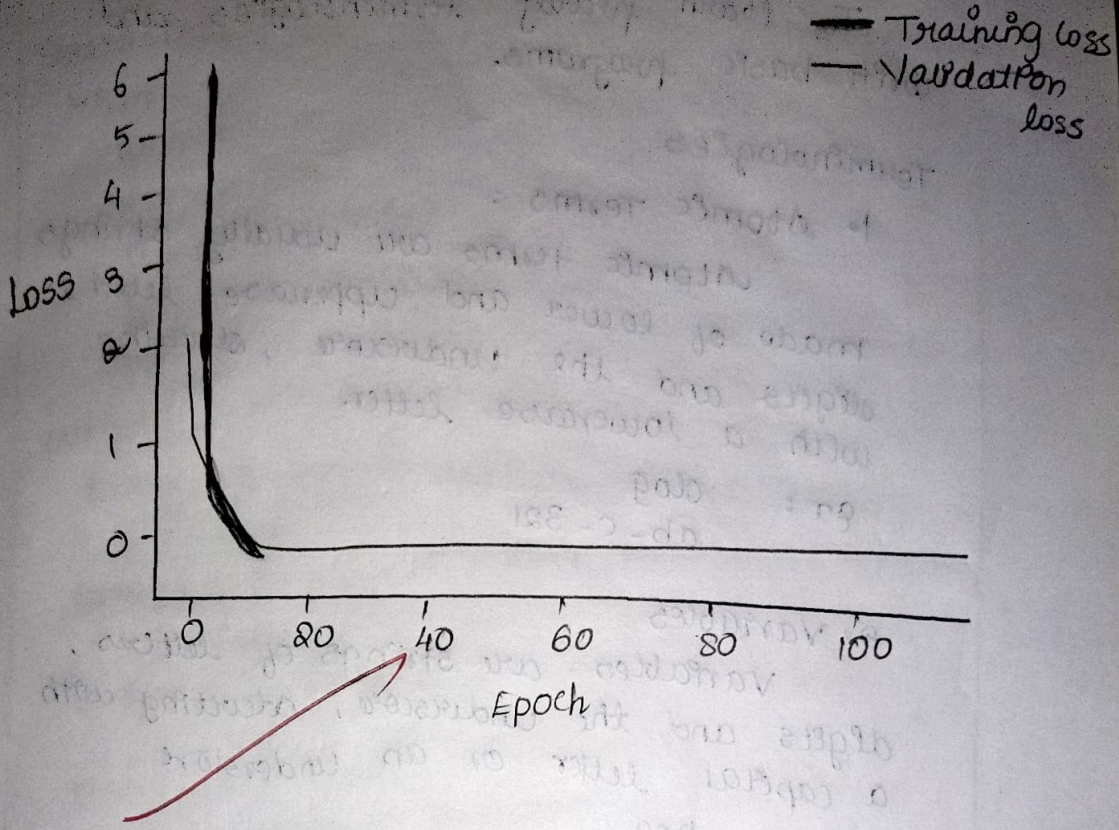
Code :

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.optimizers import Adam
import matplotlib.pyplot as plt.
# Generating synthetic dataset for demonstration
# Replace this with your own data set.
np.random.seed(42)
X = np.random.rand(1000, 3) # 1000 samples, 3 features
y = 3 * X[:,0] + 2*X[:,1]**2 + 1.5 * np.sin(X[:,2] *
    np.pi) + np.random.normal(0, 0.1, 1000)
# non linear relationship
# split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split
    (X, y, test_size = 0.2, random_state = 48)
# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Building the ANN Model.
model = Sequential()
# Input layer and first hidden layer with
10 neurons and ReLU activation
model.add(Dense(10, input_dim = X_train.shape[1],
    activation = 'relu')).
```

```
# output layer with linear activation for reg
model.add(Dense(1, activation = 'linear'))

# compiling the model
model.compile(optimizer = Adam(learning_rate
    loss = 'mean_squared_error')

# training the model
history = model.fit(X_train, y_train, epochs=10
    batch_size=32, validation_split=0.2, verbose=

# Making predictions
y_pred = model.predict(X_test)

# Evaluating the model
mse = np.mean((y_test - y_pred.flatten())**2)
print(f'Mean Squared Error: {mse:.4f}')

# plotting training history
plt.figure(figsize=(12,6))
plt.plot(history.history['loss'], label='Training
                                          loss')
plt.plot(history.history['val_loss'], label =
                    'Validation Loss')
plt.title('Training and validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output :

Mean Squared Error : 0.0168



Result :

Thus the Artificial Neural Networks for an application has been implemented successfully.

EX : 8
Date : 11/10/24

Introduction to prolog

Aim

To learn prolog terminologies and write basic programs.

