Principles of Artificial Intelligence

Lab Report

Course Code:19CSE451

Name: Geet Tej Mahesh M

Register Number: CH.EN.U4CSE21041

Section: CSE-A

List of Experiments

1) 8- Queens

Aim:

To implement the 8 Queens problem solution using backtracking and assess its efficiency in finding all possible solutions.

Algorithm:

1. Initialize an empty chessboard.

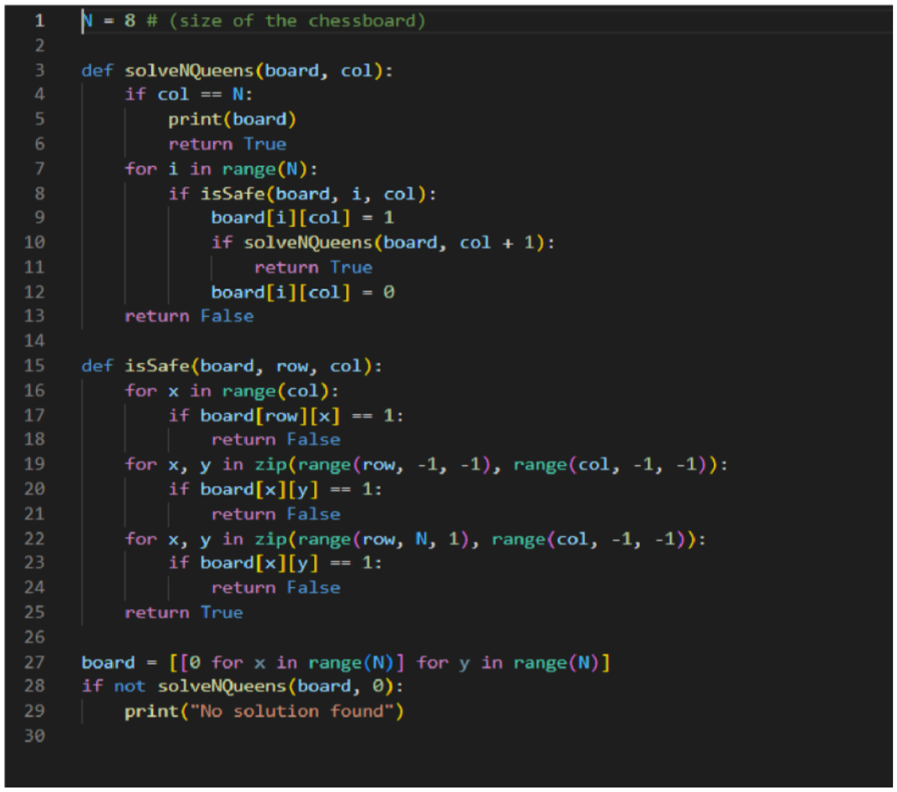
2. Place a queen in the first row and column.

3. Recursively place queens in the next row, ensuring that no two queens attack each other.

4. If all queens are placed successfully, print the solution.

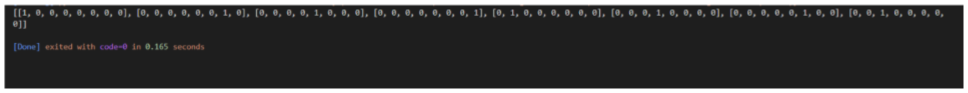
5. Backtrack if a solution is not possible.

Code:



```python
N = 8 # (size of the chessboard)

def solveNQueens(board, col):
    if col == N:
        print(board)
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQueens(board, col + 1):
                return True
            board[i][col] = 0
    return False

def isSafe(board, row, col):
    for x in range(col):
        if board[row][x] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    return True

board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
    print("No solution found")
```

Output:



```
[[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]]

[Done] exited with code=0 in 0.165 seconds
```

Result:

The program generates and prints all possible solutions to the 8 Queens problem.

Inference:

The backtracking algorithm efficiently finds all possible solutions to the 8 Queens problem without conflicts.

2) Depth First Search

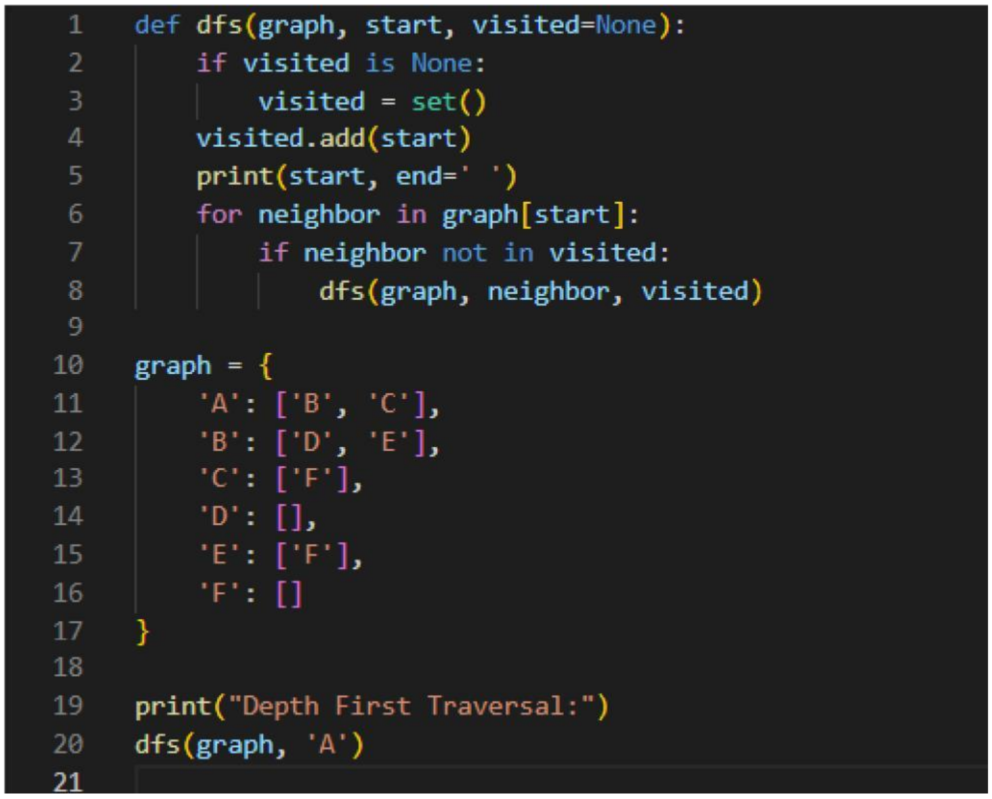
Aim:

To implement the Depth First Search (DFS) algorithm for traversing a graph and understand its application in graph traversal.

Algorithm:

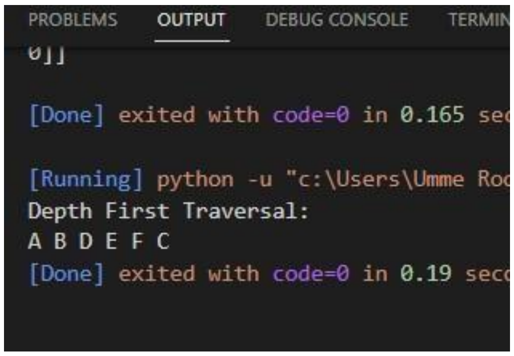
1. Choose a starting vertex and mark it as visited.
2. Explore each adjacent vertex of the current vertex recursively.
3. If an unvisited adjacent vertex is found, repeat step 2.
4. If all adjacent vertices are visited, backtrack to the previous vertex and explore its unvisited neighbours.
5. Repeat steps 2-4 until all vertices are visited.

Code:



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

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("Depth First Traversal:")
dfs(graph, 'A')
```

Output:



```
0]]

[Done] exited with code=0 in 0.165 sec

[Running] python -u "c:\Users\Umme Roc
Depth First Traversal:
A B D E F C
[Done] exited with code=0 in 0.19 seco
```

Result:

Depth First Traversal: A B D E F C

Inference:

The Depth First Search algorithm effectively traverses through the graph, visiting each node and its neighbours in a depth-first manner.

3) Depth First Search for Water Jug Problem

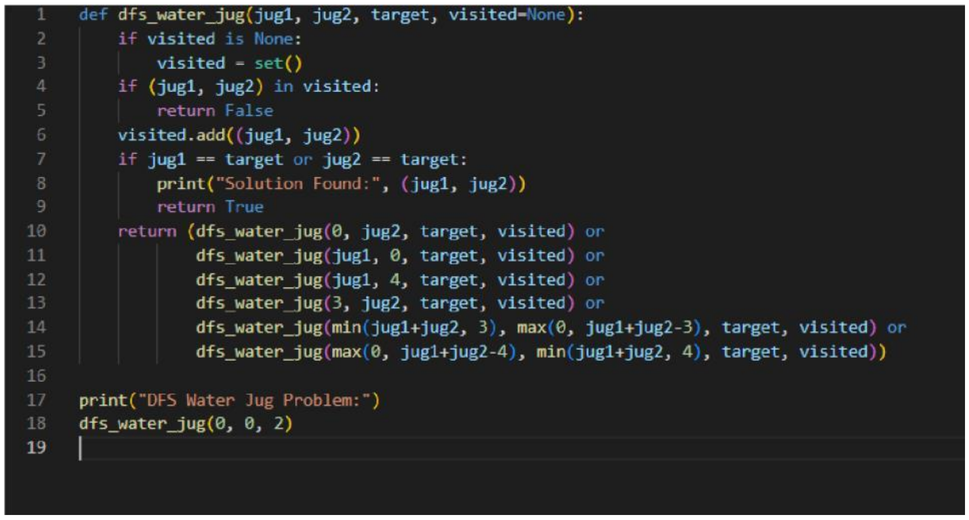
Aim:

To solve the Water Jug Problem using Depth First Search (DFS) and observe the search process for finding the solution.

Algorithm:

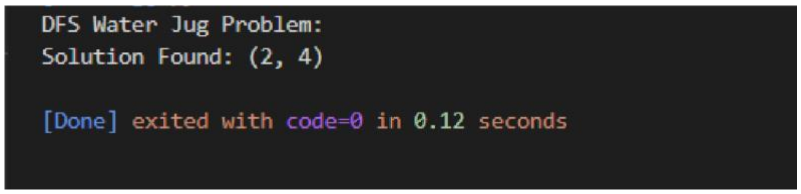
1. Define the initial states of the jugs.
2. Define the operations that can be performed on the jugs (pouring, filling, emptying).
3. Use Depth First Search to explore all possible states of the jugs by applying the defined operations.
4. Keep track of visited states to avoid revisiting them.
5. Stop the search when the desired state is reached or all possible states are explored.

Code:



```python
def dfs_water_jug(jug1, jug2, target, visited=None):
    if visited is None:
        visited = set()
    if (jug1, jug2) in visited:
        return False
    visited.add((jug1, jug2))
    if jug1 == target or jug2 == target:
        print("Solution Found:", (jug1, jug2))
        return True
    return (dfs_water_jug(0, jug2, target, visited) or
            dfs_water_jug(jug1, 0, target, visited) or
            dfs_water_jug(jug1, 4, target, visited) or
            dfs_water_jug(3, jug2, target, visited) or
            dfs_water_jug(min(jug1+jug2, 3), max(0, jug1+jug2-3), target, visited) or
            dfs_water_jug(max(0, jug1+jug2-4), min(jug1+jug2, 4), target, visited))

print("DFS Water Jug Problem:")
dfs_water_jug(0, 0, 2)
```

Output:


```
DFS Water Jug Problem:
Solution Found: (2, 4)

[Done] exited with code=0 in 0.12 seconds
```

Result:

Solution Found: (2, 4)

Inference:

The Depth First Search algorithm successfully finds a solution to the Water Jug Problem, demonstrating its effectiveness in exploring state space problems.

4) Min Max Algorithm

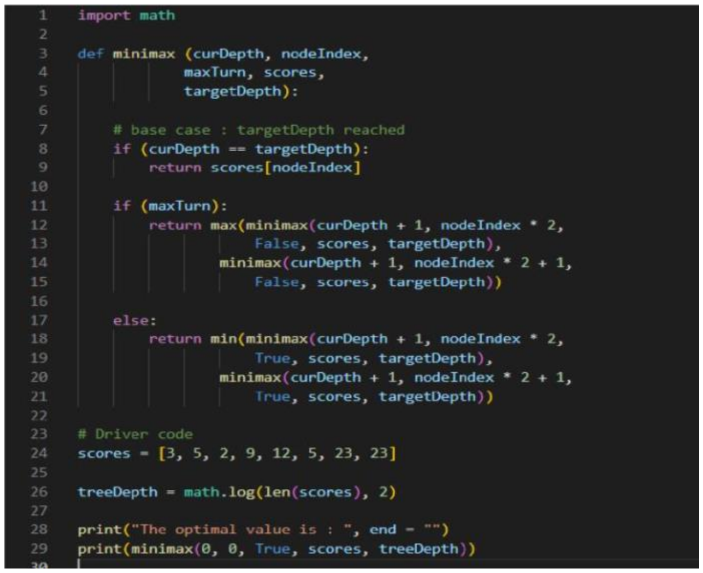
Aim:

To implement the Min Max algorithm for decision-making in zero-sum games, such as Tic-Tac Toe, and understand its application in game theory.

Algorithm:

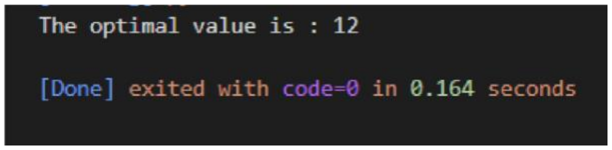
1. Define the game state representation and rules.
2. Implement the Min Max algorithm, which alternates between maximizing and minimizing players.
3. Recursively explore the game tree by considering all possible moves.
4. Assign scores to terminal states (win, lose, draw).
5. Backtrack to previous states while computing the optimal move.

Code:



```python
import math

def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

Output:



```
The optimal value is : 12

[Done] exited with code=0 in 0.164 seconds
```

Result:

The Min Max algorithm computes the best move for a given game state by recursively exploring the game tree and maximizing the player's chances of winning while minimizing the opponent's chances.

Inference:

The Min Max algorithm provides an effective strategy for decision-making in zero-sum games, ensuring optimal moves based on the current game state.

5) ANN for an application using python classification

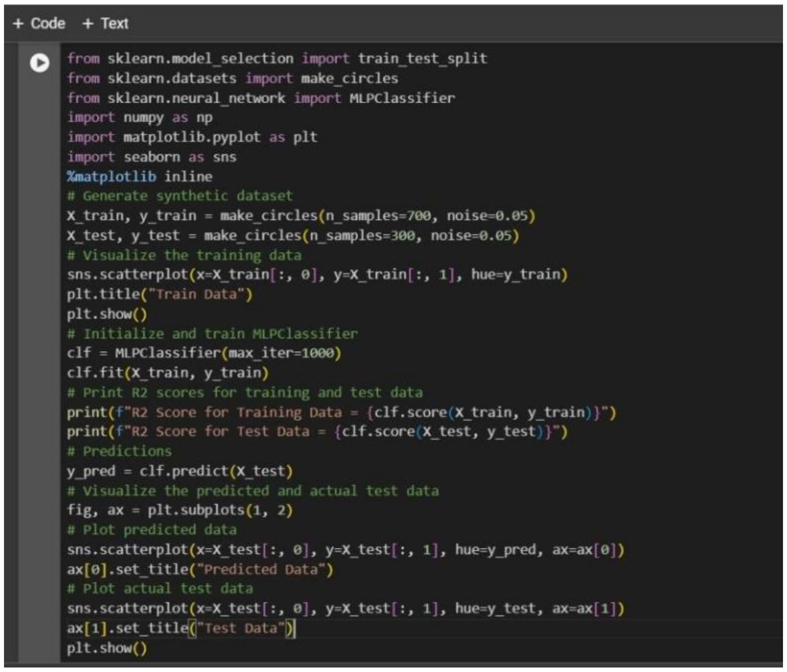
Aim:

To implement an artificial neural network (ANN) for a classification task using Python and demonstrate its ability to learn complex patterns in data.

Algorithm:

1. Data Preparation: Generate or load dataset, split into training and testing sets.
2. Neural Network Initialization: Initialize MLP classifier with max iterations.
3. Training: Train MLP classifier with training data.
4. Evaluation: Evaluate trained classifier on both training and testing datasets, calculate R2 scores.
5. Prediction: Use trained classifier for predictions on the testing dataset.
6. Visualization (Optional): Visualize predicted and actual test data points for performance observation.

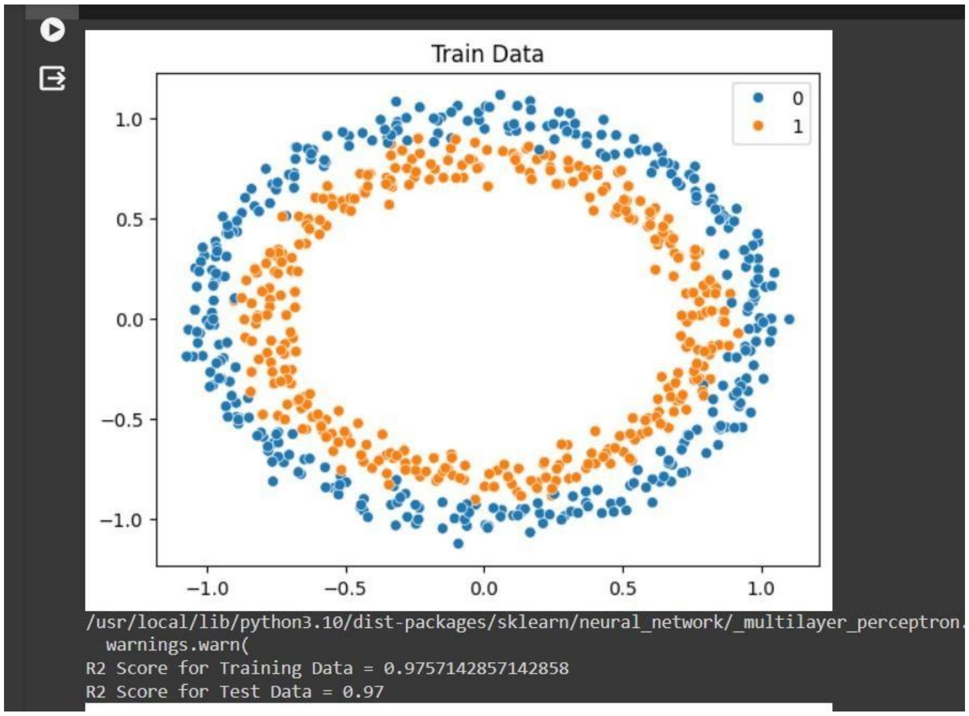
Code:



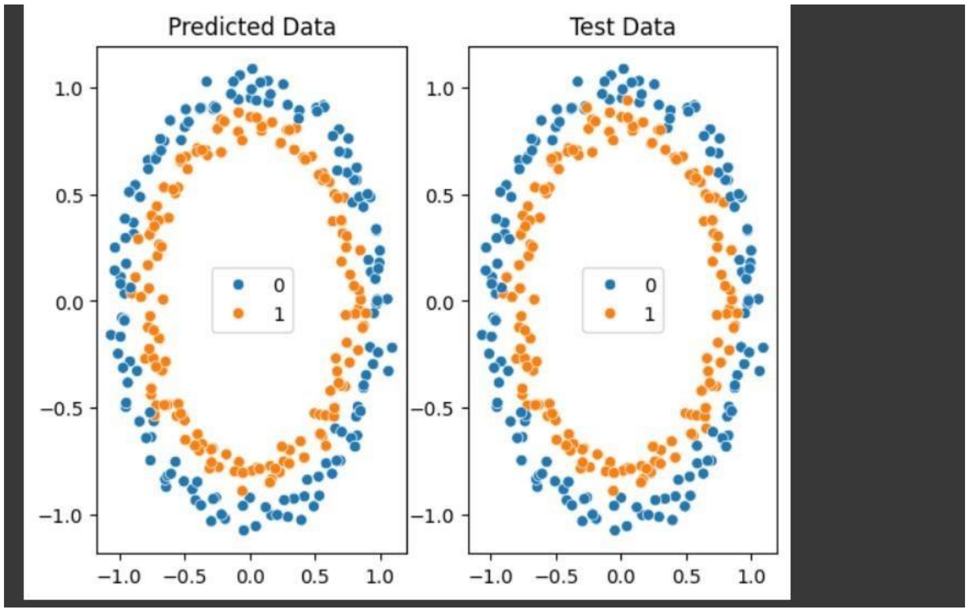
```python
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_circles
from sklearn.neural_network import MLPClassifier
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
# Generate synthetic dataset
X_train, y_train = make_circles(n_samples=700, noise=0.05)
X_test, y_test = make_circles(n_samples=300, noise=0.05)
# Visualize the training data
sns.scatterplot(x=X_train[:, 0], y=X_train[:, 1], hue=y_train)
plt.title("Train Data")
plt.show()
# Initialize and train MLPClassifier
clf = MLPClassifier(max_iter=1000)
clf.fit(X_train, y_train)
# Print R2 scores for training and test data
print(f"R2 Score for Training Data = {clf.score(X_train, y_train)}")
print(f"R2 Score for Test Data = {clf.score(X_test, y_test)}")
# Predictions
y_pred = clf.predict(X_test)
# Visualize the predicted and actual test data
fig, ax = plt.subplots(1, 2)
# Plot predicted data
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_pred, ax=ax[0])
ax[0].set_title("Predicted Data")
# Plot actual test data
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, ax=ax[1])
ax[1].set_title("Test Data")
plt.show()
```

`

Output:



```
/usr/local/lib/python3.10/dist-packages/sklearn/neural_network/_multilayer_perceptron.
  warnings.warn(
R2 Score for Training Data = 0.9757142857142858
R2 Score for Test Data = 0.97
```



Result:

The neural network is trained and evaluated on the classification task, and the accuracy of the model is printed.

Inference:

Artificial neural networks can be effectively implemented using Python libraries like NumPy for classification tasks, achieving high accuracy in predicting outcomes based on input data.

6) ANN for an application using Python Regression

Aim:

To implement an artificial neural network (ANN) for a regression task using Python and demonstrate its ability to predict continuous values.

Algorithm:

1. Import Libraries: Import necessary libraries including `MLPRegressor` from `sklearn.neural_network`, `train_test_split` from `sklearn.model_selection`, `make_regression` from `sklearn.datasets`, `numpy` as `np`, `matplotlib.pyplot` as `plt`, and `seaborn` as `sns`.

2. Generate Dataset: Create a synthetic dataset using `make_regression` with specified parameters such as number of samples, noise, and number of features.

3. Data Preparation: Split the dataset into training and testing sets using `train_test_split`, specifying the test size, shuffling, and random state.

4. Neural Network Initialization: Initialize an MLPRegressor with a maximum number of iterations.

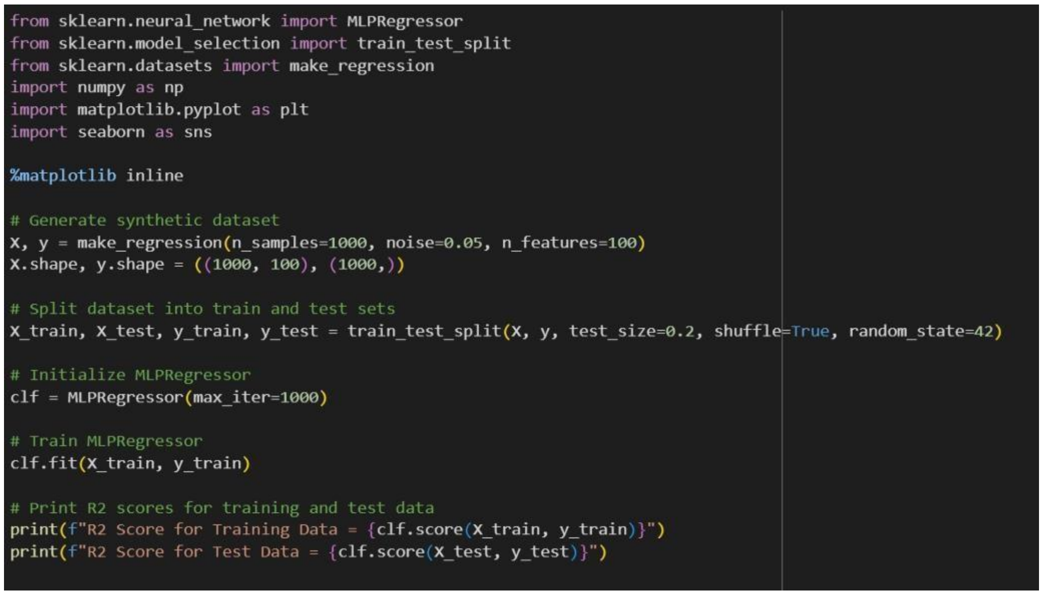
5. Training: Train the MLPRegressor using the training data.

6. Evaluation: Evaluate the trained model on both the training and testing datasets. Calculate and print the R2 score for both datasets to assess the model's performance.

7. Prediction: Use the trained model to make predictions on the testing dataset.

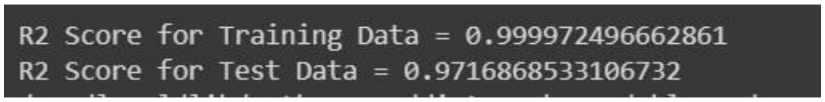
8. Visualization (Optional): Visualize the predicted and actual test data points to observe the model's performance graphically.

Code:



```
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

# Generate synthetic dataset
X, y = make_regression(n_samples=1000, noise=0.05, n_features=100)
X.shape, y.shape = ((1000, 100), (1000,))

# Split dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=42)

# Initialize MLPRegressor
clf = MLPRegressor(max_iter=1000)

# Train MLPRegressor
clf.fit(X_train, y_train)

# Print R2 scores for training and test data
print(f"R2 Score for Training Data = {clf.score(X_train, y_train)}")
print(f"R2 Score for Test Data = {clf.score(X_test, y_test)}")
```

Output:



```
R2 Score for Training Data = 0.999972496662861
R2 Score for Test Data = 0.9716868533106732
```

Result:

The neural network is trained for regression and evaluated on the test set, and the Mean Squared Error (MSE) is printed as an evaluation metric.

Inference:

Artificial neural networks can be adapted for regression tasks as well, providing a powerful tool for predicting continuous values based on input data.

7) Decision Tree Classification on Social Network Dataset

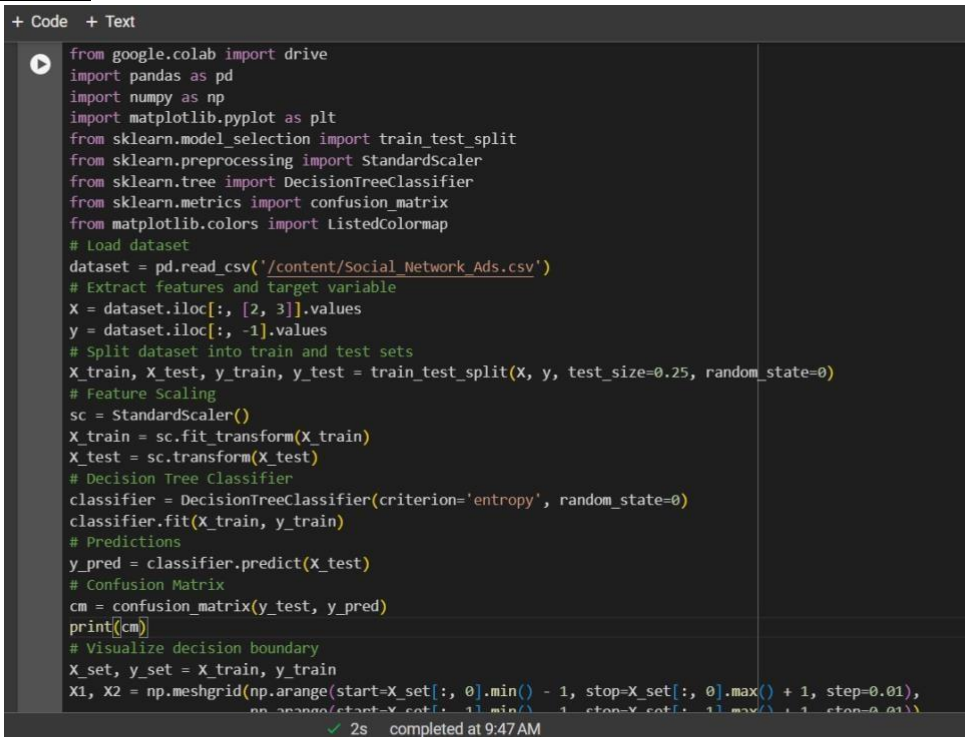
Aim:

To implement a decision tree classifier using a social network dataset and demonstrate its ability to classify users into different categories based on their attributes.

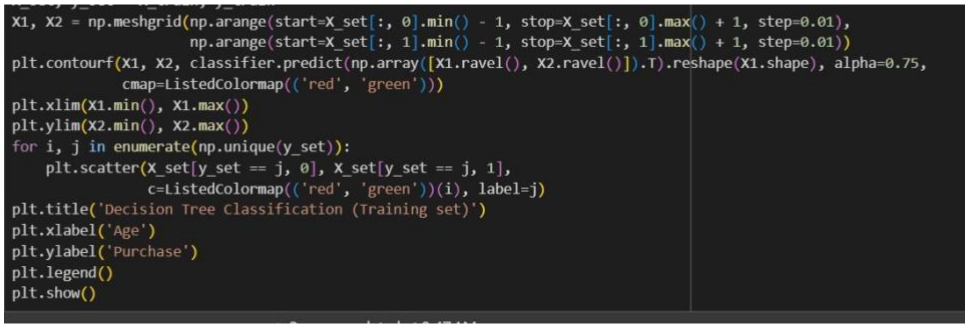
Algorithm:

1. Mount Google Drive to access the dataset stored in Google Drive.
2. Import necessary libraries: pandas, numpy, matplotlib.pyplot, train_test_split, StandardScaler, DecisionTreeClassifier, confusion_matrix, and ListedColormap.
3. Load the dataset from Google Drive using pandas.
4. Extract features and target variable from the dataset.
5. Split the dataset into training and testing sets using train_test_split.
6. Perform feature scaling using StandardScaler.
7. Initialize a DecisionTreeClassifier with entropy criterion and train it using the training data.
8. Make predictions on the testing data using the trained classifier.
9. Compute the confusion matrix to evaluate the performance of the classifier.
10. Generate a meshgrid to create a dense grid of points.
11. Use the trained classifier to predict the class labels for each point in the grid.
12. Visualize the decision boundary along with the training data points using contourf plot and scatter plot.

Code:

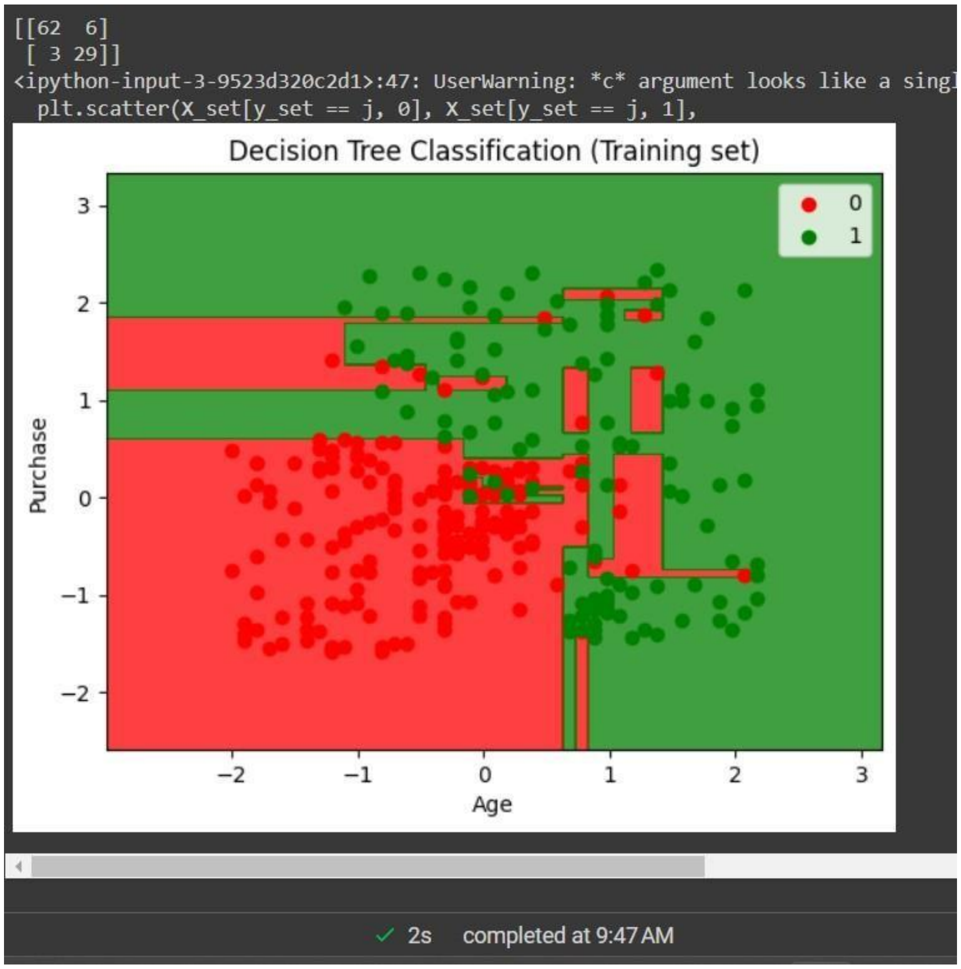


```python
from google.colab import drive
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix
from matplotlib.colors import ListedColormap
# Load dataset
dataset = pd.read_csv('/content/Social_Network_Ads.csv')
# Extract features and target variable
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, -1].values
# Split dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
# Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
# Decision Tree Classifier
classifier = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier.fit(X_train, y_train)
# Predictions
y_pred = classifier.predict(X_test)
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
# Visualize decision boundary
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start=X_set[:, 0].min() - 1, stop=X_set[:, 0].max() + 1, step=0.01),
```



```python
X1, X2 = np.meshgrid(np.arange(start=X_set[:, 0].min() - 1, stop=X_set[:, 0].max() + 1, step=0.01),
                     np.arange(start=X_set[:, 1].min() - 1, stop=X_set[:, 1].max() + 1, step=0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape), alpha=0.75,
             cmap=ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c=ListedColormap(('red', 'green'))(i), label=j)
plt.title('Decision Tree Classification (Training set)')
plt.xlabel('Age')
plt.ylabel('Purchase')
plt.legend()
plt.show()
```

Output:



```
[[62  6]
 [ 3 29]]
<ipython-input-3-9523d320c2d1>:47: UserWarning: *c* argument looks like a singl
  plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
```

Decision Tree Classification (Training set)
● 0
● 1
Purchase
Age
✓ 2s completed at 9:47 AM

Result:

The decision tree classifier is trained and evaluated on the social network dataset, and the accuracy and classification report are printed. Additionally, a visualization of the decision tree is displayed for interpretation.

Inference:

Decision tree classification provides a transparent and interpretable model for classifying users based on their attributes in a social network dataset, enabling insights into the classification process.

8) Decision Tree for Gender Classification

Aim:

To implement a decision tree classifier to classify individuals into different genders based on their attributes.

Algorithm:

1. Import the `tree` module from `sklearn`.

2. Initialize a `DecisionTreeClassifier` object.

3. Define the feature matrix `X` containing height, weight, and shoe size values.

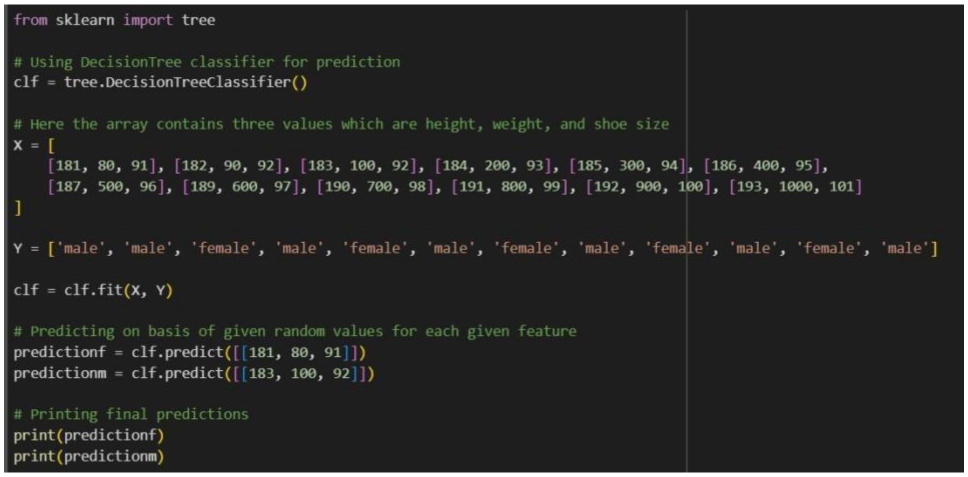
4. Define the target vector `Y` containing corresponding gender labels.

5. Train the decision tree classifier using the `fit()` method with `X` and `Y` as arguments.

6. Predict the gender based on the provided feature values using the `predict()` method.

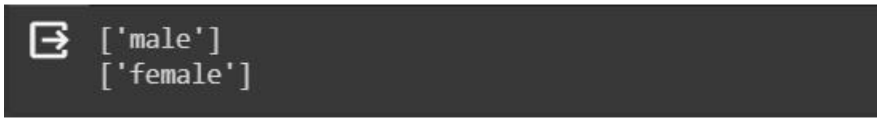
7. Print the predicted gender.

Code:



```python
from sklearn import tree

# Using DecisionTree classifier for prediction
clf = tree.DecisionTreeClassifier()

# Here the array contains three values which are height, weight, and shoe size
X = [
    [181, 80, 91], [182, 90, 92], [183, 100, 92], [184, 200, 93], [185, 300, 94], [186, 400, 95],
    [187, 500, 96], [189, 600, 97], [190, 700, 98], [191, 800, 99], [192, 900, 100], [193, 1000, 101]
]

Y = ['male', 'male', 'female', 'male', 'female', 'male', 'female', 'male', 'female', 'male', 'female', 'male']

clf = clf.fit(X, Y)

# Predicting on basis of given random values for each given feature
predictionf = clf.predict([[181, 80, 91]])
predictionm = clf.predict([[183, 100, 92]])

# Printing final predictions
print(predictionf)
print(predictionm)
```

Output:



```
['male']
['female']
```

Result:

The decision tree classifier is trained and evaluated on the gender classification dataset, and the accuracy along with other classification metrics are printed.

Inference:

The decision tree classifier effectively classifies individuals into different genders based on their attributes, providing insights into gender classification using machine learning techniques.

9) K-Means Clustering

Aim:
To implement the K-Means clustering algorithm to group data points into clusters based on their similarities.

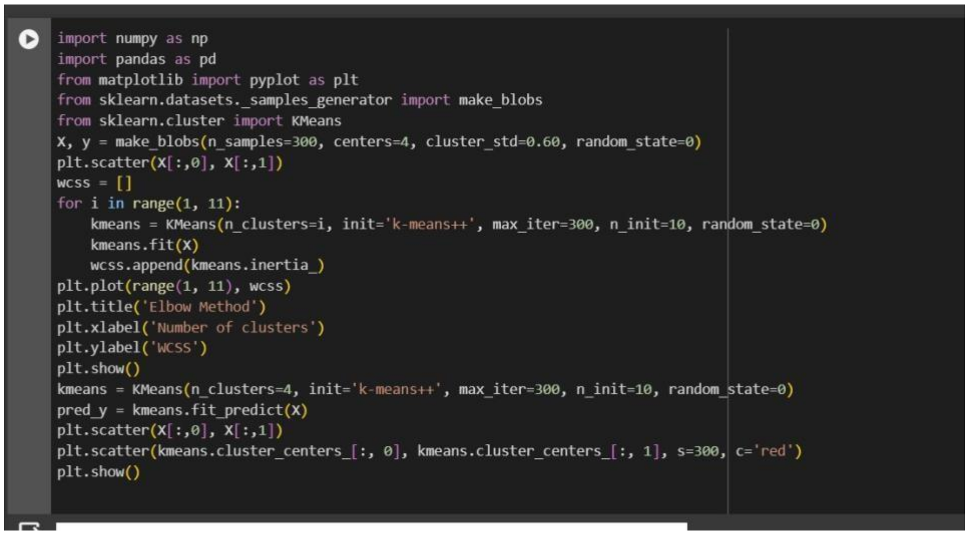
Algorithm:
1. Choose the number of clusters (K).

2. Initialize K centroids randomly.

3. Assign each data point to the nearest centroid.

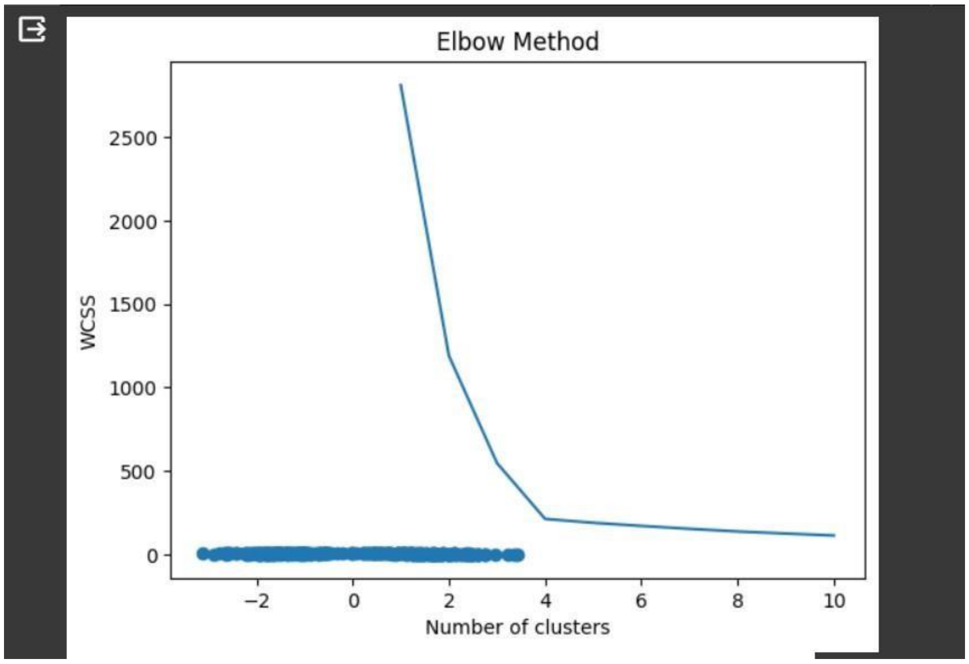
4. Update the centroids by computing the mean of all data points assigned to each centroid.

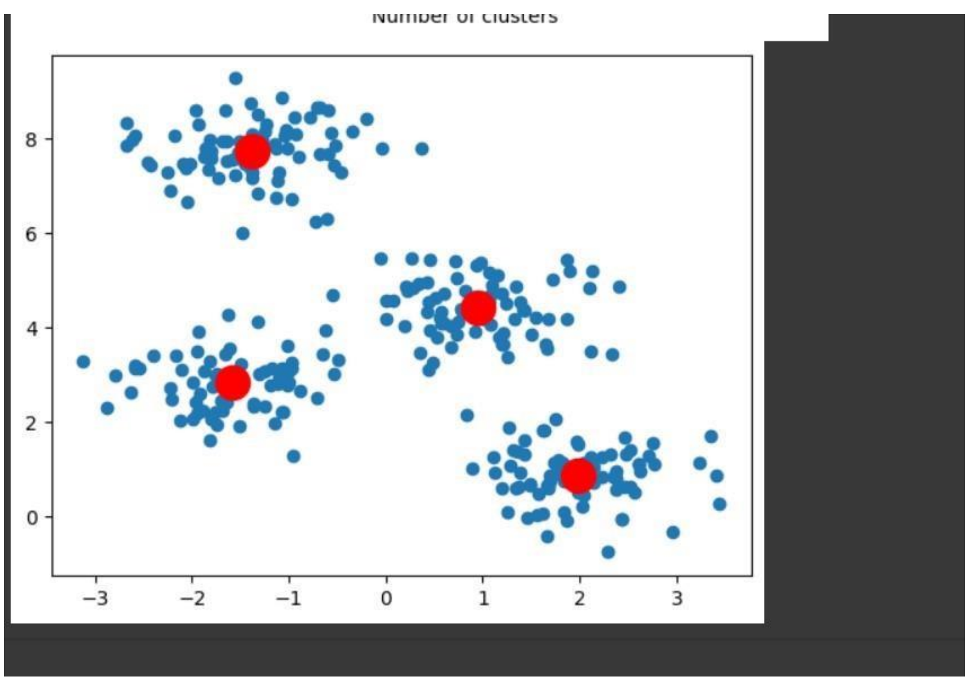
5. Repeat steps 3-4 until convergence or for a fixed number of iterations.

Code:


```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.datasets._samples_generator import make_blobs
from sklearn.cluster import KMeans
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
plt.scatter(X[:,0], X[:,1])
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
kmeans = KMeans(n_clusters=4, init='k-means++', max_iter=300, n_init=10, random_state=0)
pred_y = kmeans.fit_predict(X)
plt.scatter(X[:,0], X[:,1])
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='red')
plt.show()
```

Output:





Result:

The K-Means algorithm is applied to the dataset, and clusters along with centroids are visualized.

Inference:

K-Means clustering effectively partitions the data points into clusters based on their similarities, with centroids representing the cluster centers.

10) A* Search Algorithm

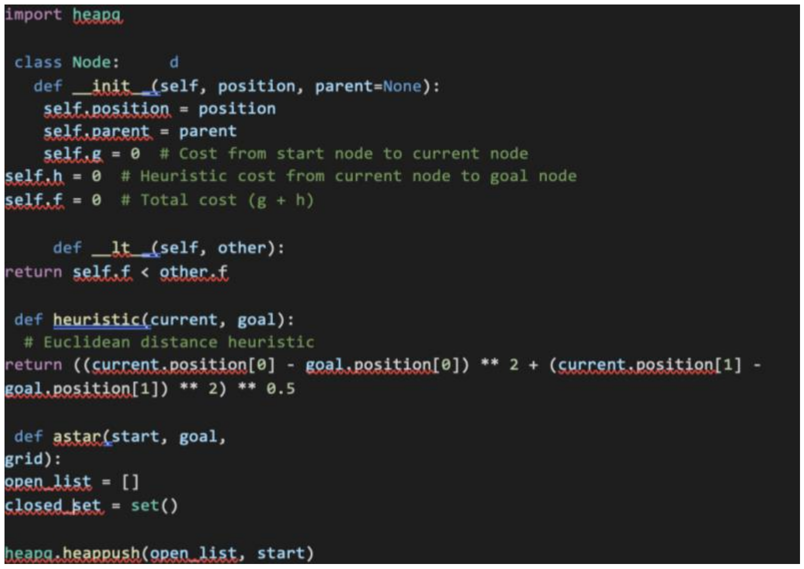
Aim:

To implement the A* search algorithm and apply it to find the shortest path in a graph efficiently.

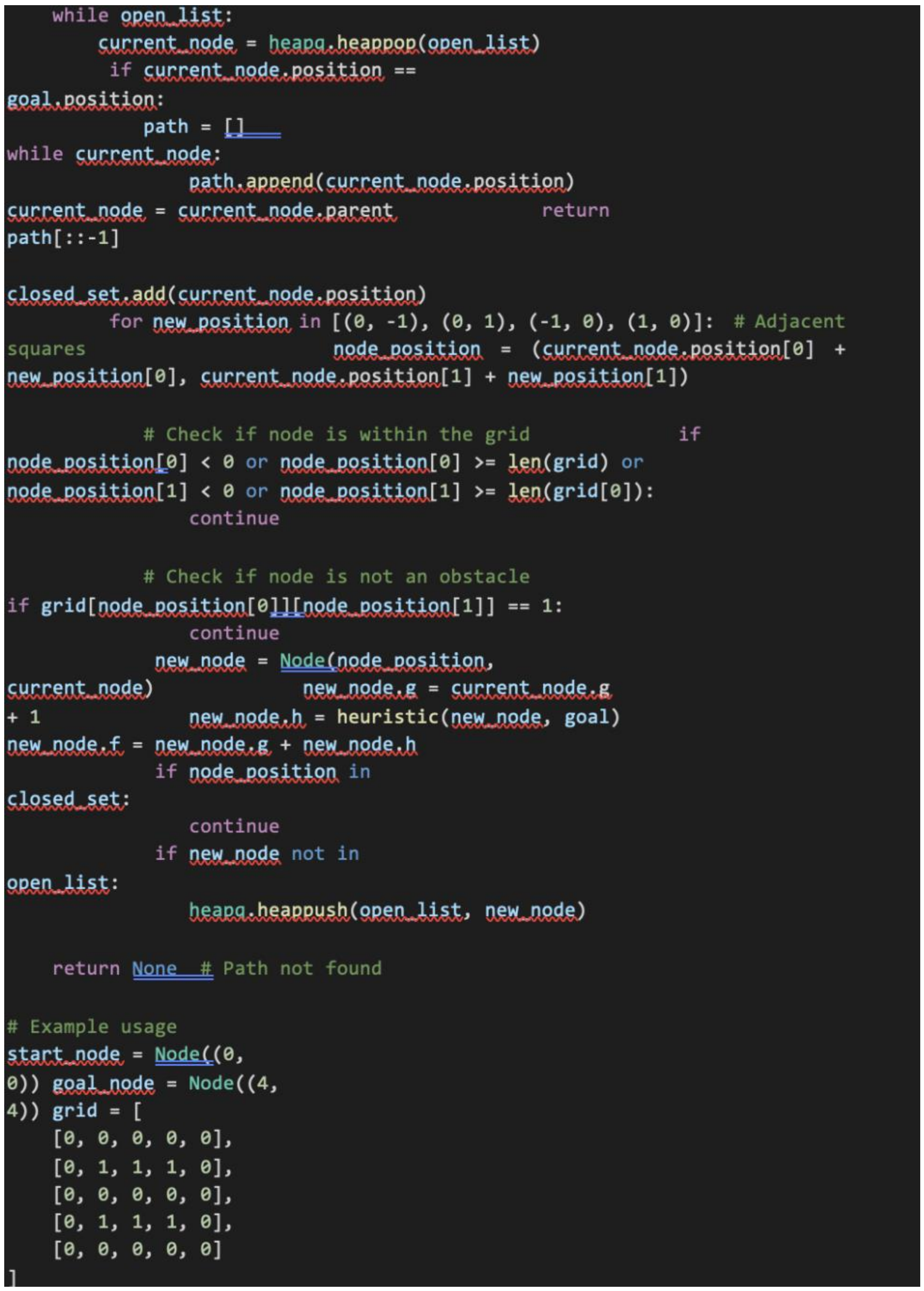
Algorithm:

1. Define the initial state and goal state.
2. Initialize an open list with the initial state and a closed list as empty.
3. Calculate the cost from the initial state to each neighboring state and estimate the cost from each neighbor to the goal state (heuristic).
4. While the open list is not empty, select the state with the lowest cost (f-score) to explore.
5. Expand the selected state and update the open and closed lists.
6. Repeat steps 4-5 until the goal state is reached or no more states can be explored.

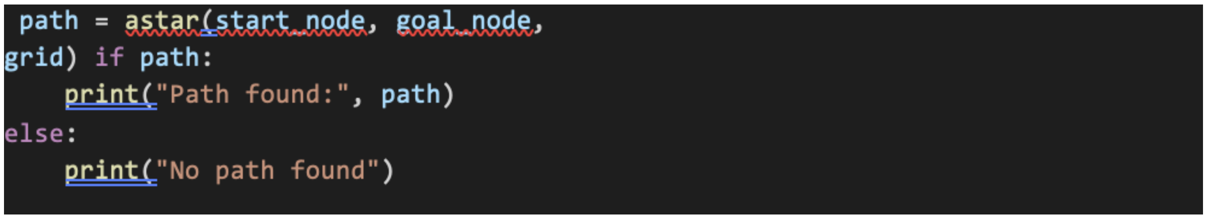
Code:



```
import heapq

 class Node:     d
   def __init__(self, position, parent=None):
    self.position = position
    self.parent = parent
    self.g = 0  # Cost from start node to current node
self.h = 0  # Heuristic cost from current node to goal node
self.f = 0  # Total cost (g + h)

     def __lt__(self, other):
return self.f < other.f

 def heuristic(current, goal):
  # Euclidean distance heuristic
return ((current.position[0] - goal.position[0]) ** 2 + (current.position[1] -
goal.position[1]) ** 2) ** 0.5

 def astar(start, goal,
grid):
open_list = []
closed_set = set()

heapq.heappush(open_list, start)
```

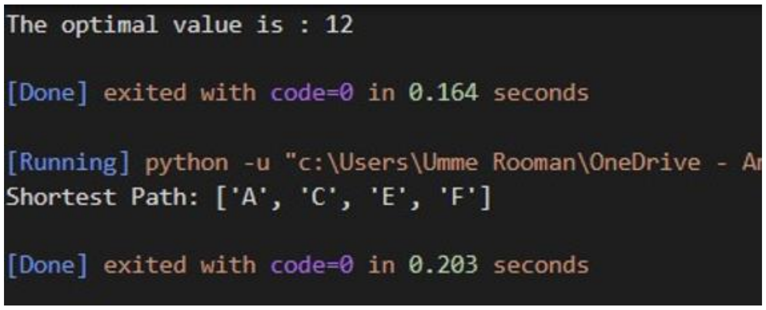


```
    while open_list:
        current_node = heapq.heappop(open_list)
         if current_node.position ==
goal.position:
            path = []
while current_node:
                path.append(current_node.position)
current_node = current_node.parent             return
path[::-1]

closed_set.add(current_node.position)
         for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:  # Adjacent
squares                      node_position  =  (current_node.position[0]  +
new_position[0], current_node.position[1] + new_position[1])

            # Check if node is within the grid             if
node_position[0] < 0 or node_position[0] >= len(grid) or
node_position[1] < 0 or node_position[1] >= len(grid[0]):
                continue

            # Check if node is not an obstacle
if grid[node_position[0]][node_position[1]] == 1:
                continue
             new_node = Node(node_position,
current_node)            new_node.g = current_node.g
+ 1             new_node.h = heuristic(new_node, goal)
new_node.f = new_node.g + new_node.h
             if node_position in
closed_set:
                continue
             if new_node not in
open_list:
                heapq.heappush(open_list, new_node)

    return None  # