RAJALAKSHMI ENGINEERING COLLEGE (Autonomous)

RAJALAKSHMI NAGAR, THANDALAM, CHENNAI-602105

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RAJALAKSHMI ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

AI19341

PRINCIPLES OF ARTIFICIAL INTELLIGENCE LAB

THIRD YEAR

FIFTH SEMESTER

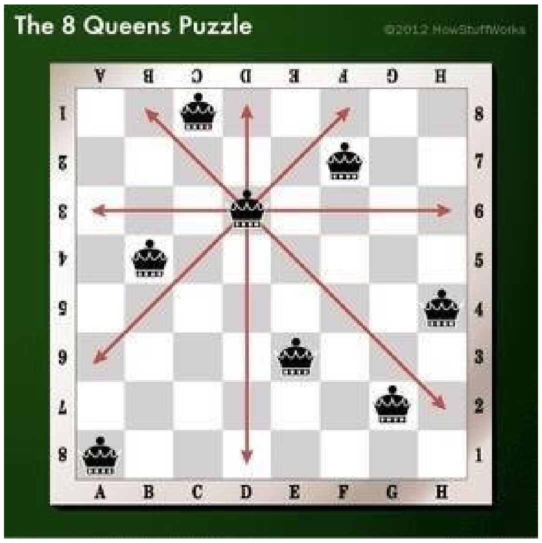
INDEX

EX.NO: 1 **DATE: 13-09-24**

8- QUEENS PROBLEM

AIM :
To implement an 8-Queesns 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.

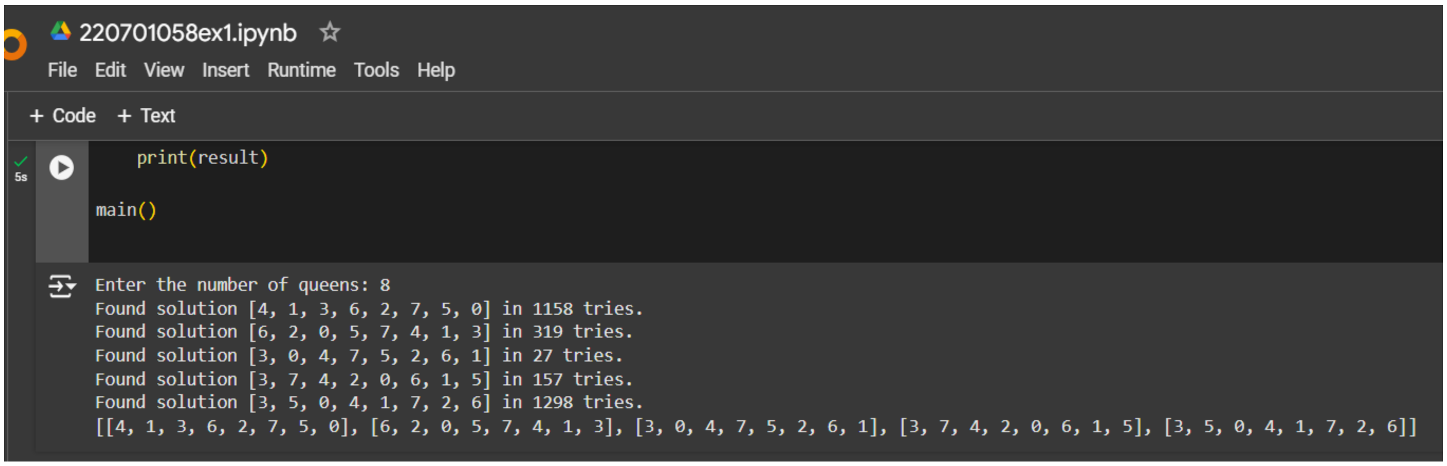


CODE:

```
 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:



```
    print(result)

main()
```

```
Enter the number of queens: 8
Found solution [4, 1, 3, 6, 2, 7, 5, 0] in 1158 tries.
Found solution [6, 2, 0, 5, 7, 4, 1, 3] in 319 tries.
Found solution [3, 0, 4, 7, 5, 2, 6, 1] in 27 tries.
Found solution [3, 7, 4, 2, 0, 6, 1, 5] in 157 tries.
Found solution [3, 5, 0, 4, 1, 7, 2, 6] in 1298 tries.
[[4, 1, 3, 6, 2, 7, 5, 0], [6, 2, 0, 5, 7, 4, 1, 3], [3, 0, 4, 7, 5, 2, 6, 1], [3, 7, 4, 2, 0, 6, 1, 5], [3, 5, 0, 4, 1, 7, 2, 6]]
```

RESULT:

The code for N-Queens problem has been executed successfully.

EX.NO:2

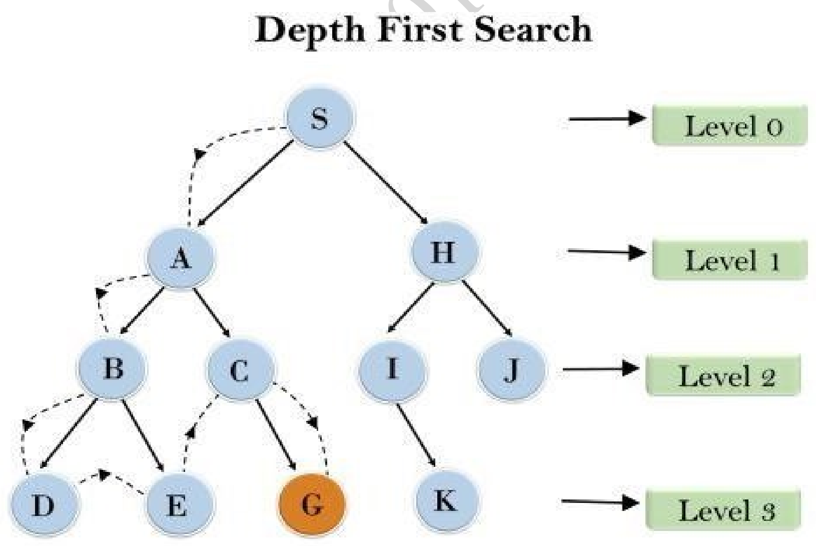
DATE:13-09-24

DEPTH-FIRST SEARCH

AIM :

To implement a depth-first search problem using Python.

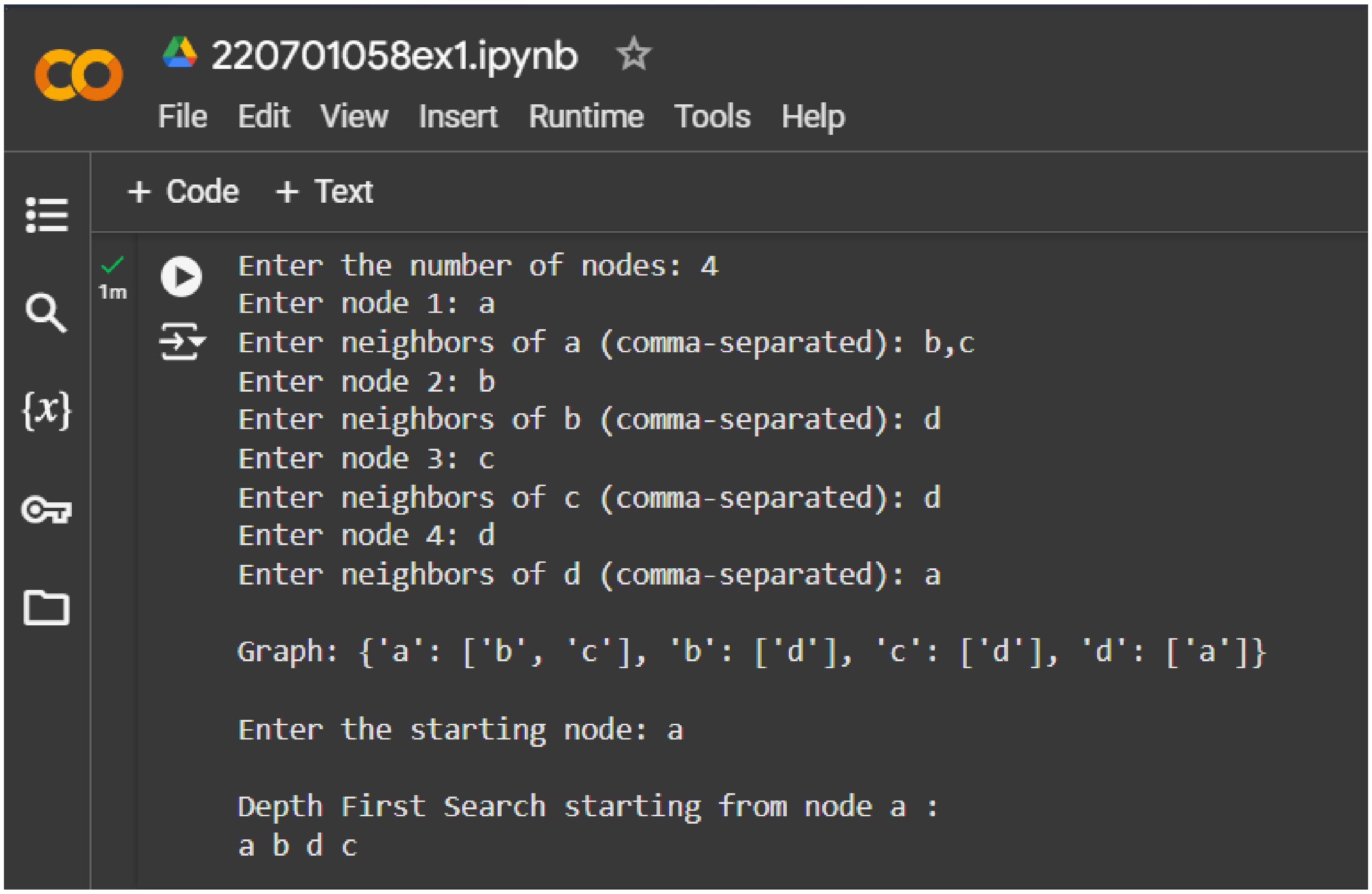
- Depth-first search (DFS) algorithm or searching technique starts with the root node of graph G, and then travel deeper and deeper until we find the goal node or the node which has no children by visiting different node of the tree.
- The algorithm, then backtracks or returns back from the dead end 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.



CODE:

```
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:



```
Enter the number of nodes: 4
Enter node 1: a
Enter neighbors of a (comma-separated): b,c
Enter node 2: b
Enter neighbors of b (comma-separated): d
Enter node 3: c
Enter neighbors of c (comma-separated): d
Enter node 4: d
Enter neighbors of d (comma-separated): a

Graph: {'a': ['b', 'c'], 'b': ['d'], 'c': ['d'], 'd': ['a']}

Enter the starting node: a

Depth First Search starting from node a :
a b d c
```

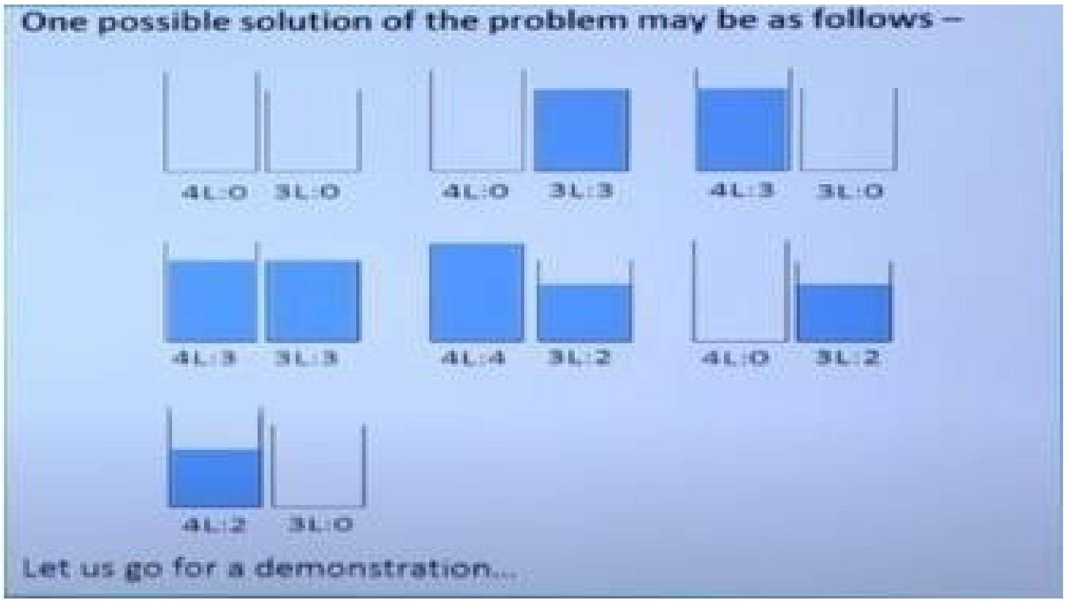
RESULT:

The code for DFS has been successfully executed.

EX.NO: 3 **DATE:** **20-09-24**

DEPTH-FIRST SEARCH – WATER JUG PROBLEM

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.


One possible solution of the problem may be as follows –
4L:0 3L:0
4L:0 3L:3
4L:3 3L:0
4L:3 3L:3
4L:4 3L:2
4L:0 3L:2
4L:2 3L:0
Let us go for a demonstration...

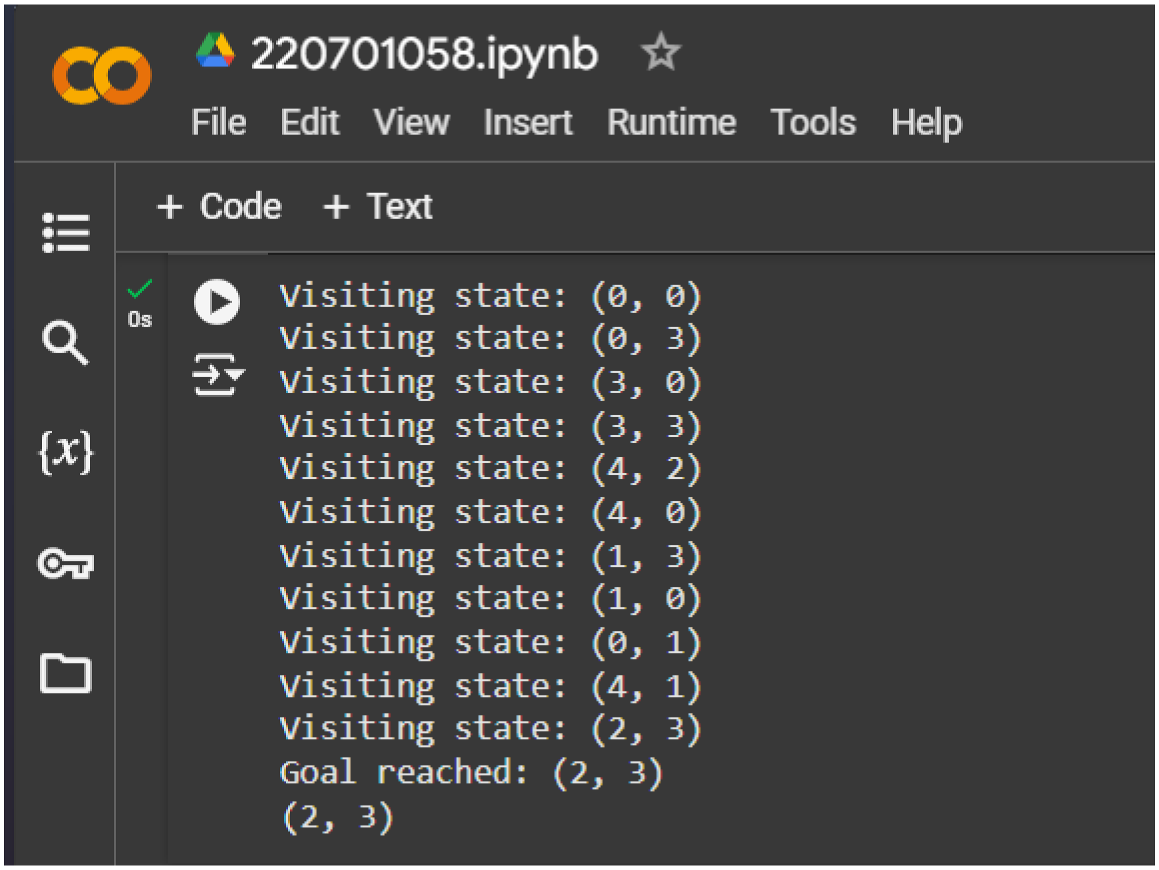
CODE:

```
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:



```
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)
```

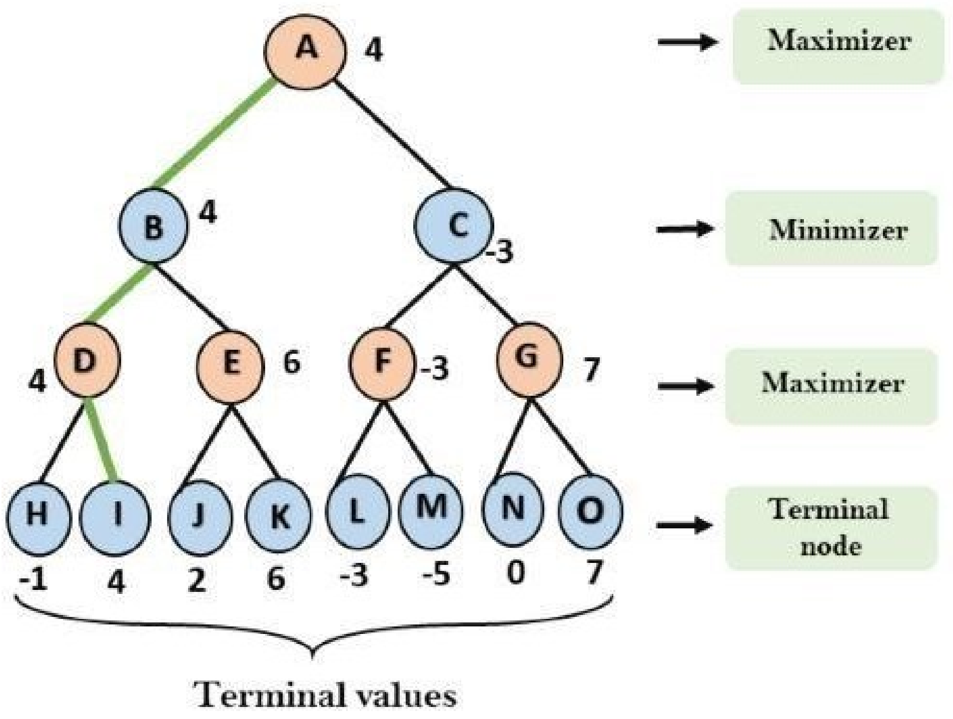
RESULT:

The code for Water jug problem has been executed successfully.

EX.NO: 4 **DATE:20-09-24**

MINIMAX ALGORITHM

- 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.

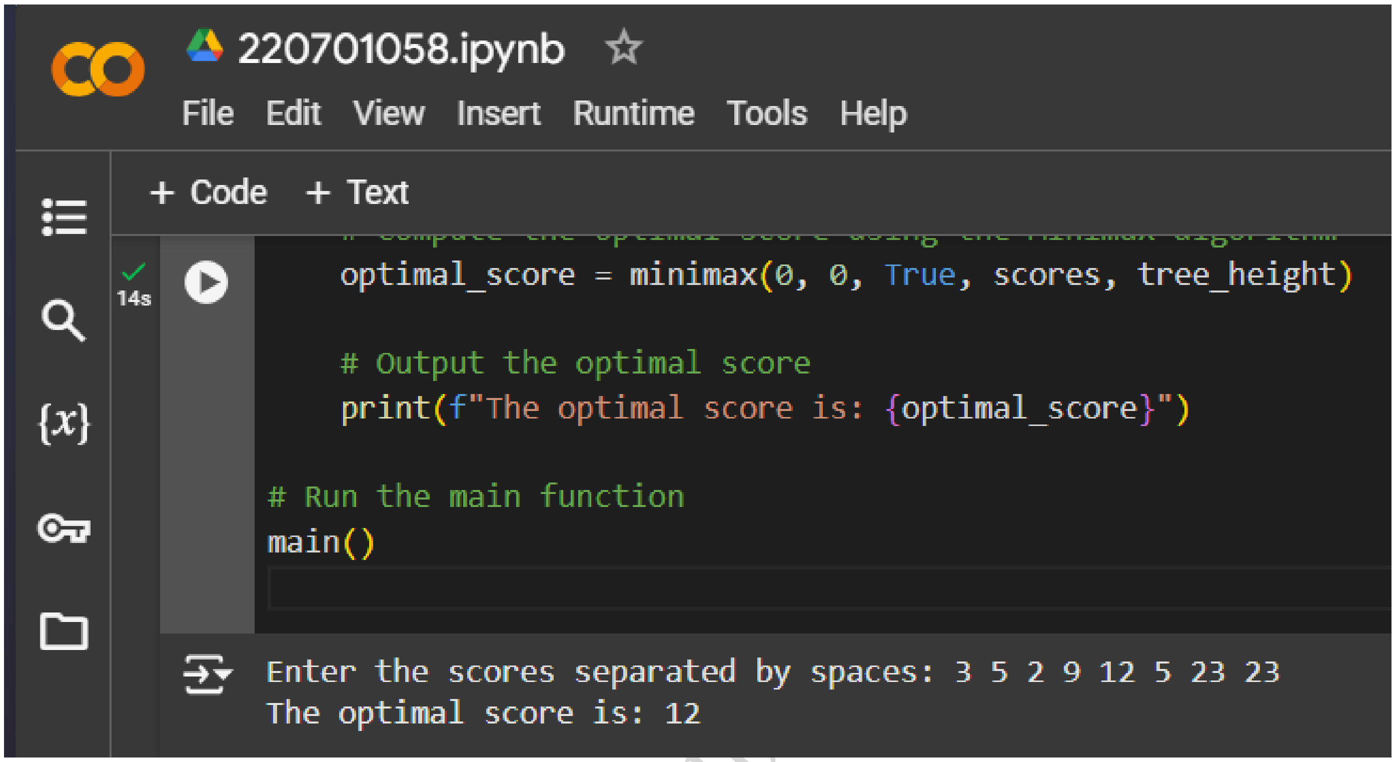


CODE:

```
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:



```
    optimal_score = minimax(0, 0, True, scores, tree_height)

    # Output the optimal score
    print(f"The optimal score is: {optimal_score}")

# Run the main function
main()
```

```
Enter the scores separated by spaces: 3 5 2 9 12 5 23 23
The optimal score is: 12
```

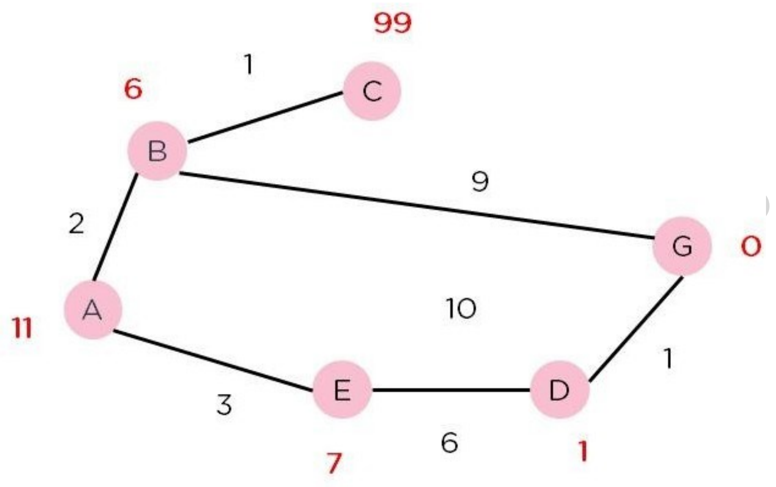
RESULT:

The code for DFS has successfully been executed.

EX.No : 5 **DATE : 27-09-24**

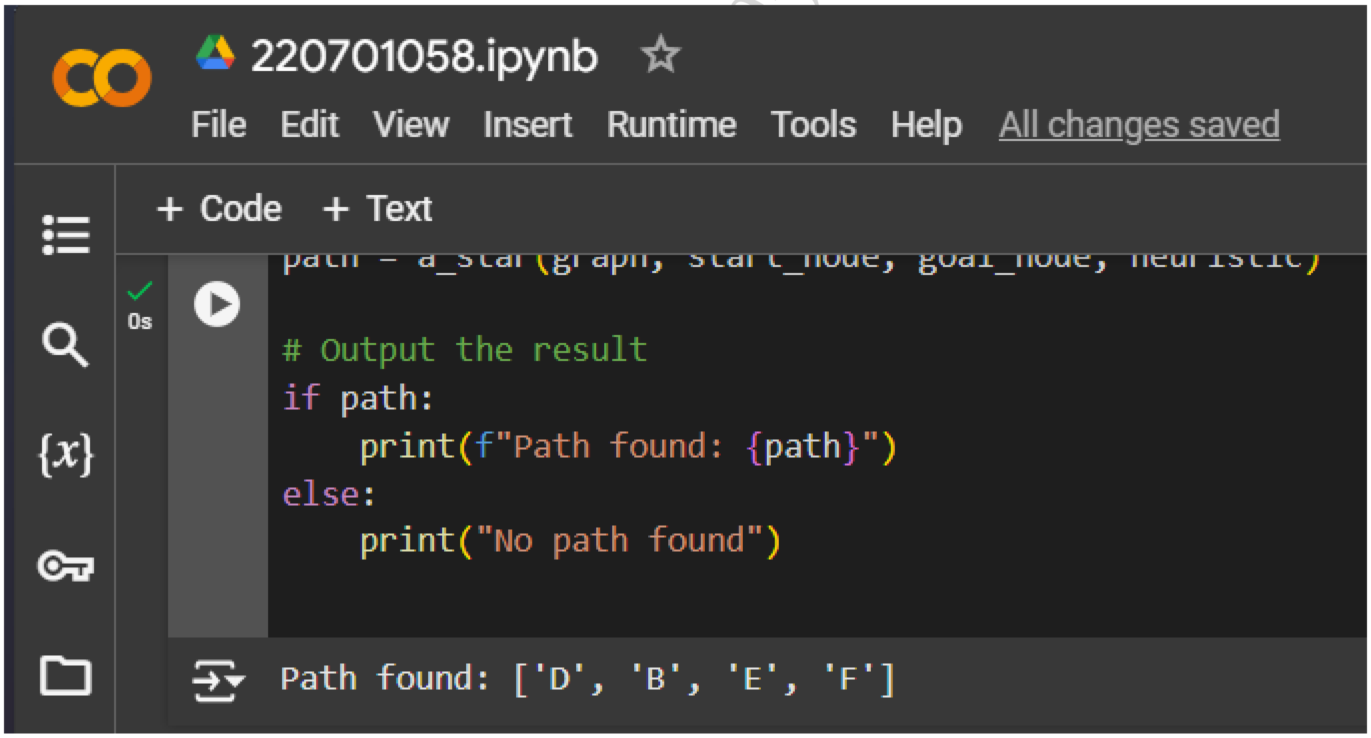
A* SEARCH ALGORITHM

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.



CODE:

```
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_func):
    open_list = []
    heapq.heappush(open_list, Node(start, None, 0, h_func(start, goal)))
    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[current_node.name]:
            if neighbor in closed_list:
                continue

            g_new = current_node.g + cost
            h_new = h_func(neighbor, goal)
            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 = {
    'A': [('D', 1), ('C', 3)],
    'B': [('A', 1), ('D', 1), ('E', 3)],
    'C': [('A', 3), ('F', 2)],
    'D': [('B', 1)],
    'E': [('B', 3), ('F', 1)],
    'F': [('C', 2), ('E', 1)]
}

def heuristic(node, goal):
    return 0


start_node = 'D'
goal_node = 'F'
path = a_star(graph, start_node, goal_node, heuristic)


if path:
    print(f"Path found: {path}")
else:
    print("No path found")
```



```
path = a_star(graph, start_node, goal_node, heuristic)

# Output the result
if path:
    print(f"Path found: {path}")
else:
    print("No path found")
```

```
Path found: ['D', 'B', 'E', 'F']
```

RESULT:

The code for A* algorithm has successfully executed.

EX.NO :6 **DATE: 27-09-24**

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
   ```

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 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)
   ```

4. Facts: -

   A fact is a predicate followed by a dot.

   Ex:

   ```
   bigger_animal(whale).
   life_is_beautiful.
   ```

5. Rules: -

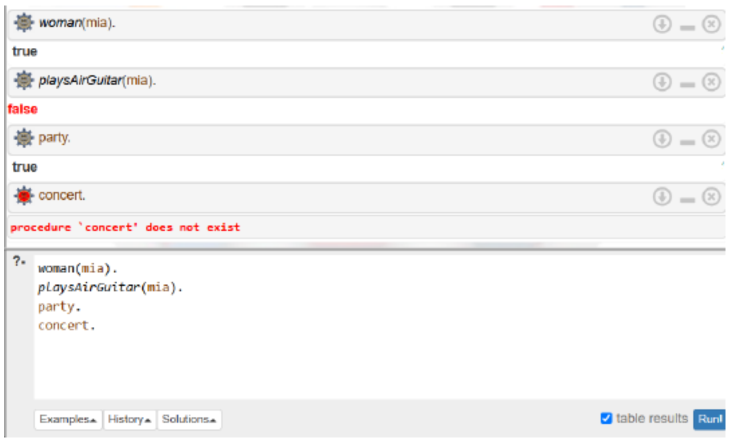
   A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).

   Ex:

   ```
   is_smaller(X,Y):-is_bigger(Y,X).
   aunt(Aunt,Child):-sister(Aunt,Parent),parent(Parent,Child).
   ```

SOURCE CODE:

KB1:

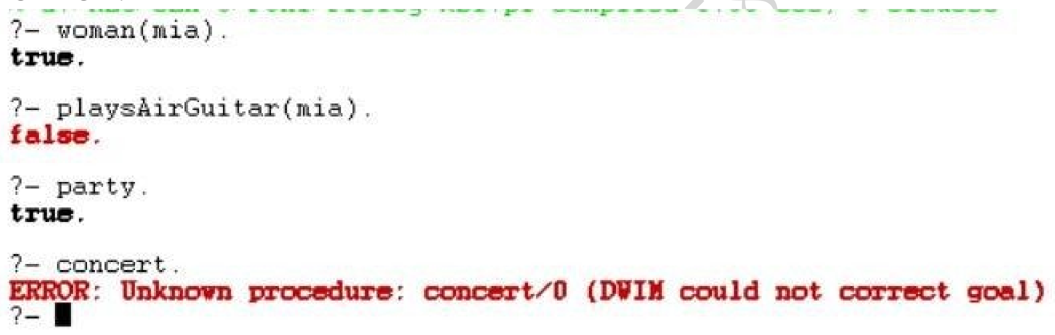


```
woman(mia).
true
playsAirGuitar(mia).
false
party.
true
concert.
procedure `concert' does not exist
```

```
?- woman(mia).
playsAirGuitar(mia).
party.
concert.
```

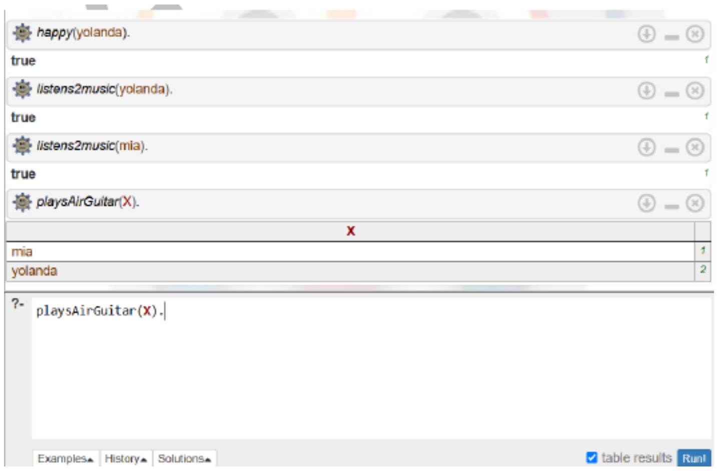
woman(mia). woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
Query 1: ?-woman(mia).
Query 2: ?-playsAirGuitar(mia).
Query 3: ?-party.
Query 4: ?-concert.

OUTPUT: -

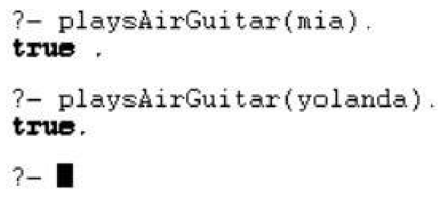


```
?- woman(mia).
true.

?- playsAirGuitar(mia).
false.

?- party.
true.

?- concert.
ERROR: Unknown procedure: concert/0 (DWIM could not correct goal)
?- █
```

KB2:
happy(yolanda). listens2music(mia).
Listens2music(yolanda):-happy(yolanda). playsAirGuitar(mia):-listens2music(mia).
playsAirGuitar(Yolanda):-listens2music(yolanda).



```
happy(yolanda).
true
listens2music(yolanda).
true
listens2music(mia).
true
playsAirGuitar(X).
X
mia        1
yolanda    2
?- playsAirGuitar(X).
```

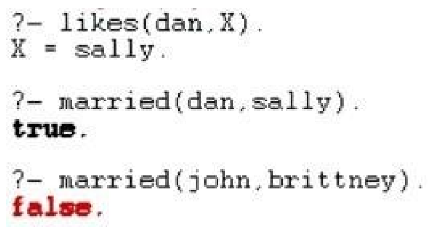


```
?- playsAirGuitar(mia).
true .

?- playsAirGuitar(yolanda).
true.

?-
```

OUTPUT: -

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).

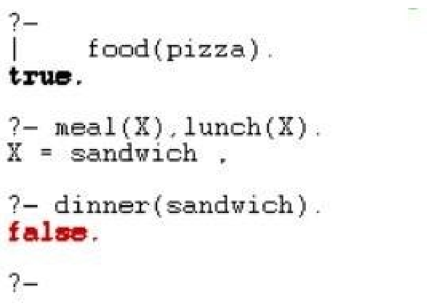
OUTPUT: -



```
?- likes(dan,X).
X = sally.

?- married(dan,sally).
true.

?- married(john,brittney).
false.
```

KB4: food(burger).
food(sandwich).
food(pizza).
lunch(sandwich).
dinner(pizza).
meal(X):-food(X).

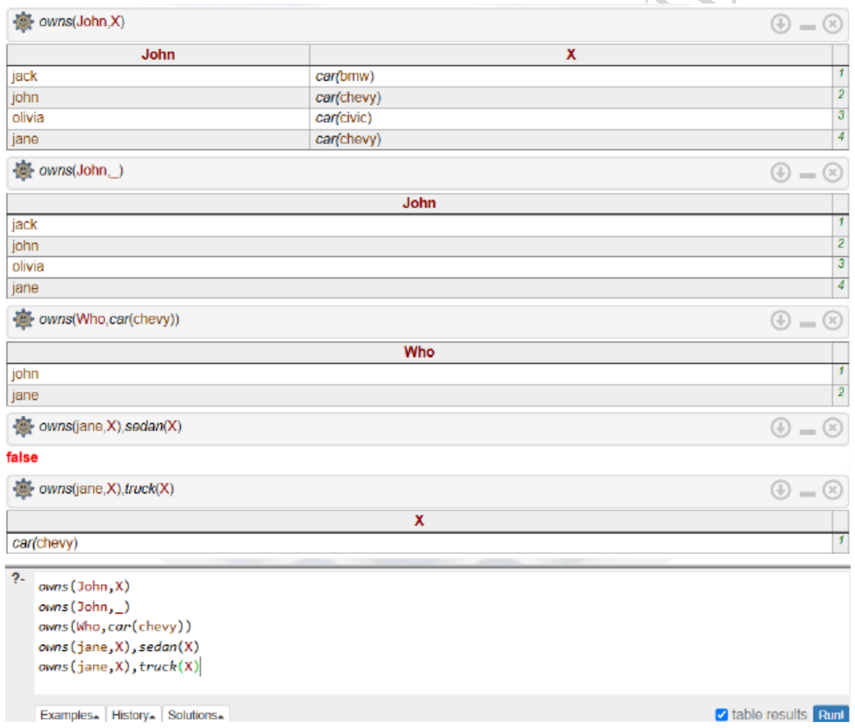
OUTPUT:



```
?-
|    food(pizza).
true.

?- meal(X),lunch(X).
X = sandwich ,

?- dinner(sandwich).
false.

?-
```

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)).



owns(John,X)

John	X	
jack	car(bmw)	1
john	car(chevy)	2
olivia	car(civic)	3
jane	car(chevy)	4

owns(John,_)

John	
jack	1
john	2
olivia	3
jane	4

owns(Who,car(chevy))

Who	
john	1
jane	2

owns(jane,X),sedan(X)

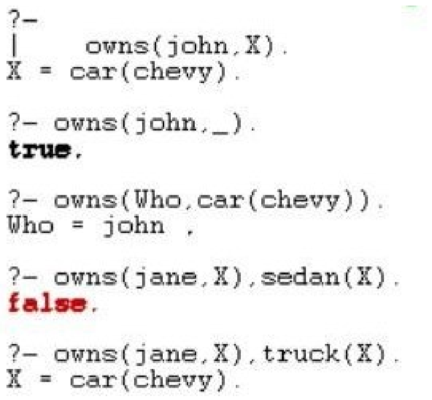
false

owns(jane,X),truck(X)

X	
car(chevy)	1

```
?- owns(John,X)
   owns(John,_)
   owns(Who,car(chevy))
   owns(jane,X),sedan(X)
   owns(jane,X),truck(X)
```

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:

The prolog has been successfully executed and verified.

EX.NO :7 **DATE: 07-10-24**

PROLOG- FAMILY TREE

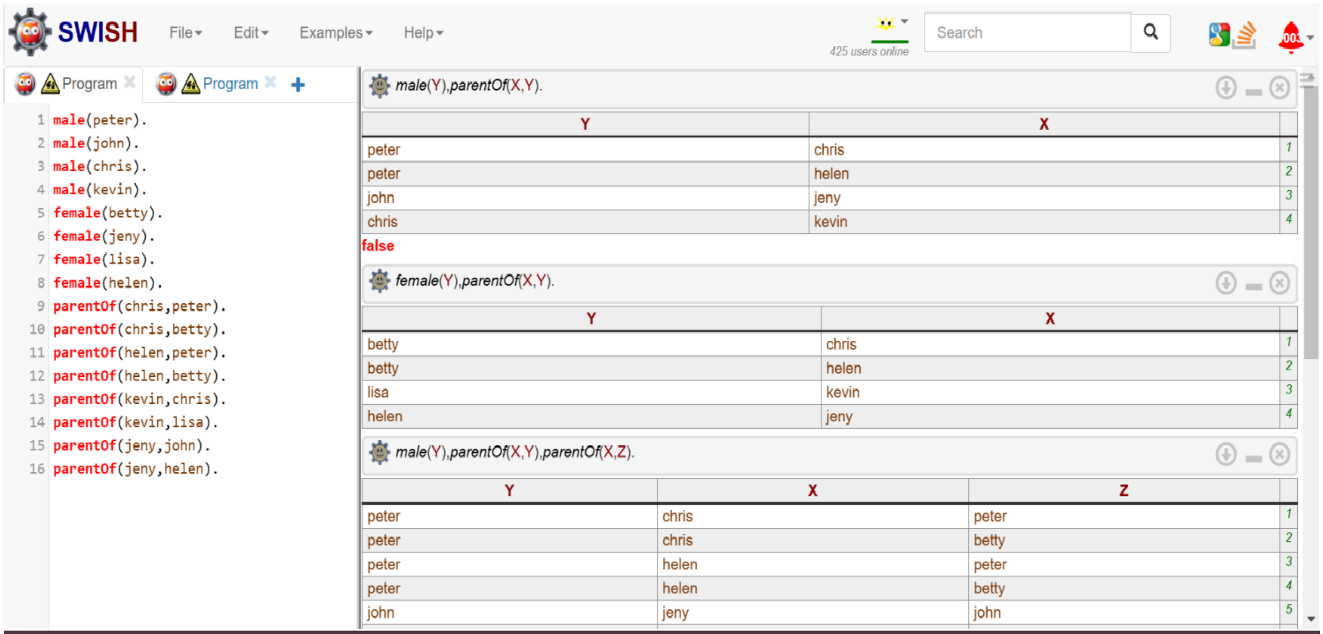
AIM :

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:



```
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).
```

male(Y),parentOf(X,Y).

Y	X	
peter	chris	1
peter	helen	2
john	jeny	3
chris	kevin	4

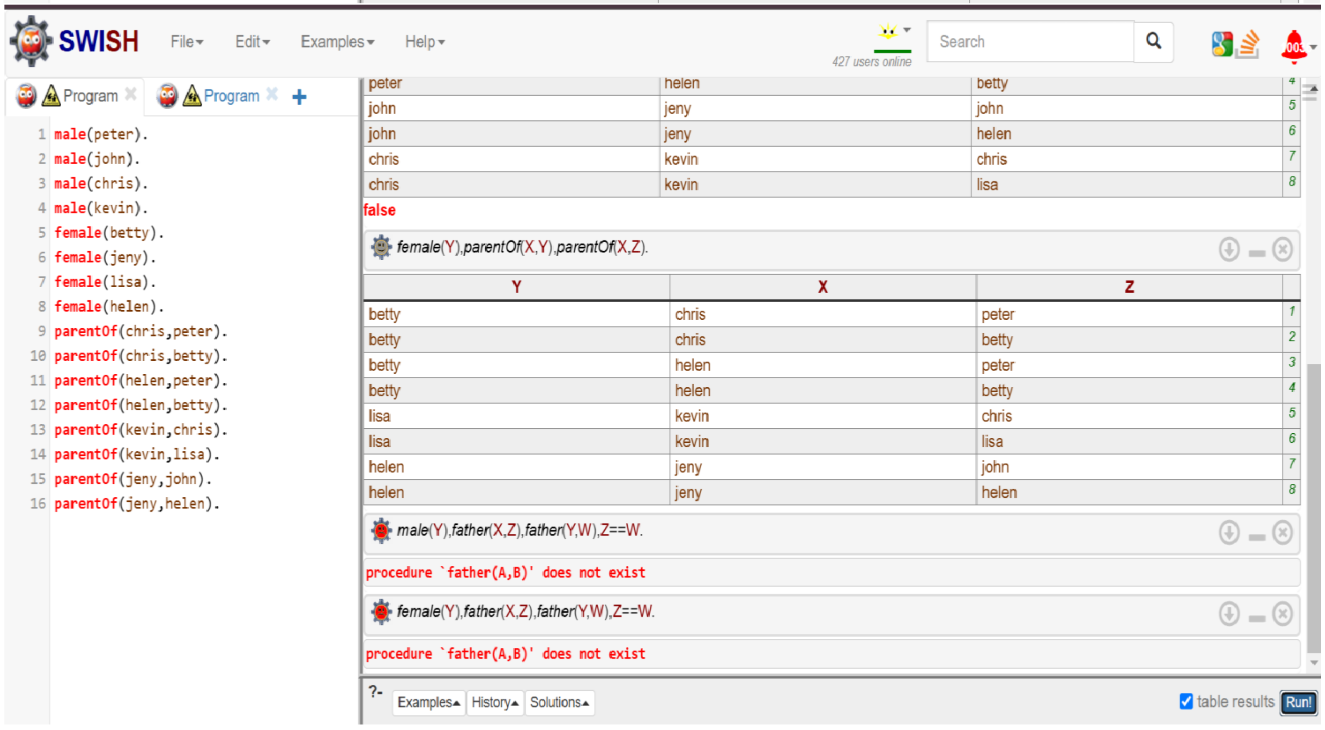
false

female(Y),parentOf(X,Y).

Y	X	
betty	chris	1
betty	helen	2
lisa	kevin	3
helen	jeny	4

male(Y),parentOf(X,Y),parentOf(X,Z).

Y	X	Z	
peter	chris	peter	1
peter	chris	betty	2
peter	helen	peter	3
peter	helen	betty	4
john	jeny	john	5



```
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).
```

peter	helen	betty	4
john	jeny	john	5
john	jeny	helen	6
chris	kevin	chris	7
chris	kevin	lisa	8

false

female(Y),parentOf(X,Y),parentOf(X,Z).

Y	X	Z	
betty	chris	peter	1
betty	chris	betty	2
betty	helen	peter	3
betty	helen	betty	4
lisa	kevin	chris	5
lisa	kevin	lisa	6
helen	jeny	john	7
helen	jeny	helen	8

male(Y),father(X,Z),father(Y,W),Z==W.

```
procedure `father(A,B)' does not exist
```

female(Y),father(X,Z),father(Y,W),Z==W.

```
procedure `father(A,B)' does not exist
```

RESULT:

The prolog family-tree has been implemented successfully and verified.

EX.NO : 8 **DATE :** 18-11-24

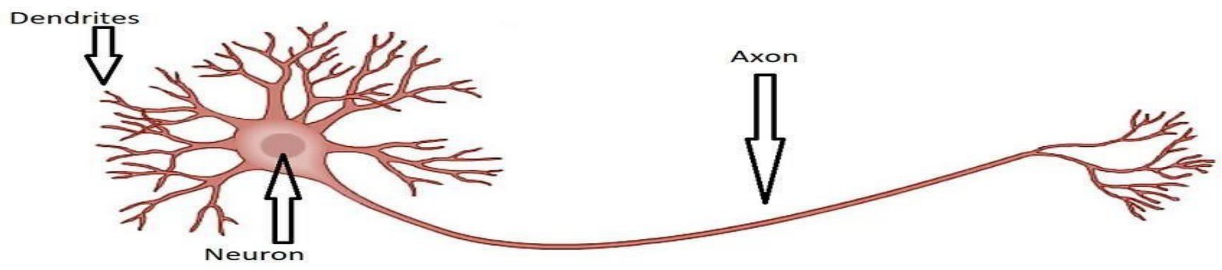
IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - CLASSIFICATION AIM

:

To implementing artificial neural networks for an application in classification using python.

What is an Artificial Neural Network?

Artificial Neural Network is much similar to the human brain. The human Brain consist of **neurons**. These neurons are connected. In the human brain, neuron looks something like this…

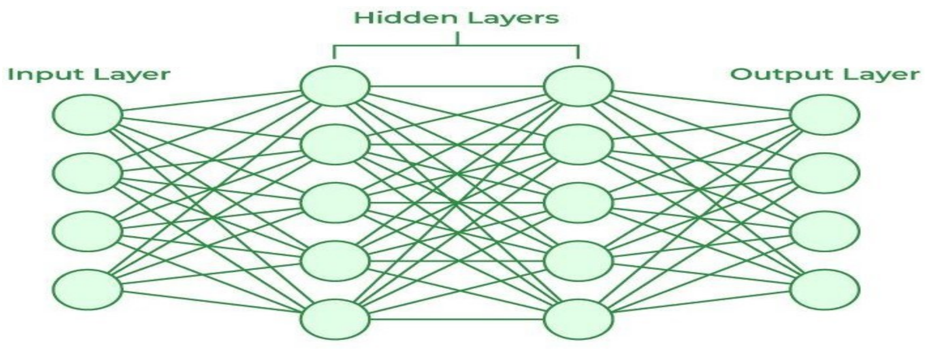


As you can see in this image, There are **neurons, Dendrites, and axons.**
What do you think?
When you touch the hot surface, how you suddenly remove your hand?. This is the procedure that happens inside you.When you touch some hot surface. Then automatically your skin sends a signal to the neuron. And then the neuron takes a decision, **"Remove your hand"**. So that's all about the **Human Brain.** In the same way, **Artificial Neural Network works.**

Artificial Neural Networks

Artificial Neural Networks contain artificial neurons which are called **units**. These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers. The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks to input data provided.

The structures and operations of human neurons serve as the basis for artificial neural networks. It is also known as neural networks or neural nets. The input layer of an artificial neural network is the first layer, and it receives input from external sources and releases it to the hidden layer, which is the second layer. In the hidden layer, each neuron receives input from the previous layer neurons, computes the weighted sum, and sends it to the neurons in the next layer. These connections are weighted means effects of the inputs from the previous layer are optimized more or less by assigning different-different weights to each input and it is adjusted during the training process by optimizing these weights for improved model performance.

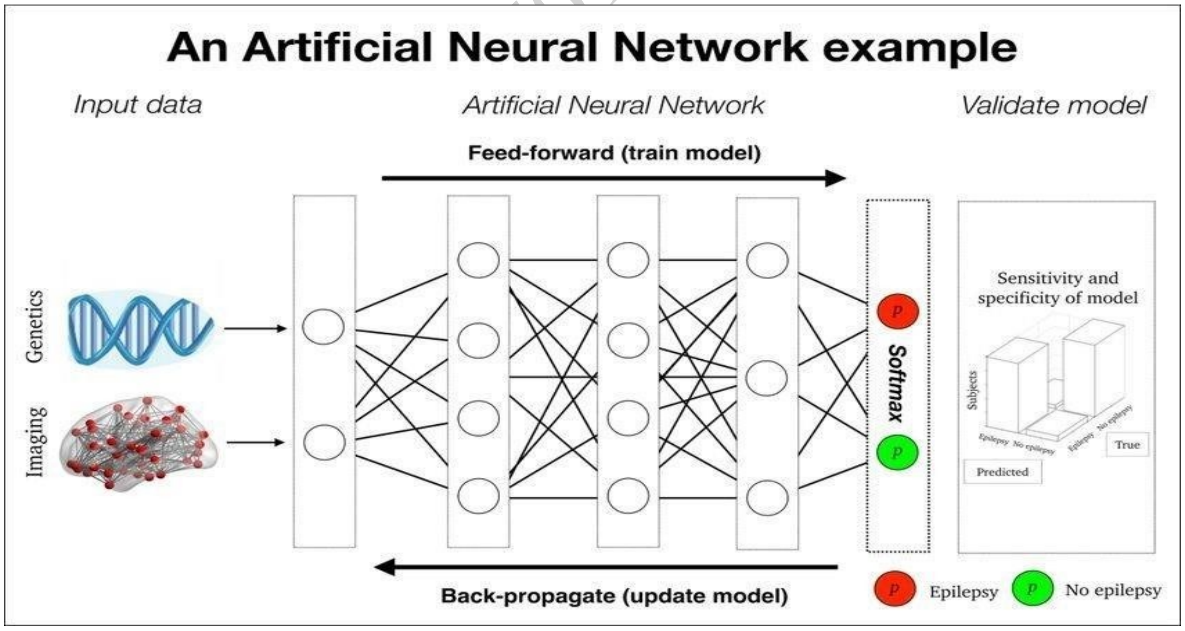


What are the types of Artificial Neural Networks?

1. Feedforward Neural Network
2. Convolutional Neural Network
3. Modular Neural Network
4. Radial basis function Neural Network
5. Recurrent Neural Network

Applications of Artificial Neural Networks

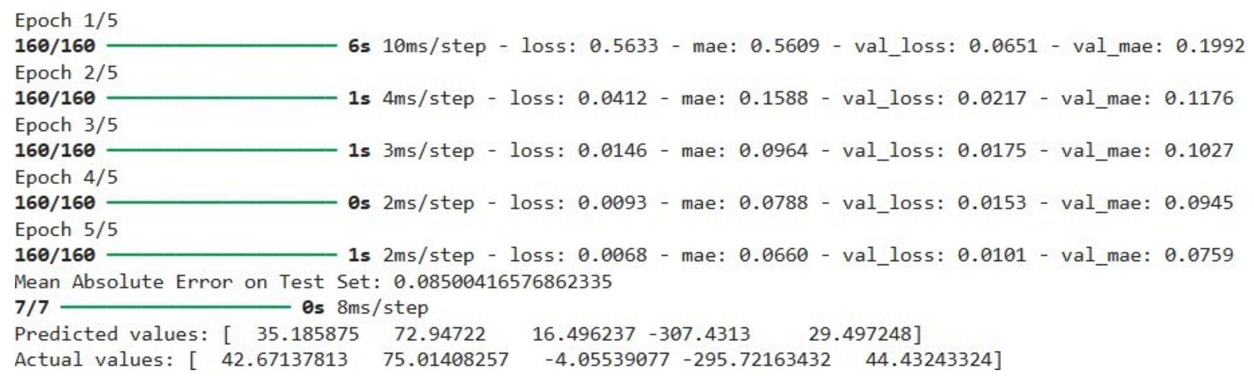
1. Social Media
2. Marketing and Sales
3. Healthcare
4. Personal Assistants


An Artificial Neural Network example
Input data
Artificial Neural Network
Validate model
Feed-forward (train model)
Genetics
Imaging
Softmax
Sensitivity and specificity of model
Subjects
Epilepsy
No epilepsy
True
Predicted
Back-propagate (update model)
Epilepsy
No epilepsy

CODE:

```
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



```
Epoch 1/5
160/160 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - loss: 0.5633 - mae: 0.5609 - val_loss: 0.0651 - val_mae: 0.1992
Epoch 2/5
160/160 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 0.0412 - mae: 0.1588 - val_loss: 0.0217 - val_mae: 0.1176
Epoch 3/5
160/160 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0146 - mae: 0.0964 - val_loss: 0.0175 - val_mae: 0.1027
Epoch 4/5
160/160 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0093 - mae: 0.0788 - val_loss: 0.0153 - val_mae: 0.0945
Epoch 5/5
160/160 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.0068 - mae: 0.0660 - val_loss: 0.0101 - val_mae: 0.0759
Mean Absolute Error on Test Set: 0.08500416576862335
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step
Predicted values: [  35.185875   72.94722    16.496237 -307.4313     29.497248]
Actual values: [  42.67137813   75.01408257   -4.05539077 -295.72163432   44.43243324]
```

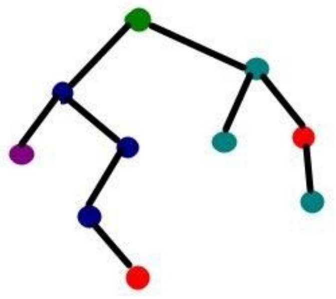
RESULT:

The Code for ANN is successfully executed and verified.

EX.NO :9 **DATE : 25-10-24**

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.

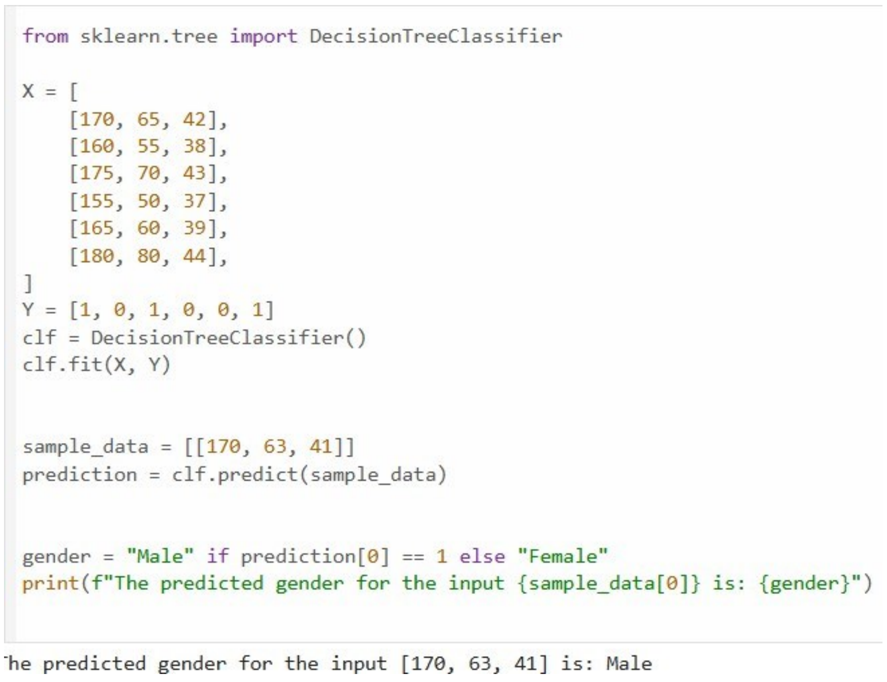
EXPLANATION:

- 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.

CODE:

```
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:



```
from sklearn.tree import DecisionTreeClassifier

X = [
    [170, 65, 42],
    [160, 55, 38],
    [175, 70, 43],
    [155, 50, 37],
    [165, 60, 39],
    [180, 80, 44],
]
Y = [1, 0, 1, 0, 0, 1]
clf = DecisionTreeClassifier()
clf.fit(X, Y)


sample_data = [[170, 63, 41]]
prediction = clf.predict(sample_data)


gender = "Male" if prediction[0] == 1 else "Female"
print(f"The predicted gender for the input {sample_data[0]} is: {gender}")
```

```
he predicted gender for the input [170, 63, 41] is: Male
```

RESULT:

The decision tree classification technique for gender classification using python has been executed successfully.

EX NO :10 **DATE : 1-11-24**

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:

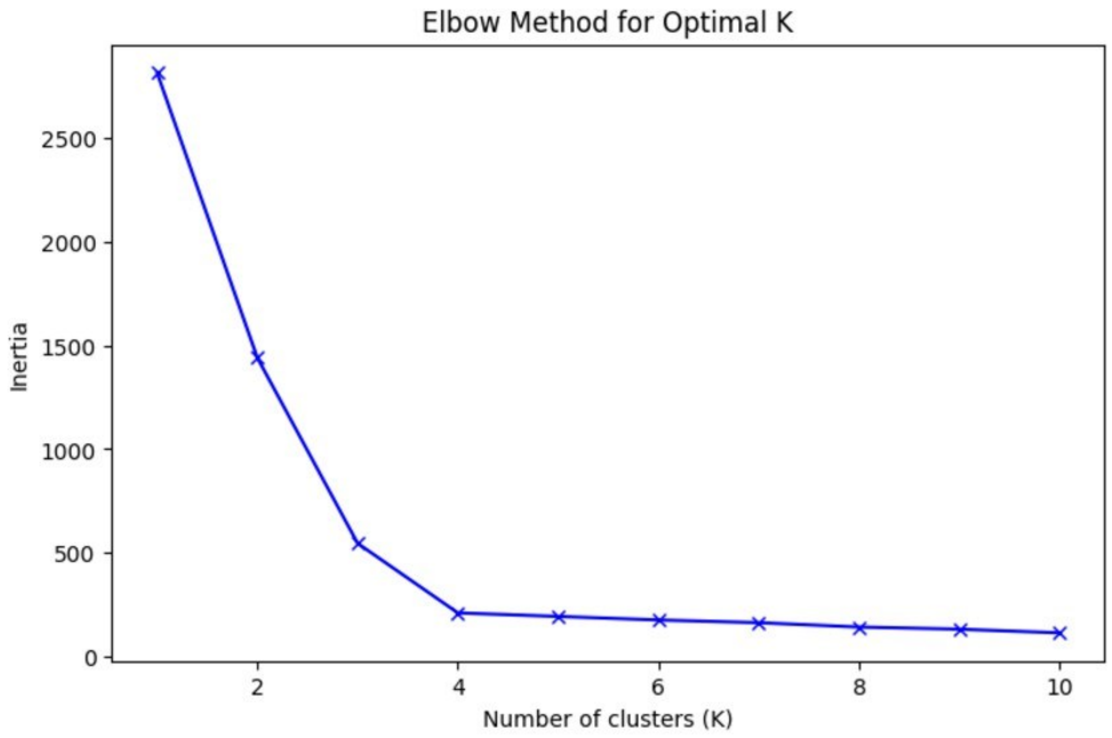
- Import KMeans from sklearn.cluster
- Assign X and Y.
- Call the function KMeans().
- Perform scatter operation and display the output.

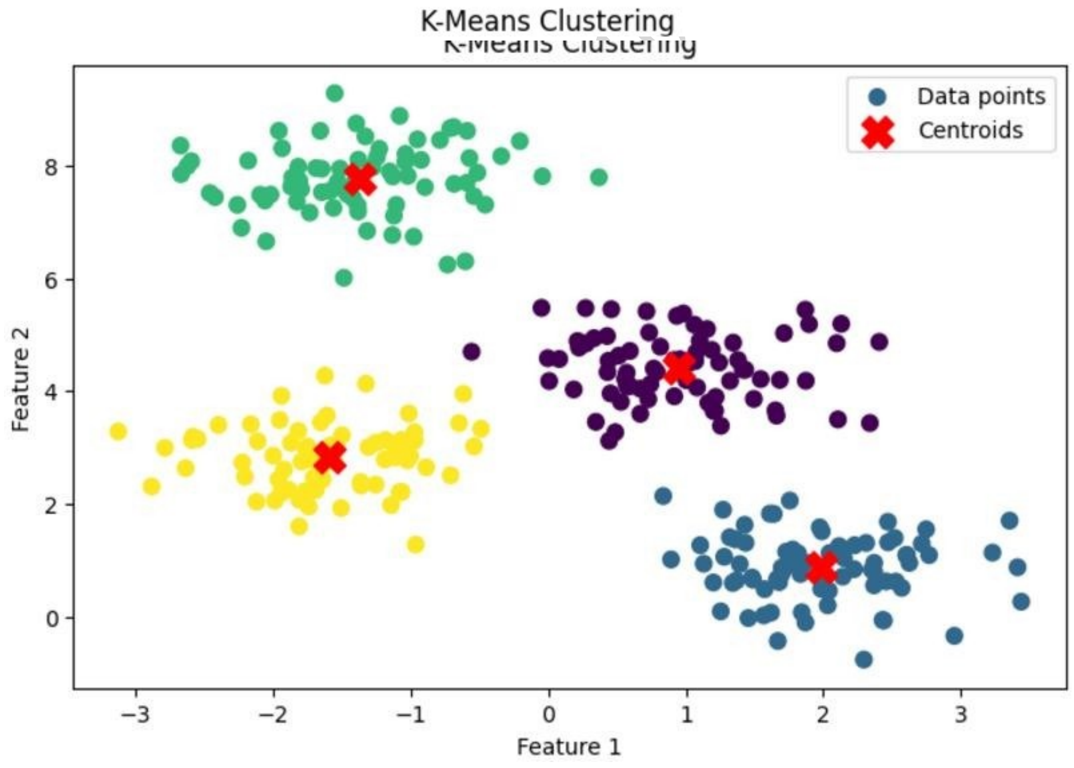
:

CODE:

```
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.