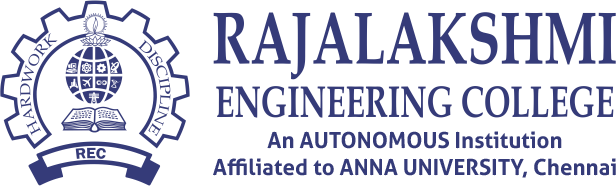
**RAJALAKSHMI ENGINEERING COLLEGE**

**RAJALAKSHMI NAGAR, THANDALAM, CHENNAI 602105**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**AI19341**

**PRINCIPLES OF ARTIFICIAL INTELLIGENCE**

**THIRD YEAR**

**FIFTH SEMESTER**

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.NO** | **DATE** | **EXP NAME** | **VIVA MARK** | **SIGNATURE** |
| **1** | **09 - 08 - 2024** | **8QUEENS PROBLEM** |  |  |
| **2** | **16 - 08 - 2024** | **DEPTH FIRST SEARCH** |  |  |
| **3** | **23 - 08 - 2024** | **DEPTH FIRST SEARCH - WATER JUG PROBLEM** |  |  |
| **4** | **30 - 08 - 2024** | **MINIMAX ALGORITHM** |  |  |
| **5** | **06 - 09 - 2024** | **A\* SEARCH ALGORITHM** |  |  |
| **6** | **27 - 09 -2024** | **INTRODUCTION TO PROLOG** |  |  |
| **7** | **04 - 10 - 2024** | **PROLOG FAMILY TREE** |  |  |
| **8** | **18 - 10 - 2024** | **IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - REGRESSION** |  |  |
| **9** | **25 - 10 - 2024** | **IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUES** |  |  |
| **10** | **08 - 11 - 2024** | **IMPLEMENTATION OF CLUSTERING TECHNIQUES K - MEANS** |  |  |

**EX.NO: 1 DATE: 09 - 08 - 2024**

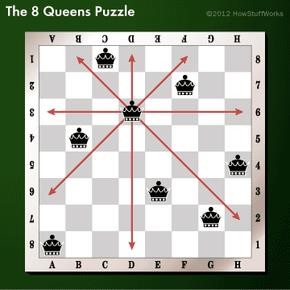
**8- QUEENS PROBLEM**

**AIM:**

To implement an 8-Queens problem using Python.

You are given an 8x8 board; find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, same column, or the same diagonal as any other queen. Print all the possible configurations.

To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For the given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.



**PROGRAM:**

def share\_diagonal(x0, y0, x1, y1):

dx = abs(x0 - x1)

dy = abs(y0 - y1)

return dy == dx

def col\_clashes(bs, c):

for i in range(c):

if share\_diagonal(i, bs[i], c, bs[c]):

return True

return False

def has\_clashes(the\_board):

for col in range(1, len(the\_board)):

if col\_clashes(the\_board, col):

return True

return False

def main():

import random

n=int(input("Enter the number of queens: "))

rng = random.Random()

bd = list(range(n))

num\_found = 0

tries = 0

result = []

while num\_found < 5:

rng.shuffle(bd)

tries += 1

if not has\_clashes(bd) and bd not in result:

print("Found solution {0} in {1} tries.".format(bd, tries))

tries = 0

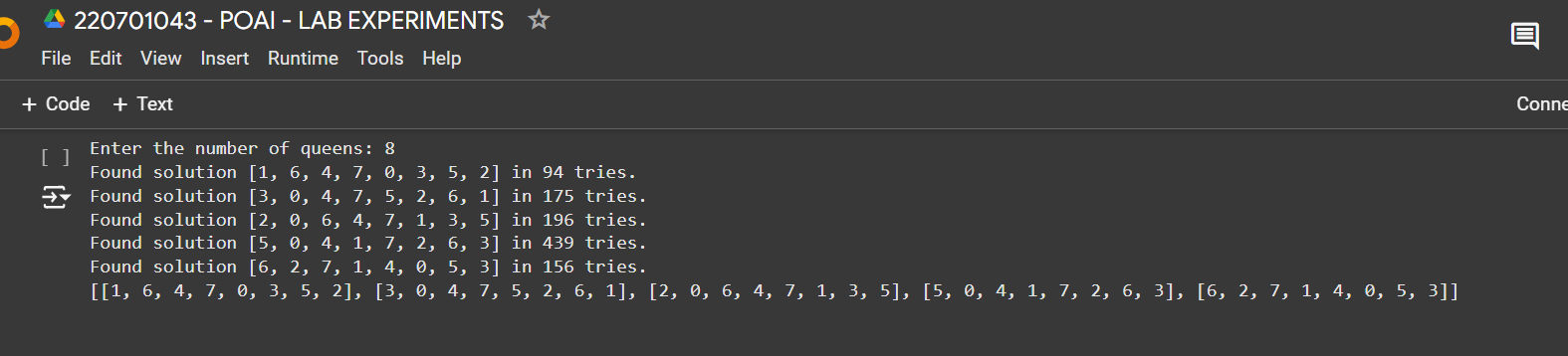
num\_found += 1

result.append(list(bd))

print(result)

main()

**OUTPUT:**



**RESULT:**

Thus, the 8Queens problem was implemented successfully using backtracking algorithm.

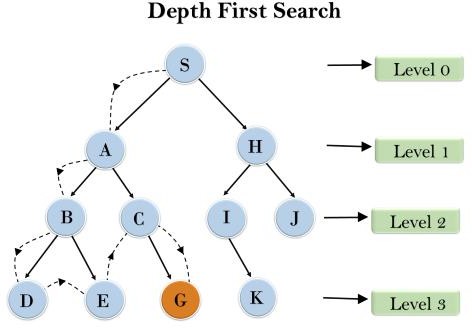
**EX.NO: 2 DATE: 16 - 08 - 2024**

**DEPTH FIRST SEARCH**

**AIM:**

To implement a depth-first search problem using Python.

* Depth-first search (DFS) algorithm or searching technique starts with the root node of graph G, and then travel deeper and deeper until we find the goal node or the node which has no children by visiting different node of the tree.
* The algorithm, then backtracks or returns back from the dead end or last node towards the most recent node that is yet to be completely unexplored.
* The data structure (DS) which is being used in DFS Depth-first search is stack. The process is quite similar to the BFS algorithm.
* In DFS, the edges that go to an unvisited node are called discovery edges while the edges that go to an already visited node are called block edges.



**PROGRAM:**

def dfs(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(start, end=" ")

    for neighbour in graph.get(start, []):

        if neighbour not in visited:

            dfs(graph, neighbour, visited)

num\_nodes = int(input("Enter the number of nodes: "))

graph = {}

for i in range(num\_nodes):

    node = input("Enter node " + str(i+1) + ": ").strip()

    neighbors = input("Enter neighbors of " + node + " (comma-separated): ").strip().split(',')

    neighbors = [n.strip() for n in neighbors]

    graph[node] = neighbors

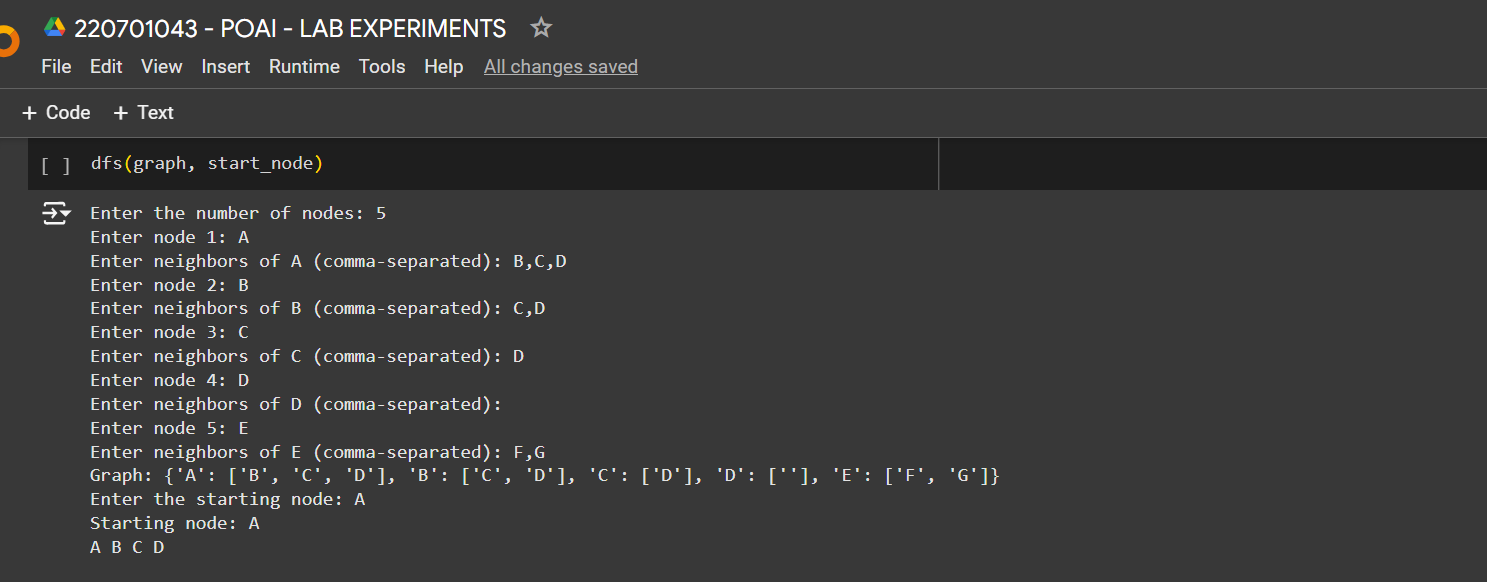
print("Graph:", graph)

start\_node = input("Enter the starting node: ").strip()

print("Starting node:", start\_node)

dfs(graph, start\_node)

**OUTPUT:**



**RESULT:**

Thus, a searching technique using Depth-First search algorithm was implemented successfully.

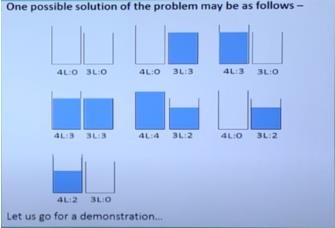
**EX.NO: 3 DATE: 23 - 08 - 2024**

**DEPTH FIRST SEARCH-WATER JUG PROBLEM**

**AIM:**

To implement **Water – jug problem** using depth first search algorithm.

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent’s task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.



**PROGRAM:**

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)

def dfs\_water\_jug(x\_max, y\_max, goal\_x, 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}")

        if x == goal\_x:

            print(f"Goal reached: {state}")

            return 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)

        ]

        for new\_state in next\_states:

            if new\_state not in visited:

                stack.append(new\_state)

    return None

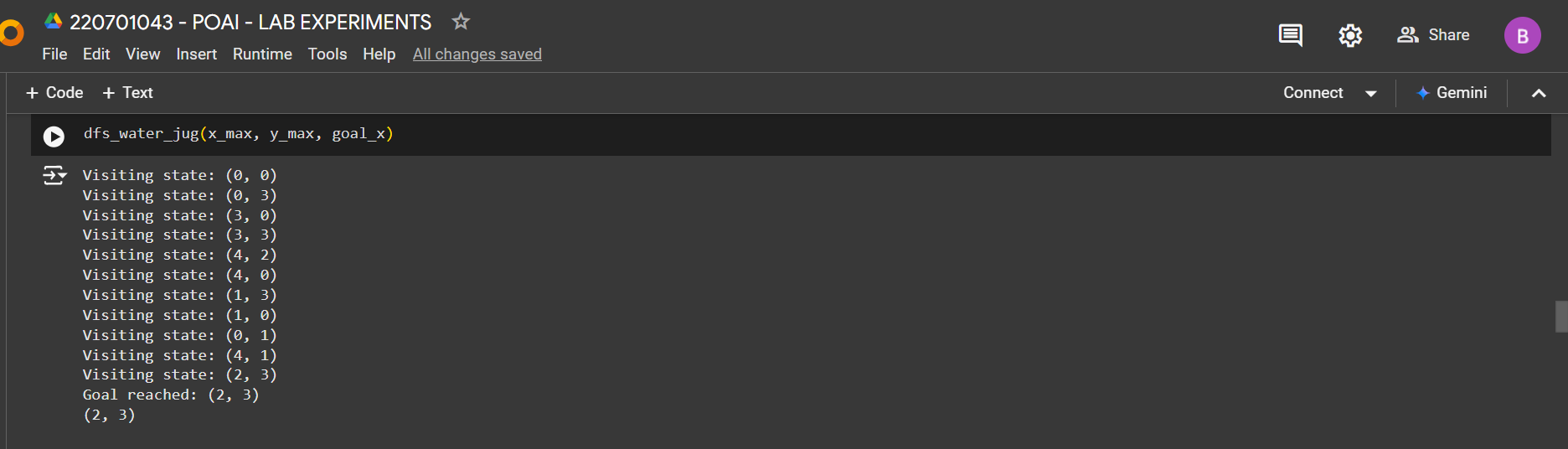
x\_max = 4

y\_max = 3

goal\_x = 2

dfs\_water\_jug(x\_max, y\_max, goal\_x)

**OUTPUT:**



**RESULT:**

Thus the water-jug problem is implemented successfully using depth-first search algorithm.

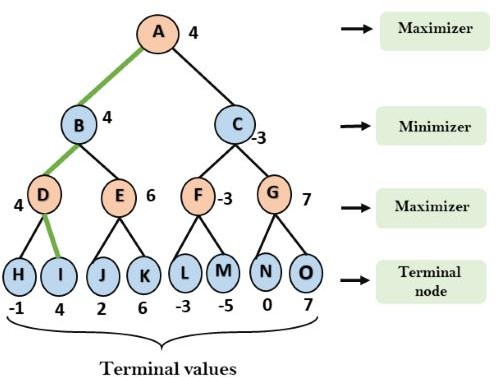
**EX.NO: 4 DATE: 30 - 08 - 2024**

**MINIMAX ALGORITHM**

**AIM:**

To implement the Minimax Algorithm for a two-player game, with Maximizer maximizing the score and Minimizer minimizing it through DFS evaluation.

* A simple example can be used to explain how the minimax algorithm works. We've included an example of a game-tree below, which represents a two-player game.
* There are two players in this scenario, one named Maximizer and the other named Minimizer.
* Maximizer will strive for the highest possible score, while Minimizer will strive for the lowest possible score.
* Because this algorithm uses DFS, we must go all the way through the leaves to reach the terminal nodes in this game-tree.
* The terminal values are given at the terminal node, so we'll compare them and retrace the tree till we reach the original state.



**PROGRAM:**

import math

def minimax(depth, node\_index, is\_maximizer, scores, height):

    if depth == height:

        return scores[node\_index]

    if is\_maximizer:

        return max(minimax(depth + 1, node\_index \* 2, False, scores, height),

                   minimax(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))

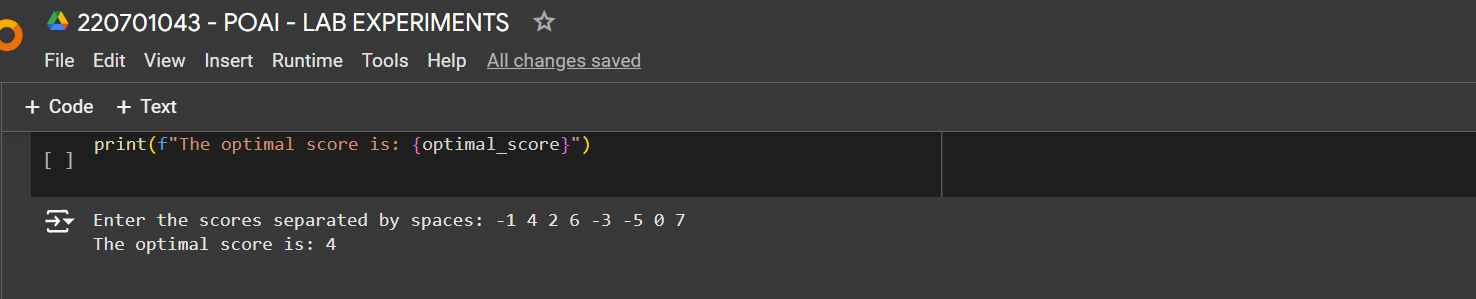
scores = list(map(int, input("Enter the scores separated by spaces: ").split()))

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:**



**RESULT:**

Thus ,the Minimax Algorithm successfully determines the optimal moves for both players by evaluating the game-tree and selecting the best possible scores for Maximizer and Minimizer.

**EX.NO: 5 DATE: 06 - 09 - 2024**

**A\* SEARCH ALGORITHM**

**AIM:**

To implement a A\* heuristic algorithm to find the least-cost path in a graph using node weights and heuristic approximations for efficient traversal.

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

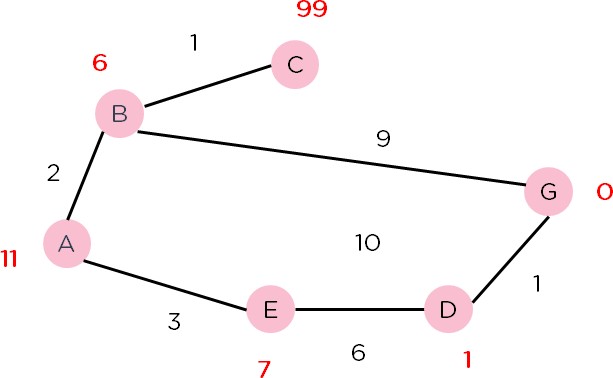
Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If *f(n)* represents the final cost, then it can be denoted as :

*f(n) = g(n) + h(n),*

where:

*g(n)* = cost of traversing from one node to another. This will vary from node to node

*h(n)* = heuristic approximation of the node's value. This is not a real value but an approximation cost.



**PROGRAM:**

import heapq

class Node:

    def \_\_init\_\_(self, name, parent=None, g=0, h=0):

        self.name = name

        self.parent = parent

        self.g = g

        self.h = h

        self.f = g + h

    def \_\_lt\_\_(self, other):

        return self.f < other.f

def a\_star(graph, start, goal, h\_values):

    open\_list = []

    heapq.heappush(open\_list, Node(start, None, 0, h\_values[start]))

    closed\_list = set()

    while open\_list:

        current\_node = heapq.heappop(open\_list)

        if current\_node.name == goal:

            path = []

            while current\_node:

                path.append(current\_node.name)

                current\_node = current\_node.parent

            return path[::-1]

        closed\_list.add(current\_node.name)

        for neighbor, cost in graph.get(current\_node.name, []):

            if neighbor in closed\_list:

                continue

            g\_new = current\_node.g + cost

            h\_new = h\_values[neighbor]

            f\_new = g\_new + h\_new

            neighbor\_node = Node(neighbor, current\_node, g\_new, h\_new)

            heapq.heappush(open\_list, neighbor\_node)

    return None

graph = {}

h\_values = {}

print("Enter heuristic values for each node. Type 'nil' to stop.")

while True:

    node = input("Enter node name (or 'nil' to finish): ")

    if node.lower() == 'nil':

        break

    h\_value = int(input(f"Enter heuristic value for {node}: "))

    h\_values[node] = h\_value

print("Enter edges and their costs. Type 'nil' to stop.")

while True:

    node1 = input("Enter the start node (or 'nil' to finish): ")

    if node1.lower() == 'nil':

        break

    node2 = input("Enter the end node: ")

    cost = int(input(f"Enter the cost from {node1} to {node2}: "))

    if node1 not in graph:

        graph[node1] = []

    graph[node1].append((node2, cost))

start\_node = input("Enter the start node: ")

goal\_node = input("Enter the goal node: ")

path = a\_star(graph, start\_node, goal\_node, h\_values)

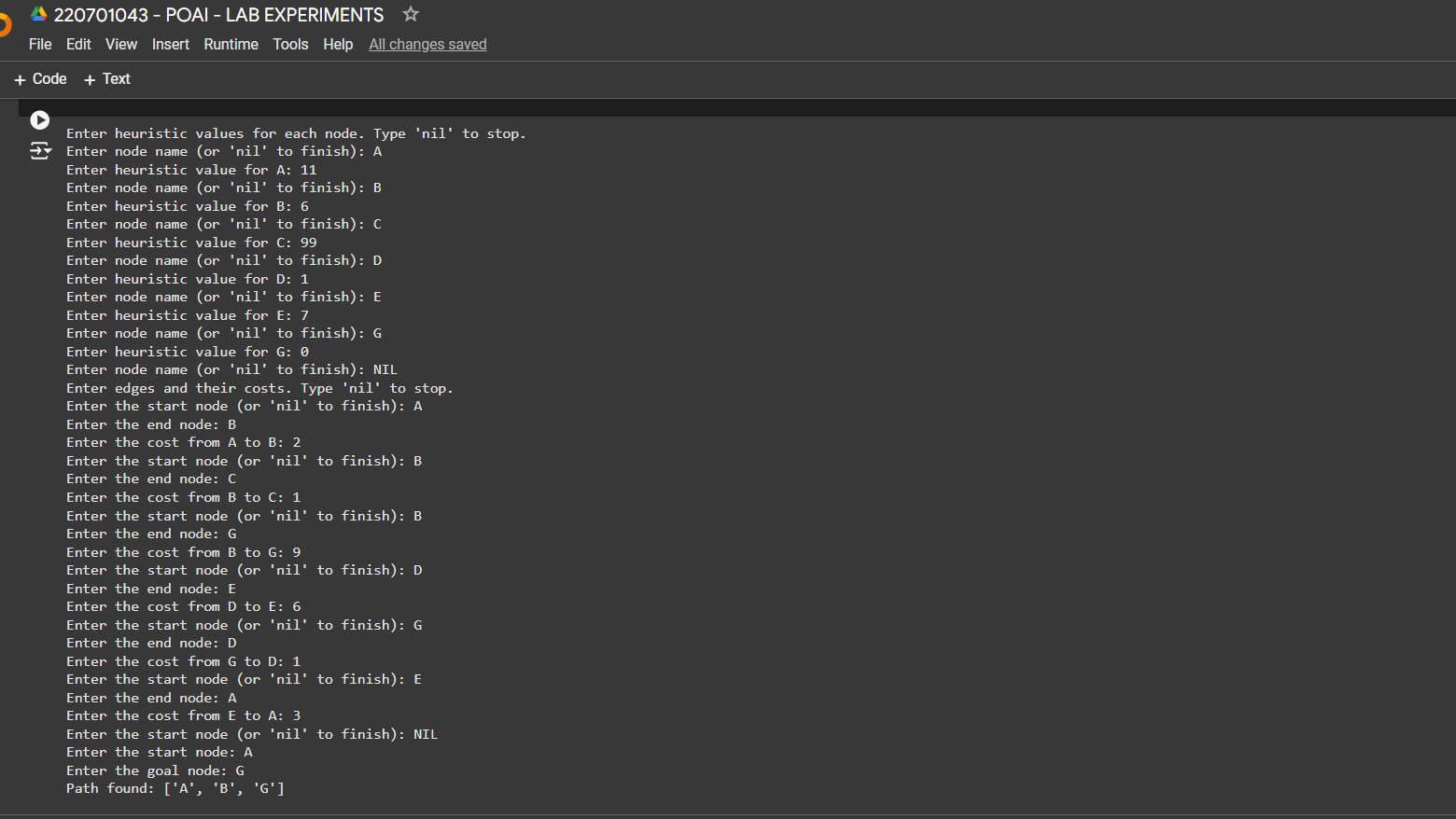
if path:

    print(f"Path found: {path}")

else:

    print("No path found")

**OUTPUT:**



**RESULT:**

Thus , the heuristic algorithm successfully identifies an efficient, least-cost path in the graph by evaluating node weights and heuristic estimates.

**EX.NO: 6 DATE: 27 - 09 -2024**

**INTRODUCTION TO PROLOG**

**AIM:**

To learn PROLOG terminologies and write basic programs.

**TERMINOLOGIES:**

1. Atomic Terms:

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

Ex:

dog

ab\_c\_321

1. Variables:

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

Ex:

Dog

Apple\_420

1. Compound Terms:

Compound terms are made up of a PROLOG atom and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.

Ex:

is\_bigger(elephant,X) f(g(X,\_),7)

1. Facts:

A fact is a predicate followed by a dot.

Ex:

bigger\_animal(whale). life\_is\_beautiful.

1. Rules:

A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).

Ex:

is\_smaller(X,Y):-is\_bigger(Y,X).

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

**SOURCE CODE:**

**KB1:**

woman(mia).

woman(jody).

woman(yolanda).

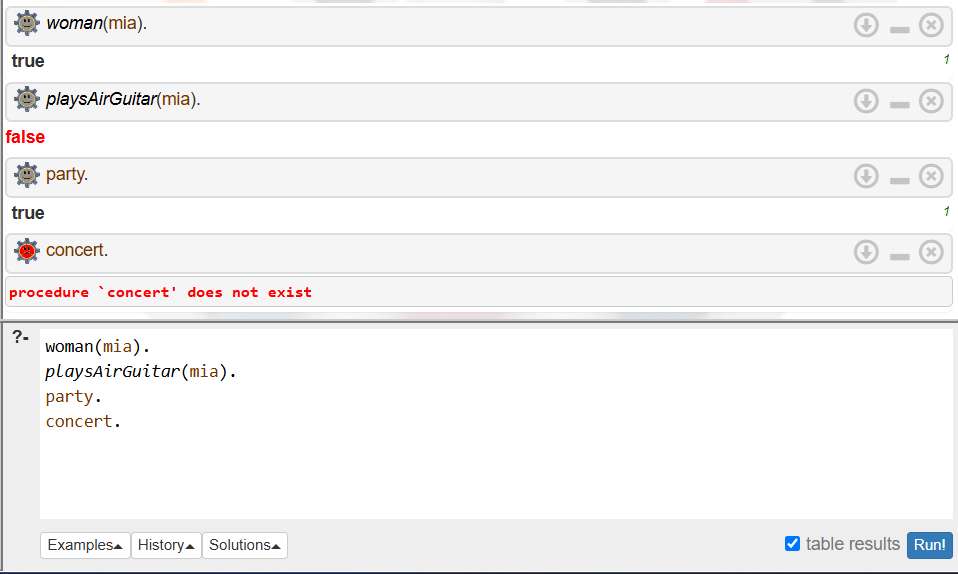
playsAirGuitar(jody). party.

Query 1: ?-woman(mia).

Query 2: ?-playsAirGuitar(mia).

Query 3: ?-party.

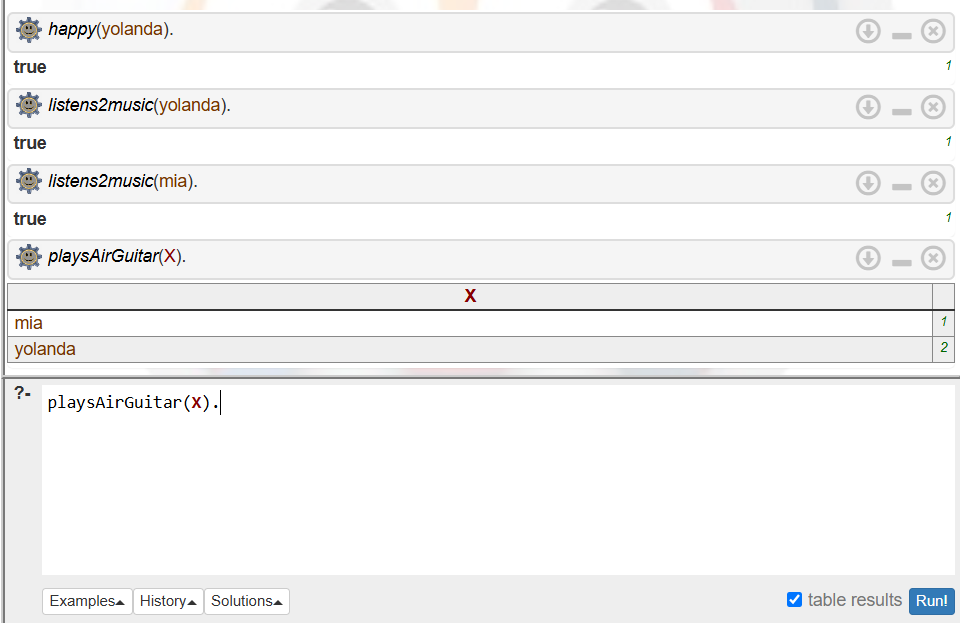
Query 4: ?-concert.



**KB2:**

happy(yolanda). listens2music(mia).

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

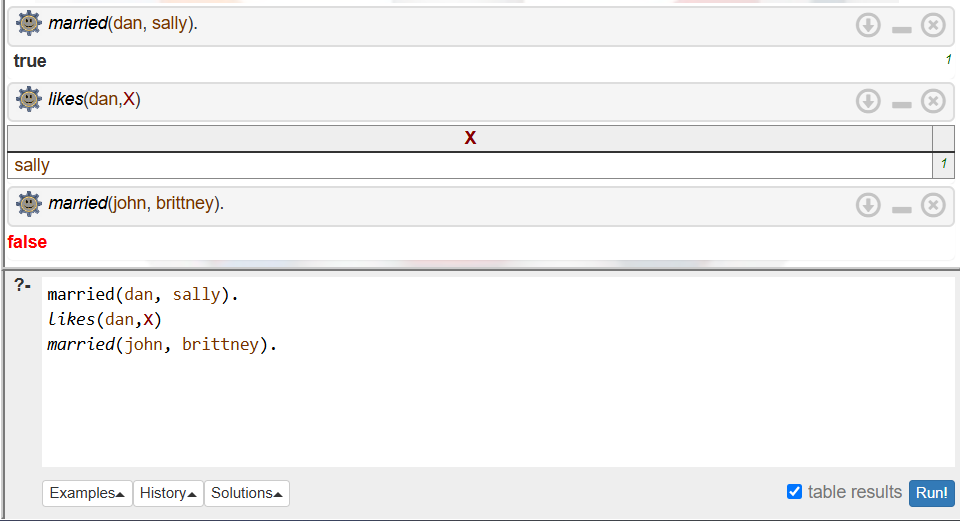


**KB3:**

likes(dan,sally). likes(sally,dan). likes(john,brittney).

married(X,Y) :- likes(X,Y) , likes(Y,X).

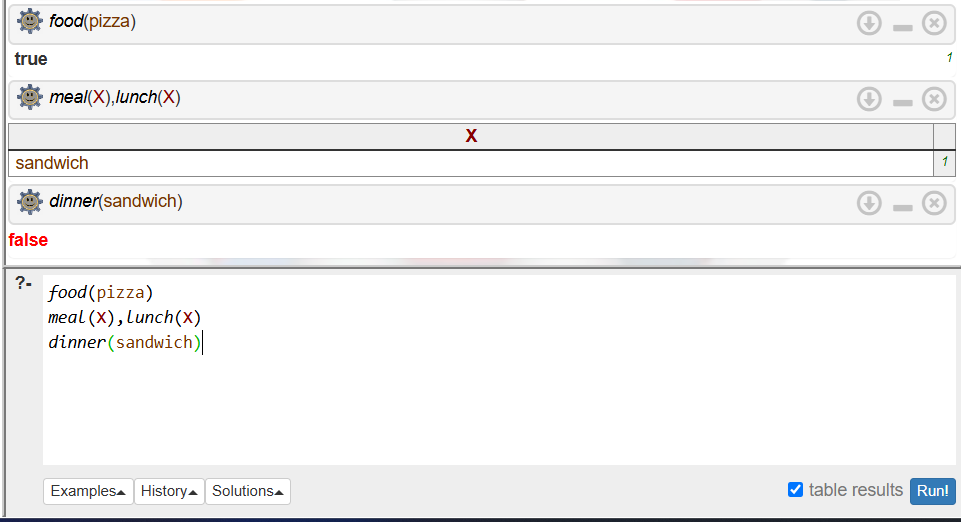
friends(X,Y) :- likes(X,Y) ; likes(Y,X).



**KB4:**

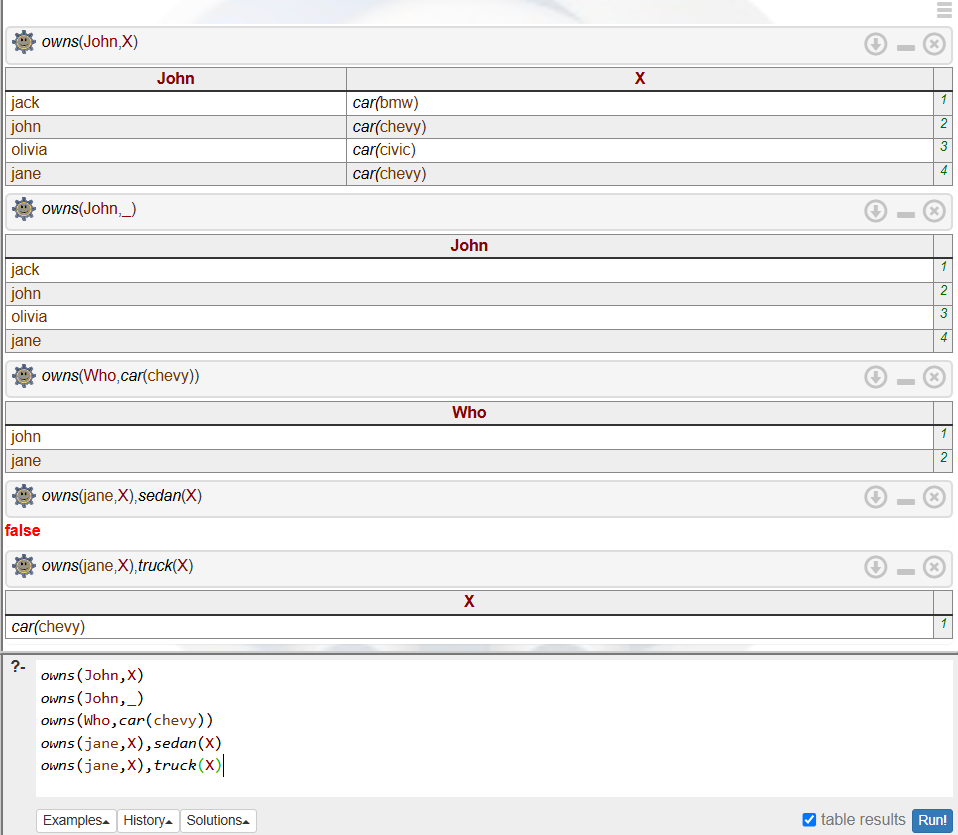
food(burger). food(sandwich). food(pizza). lunch(sandwich). dinner(pizza).

meal(X):-food(X).



**KB5:**

owns(jack,car(bmw)). owns(john,car(chevy)). owns(olivia,car(civic)). owns(jane,car(chevy)). sedan(car(bmw)). sedan(car(civic)). truck(car(chevy)).



**RESULT:**

Thus, we have written basic programs to learn prolog terminologies.

**EX.NO: 7 DATE: 04 - 10 - 2024**

**PROLOG FAMILY TREE**

**AIM:**

To develop a family tree program using PROLOG with all possible facts, rules, 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).

mother(X,Y):- female(Y), parentOf(X,Y).

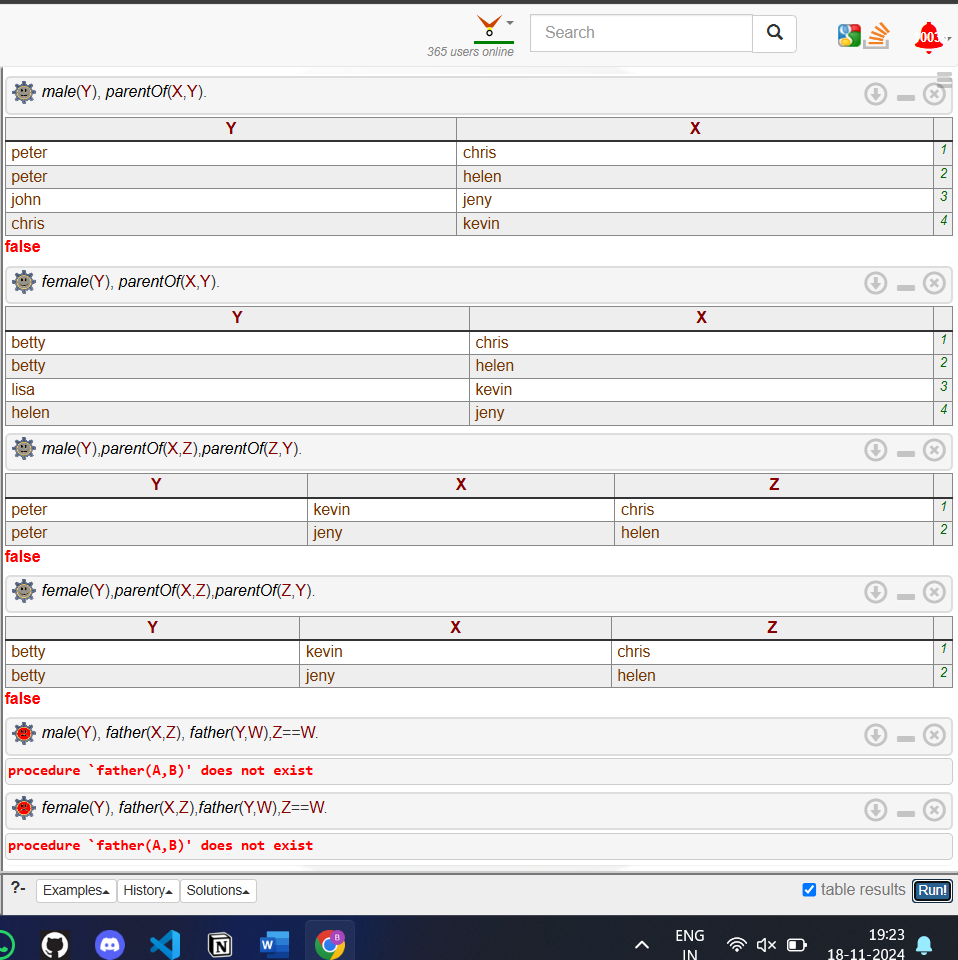
grandfather(X,Y):- male(Y),parentOf(X,Z),parentOf(Z,Y).

grandmother(X,Y):- female(Y),parentOf(X,Z),parentOf(Z,Y).

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

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

**OUTPUT:**



**RESULT:**

Thus, we have developed a family tree program using PROLOG with all possible facts, rules, and queries.

**EX.NO: 8 DATE: 18 - 10 - 2024**

**IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON – REGRESSION**

**Regression using Artificial Neural Networks**

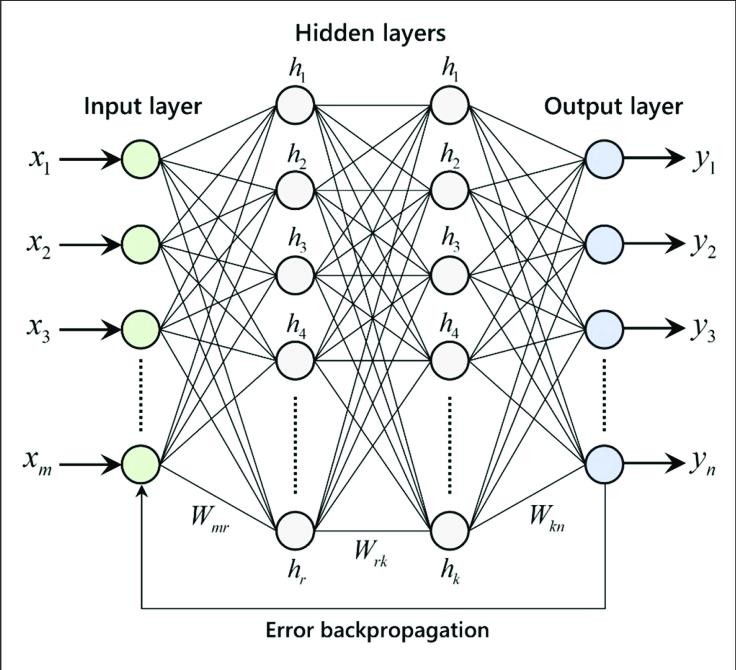
Why do we need to use Artificial Neural Networks for Regression instead of simply using Linear Regression?

The purpose of using Artificial Neural Networks for Regression over Linear Regression is that the linear regression can only learn the linear relationship between the features and target and therefore cannot learn the complex non-linear relationship. In order to learn the complex non-linear relationship between the features and target, we are in need of other techniques. One of those techniques is to use Artificial Neural Networks. Artificial Neural Networks have the ability to learn the complex relationship between the features and target due to the presence of activation function in each layer. Let’s look at what are Artificial Neural Networks and how do they work.

**Artificial Neural Networks**

Artificial Neural Networks are one of the deep learning algorithms that simulate the workings of neurons in the human brain. There are many types of Artificial Neural Networks, Vanilla Neural Networks, Recurrent Neural Networks, and Convolutional Neural Networks. The Vanilla Neural Networks have the ability to handle structured data only, whereas the Recurrent Neural Networks and Convolutional Neural Networks have the ability to handle unstructured data very well. In this post, we are going to use Vanilla Neural Networks to perform the Regression Analysis.

**Structure of Artificial Neural Networks**



The Artificial Neural Networks consists of the Input layer, Hidden layers, Output layer. The hidden layer can be more than one in number. Each layer consists of n number of neurons. Each layer will be having an Activation Function associated with each of the neurons. The activation function is the function that is responsible for introducing non-linearity in the relationship. In our case, the output layer must contain a linear activation function. Each layer can also have regularizers associated with it. Regularizers are responsible for preventing overfitting.

Artificial Neural Networks consists of two phases,

•Forward Propagation

•Backward Propagation

Forward propagation is the process of multiplying weights with each feature and adding them. The bias is also added to the result.

Backward propagation is the process of updating the weights in the model. Backward propagation requires an optimization function and a loss function.

**AIM:**

To implement artificial neural networks for an application in Regression using python.

**PROGRAM:**

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import make\_regression

X, y = make\_regression(n\_samples=1000, n\_features=5, noise=0.1)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

model = Sequential()

model.add(Dense(units=64, activation='relu', input\_dim=X\_train.shape[1]))

model.add(Dense(units=32, activation='relu'))

model.add(Dense(units=1, activation='linear'))

model.compile(optimizer='adam', loss='mean\_squared\_error')

model.fit(X\_train, y\_train, epochs=5, batch\_size=32, verbose=1)

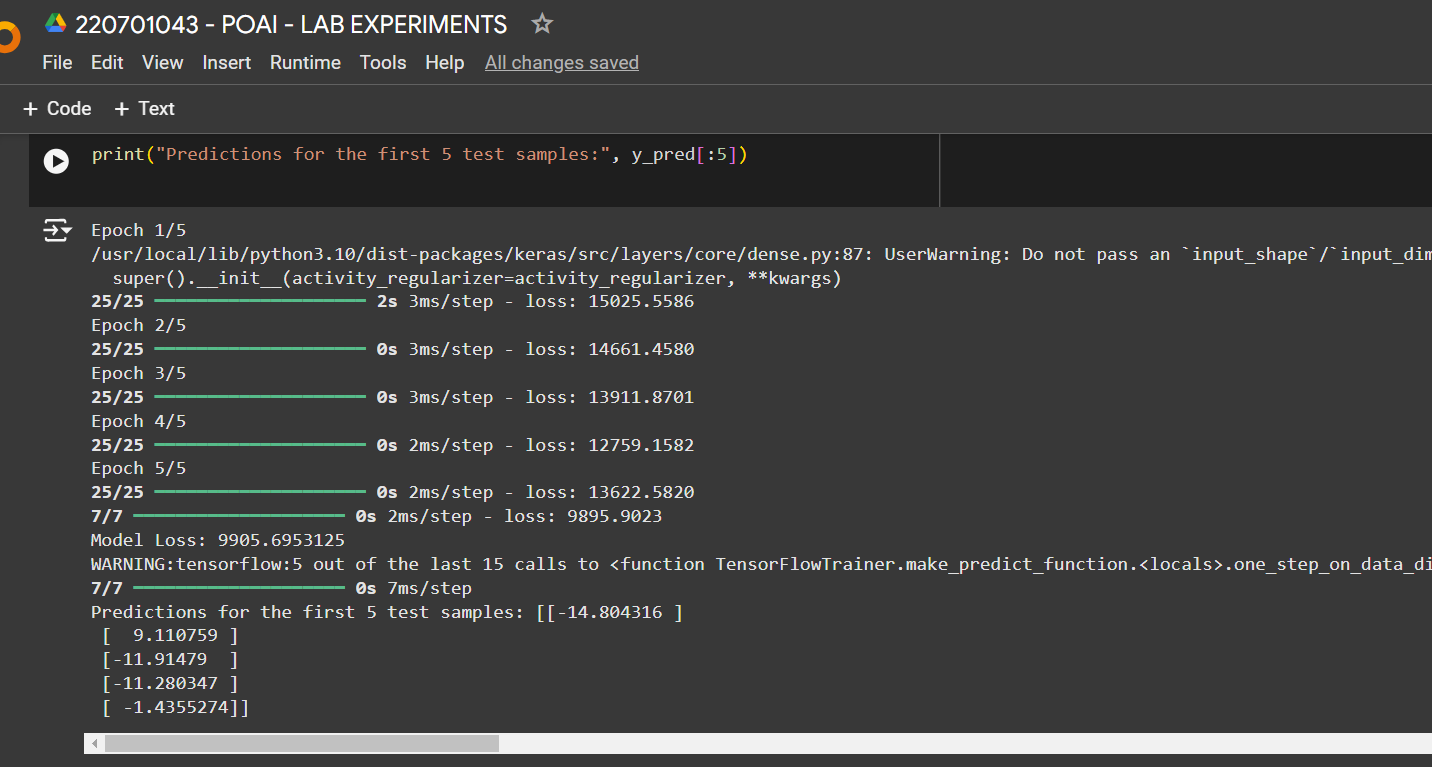
loss = model.evaluate(X\_test, y\_test)

print(f"Model Loss: {loss}")

y\_pred = model.predict(X\_test)

print("Predictions for the first 5 test samples:", y\_pred[:5])

**OUTPUT:**



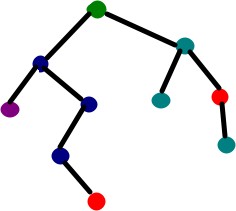
**RESULT:**

Thus , we have successfully implemented artificial neural networks for an application in regression using python.

**EX.NO: 9 DATE: 25 - 10 - 2024**

**IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUES**

Decision Tree is one of the most powerful and popular algorithm. Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.



**AIM:**

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

**EXPLANATION:**

* Import tree from sklearn.
* Call the function DecisionTreeClassifier() from tree
* Assign values for X and Y.
* Call the function predict for Predicting on the basis of given random values for each given feature.
* Display the output.

**PROGRAM:**

from sklearn import tree

X = [[150, 50, 37], [160, 60, 38], [170, 70, 39], [180, 80, 40], [165, 55, 36]]

Y = [0, 0, 1, 1, 0]

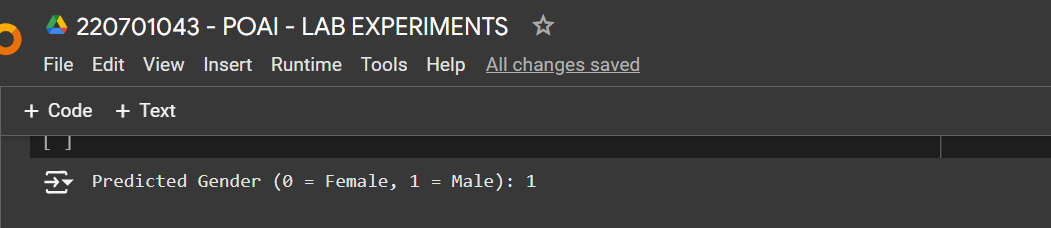
clf = tree.DecisionTreeClassifier()

clf = clf.fit(X, Y)

prediction = clf.predict([[175, 75, 41]])

print("Predicted Gender (0 = Female, 1 = Male):", prediction[0])

**OUTPUT:**

**RESULT:**

Thus, we have successfully implemented a decision tree classification techniques for gender classification.

**EX.NO: 10 DATE: 08 - 11 - 2024**

**IMPLEMENTATION OF CLUSTERING TECHNIQUES K – MEANS**

The k-means clustering method is an unsupervised machine learning technique used to identify clusters of data objects in a dataset. There are many different types of clustering methods, but k- means is one of the oldest and most approachable. These traits make implementing k-means clustering in Python reasonably straightforward, even for novice programmers and data scientists.

If you’re interested in learning how and when to implement k-means clustering in Python, then this is the right place. You’ll walk through an end-to-end example of k-means clustering using Python, from preprocessing the data to evaluating results.

**How does it work?**

First, each data point is randomly assigned to one of the K clusters. Then, we compute the centroid (functionally the center) of each cluster, and reassign each data point to the cluster with the closest centroid. We repeat this process until the cluster assignments for each data point are no longer changing.

K-means clustering requires us to select K, the number of clusters we want to group the data into. The elbow method lets us graph the inertia (a distance-based metric) and visualize the point at which it starts decreasing linearly. This point is referred to as the "eblow" and is a good estimate for the best value for K based on our data.

**AIM:**

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

**EXPLANATION:**

* Import KMeans from sklearn.cluster
* Assign X and Y.
* Call the function KMeans().
* Perform scatter operation and display the output.

**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import make\_blobs

X, \_ = make\_blobs(n\_samples=300, centers=4, random\_state=42)

plt.scatter(X[:, 0], X[:, 1], s=30)

plt.title("Generated Data Points")

plt.show()

kmeans = KMeans(n\_clusters=4, random\_state=42)

kmeans.fit(X)

centroids = kmeans.cluster\_centers\_

labels = kmeans.labels\_

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=30)

plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200, label='Centroids')

plt.title("K-Means Clustering")

plt.legend()

plt.show()

print(end="\n")

print("Centroids of the clusters:")

print(centroids,end="\n\n")

inertia = []

for k in range(1, 11):

    kmeans = KMeans(n\_clusters=k, random\_state=42)

    kmeans.fit(X)

    inertia.append(kmeans.inertia\_)

plt.plot(range(1, 11), inertia, marker='o')

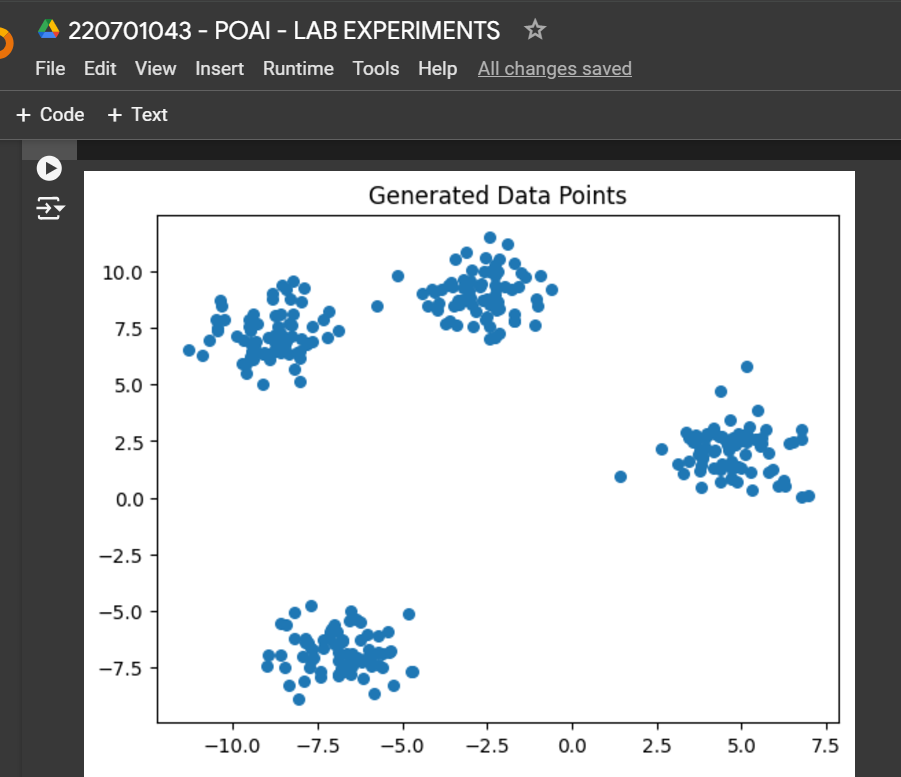
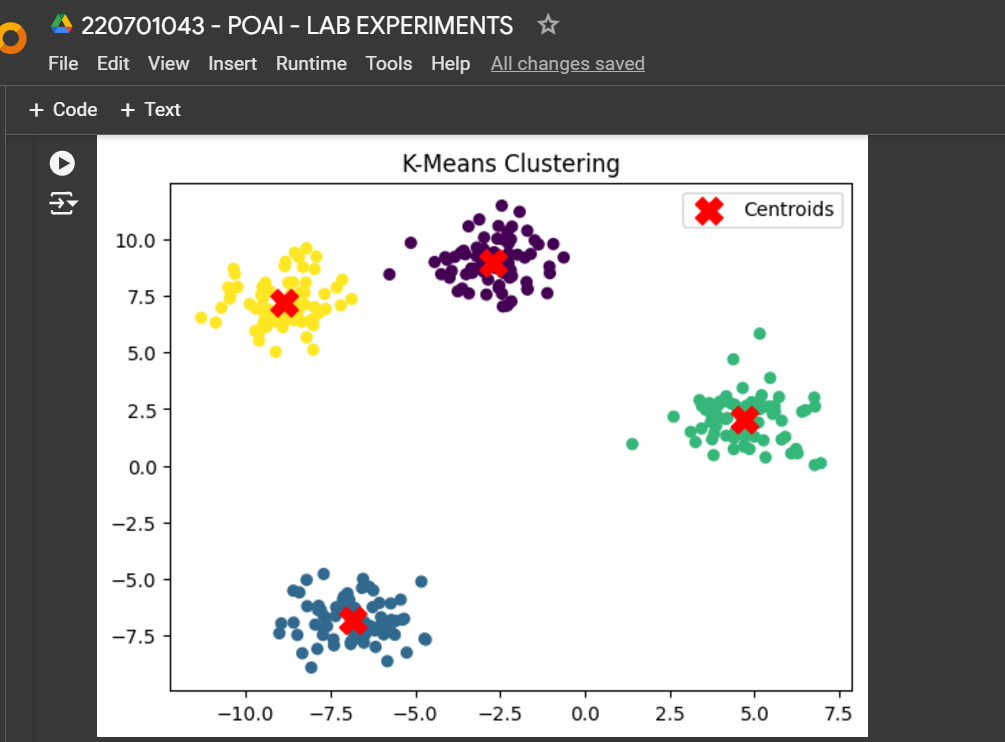
plt.title("Elbow Method to find optimal K")

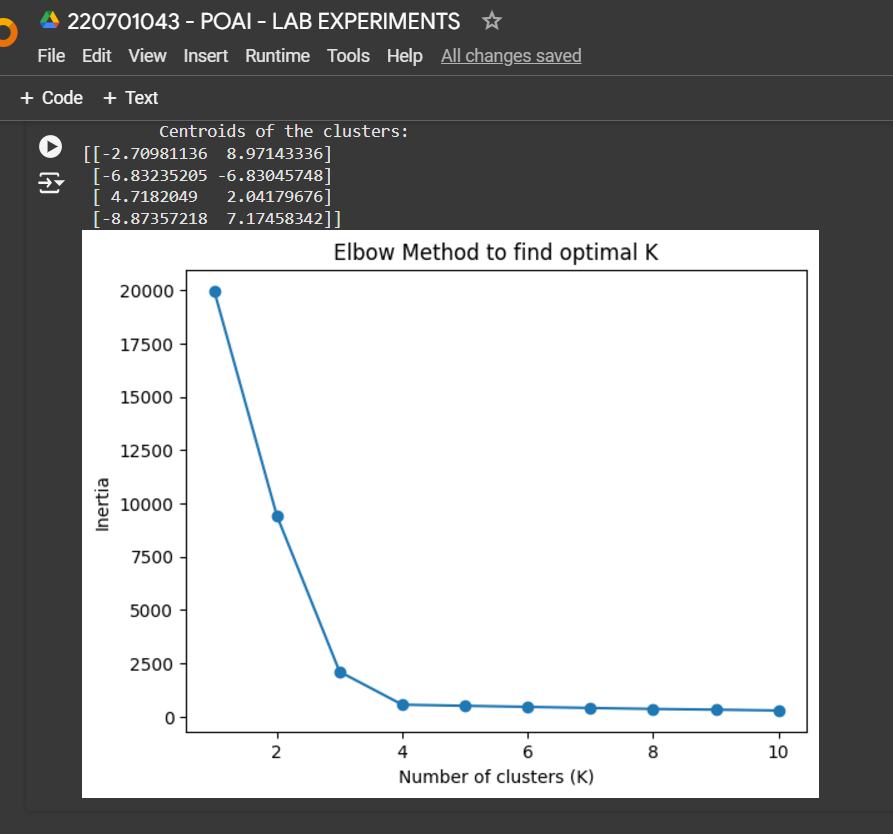
plt.xlabel("Number of clusters (K)")

plt.ylabel("Inertia")

plt.show()

**OUTPUT:**



**RESULT:**

Thus, we have successfully implemented a K-Means clustering technique using python language.