Terminologies

1. Atomic Terms :

Atomic terms are usually strings made of lower and uppercase letters digits and the underscore, starting with a lowercase letter.

ex : dog
ab_c_321

2. Variables

Variables are strings of letters, digits and the underscore, starting with a capital letter or an underscore

Ex : Dog
Apple_420

3. Compound Terms :

Compound terms are made up of PROLOG atom and a number of arguments enclosed in parantheses and separated by commas.

Ex : BS_bigger(elephant, X)
f(g(X, _), 7)

4. Facts

A fact is a predicate followed by a dot

ex : bigger_animal(whale)
life_is_beautiful

5. Rules :

A rule consists of a head (a predicate) and a body (a ~~of predicates separated by~~

Ex :

ps-smaller(x,y) : ps bigger(y,x)

aunt(Aunt, Child) :- sister(Aunt, Parent), parent (Parent, Child)

Source Code

KB1 :

woman(mia)
woman(jody)
woman(yolando)
playsAirGuitar(jody)
party.

output :

Query1 : ?- woman(mia)
true

Query2 : ?- playsAirGuitar(mia)
false

Query3 : ?- Party
true

Query4 : ?- concert
error : Unknown procedure

KB 2 :

happy(yolanda)
listen2music(mia).
listen2music(yolanda) :- happy(yolanda)
playsAirGuitar(mia) :- listen2music(mia)
playsAirGuitar(yolanda) :- listens2music(yolanda)

Output :

?- playsAirGuitar(mia)
true
?- playsAirGuitar(yolanda)
true.

KB3 :

likes(dan, sally)
likes(sally, dan)
likes(john, brittney)
married(X,Y) :- likes(X,Y), likes(Y,X)
friends(X,Y) :- likes(X,Y); likes(Y,X)

output:

?- likes(dan, x)

x = sally

?- married(dan, sally)

true

? married(john, britney)

false

KB 4:

food(burger).

food(sandwich).

food(pizza).

lunch(sandwich).

dinner(pizza).

meal(x) :- food(x)

output:

?- food(pizza)

true

?- meal(x), lunch(x)

x = sandwich

?- dinner(sandwich).

false.

KB 5:

owns(jack, car(bmw)).

owns(john, car(chevy)).

owns(olivia, car(civic)).

owns(jane, car(chevy)).

sedan(car(bmw)).

sedan(car(bmw)).

sedan(car(civic)).

truck(car(chevy)).

Output:

?- owns (John, X)

X = car (chevy)

?- owns (John, _)

true

?. owns (who, car(chevy)).

who - John,

?. -owns(Jane, X), sedan(X)

false

?- owns (Jane, X), truck(X).

X = car(chevy).

Result:

thus the prolog programs are executed sucessfully.

Ex: 6
Date: 4/10/24

Aim:
To Implement a Minimax Algorithm.

Code:

```
import math

def minimax(depth, node_index, is_maximizer,
            scores, height):
    if depth == height:
        return scores[node_index]

    if is_maximizer:
        return max(minmax(depth+1, node_index
            * 2, False, scores, height),
            minmax(depth+1, node_index * 2+1,
            False, scores, height))

    else:
        return min(minimax(depth+1, node_index
            * 2, True, scores, height),
            (minimax(depth+1, node_index * 2+1,
                True, scores, height))

def calculate_tree_height(num_leaves):
    return math.ceil(math.log2(num_leaves))

scores = [3,5,6,9,1,2,0,-1]
tree_height = calculate_tree_height(len(scores))
optimal_score = minimax(0,0, True, scores,
                        tree_height)
print(f"The optimal score is: {optimal
                        score}")
```

Output:
The optimal score is 5

Result:
Thus the algorithm is implemented successfully.

Ex no : 9
Date : 11/10/24

Aim
To develop a family tree program using prolog in with all possible facts and queries

Source code :
Knowledge base :

```
/* FACTS :: */
male(peter)
male(john)
male(chris)
male(kevin)

female(betty)
female(jeny)
female(lisa)
female(helen)

parentof(chris, peter)
parentof(chris, betty)
parentof(helen, peter)
parentof(helen, betty)
parentof(kevin, chris)
parentof(kevin, lisa)
parentof(jeny, john)
parentof(jeny, helen)

RULES
/* son, parent
* son, grandparent */
father(X, Y) :- male(Y) parentof(X, Y).
```

O/P: X = chris , Y = peter

```
Mother(X, Y) :- female(Y), parentof(X, Y)
```

O/P: X = chris Y = betty.

grandfather(X,Y) = female(Y), parentof(X,Z), parentof(Z,Y).

O/P : X=kevin Y=peter Z=chris

grandmother(X,Y): female(Y), parentof(X,Z), parentof(Z,Y)

O/P : X=kevin Y=betty Z=chris

brother(X,Y) :- male(Y), father(X,Z), father(Y,W), Z=>W

O/P : father(A,B) does not exist

sister(X,Y): female(Y), father(X,Z), father(Y,W), Z=>W

O/P : procedure 'father(A,B)' does not exist.

Result :

thus the prolog family tree is developed successfully.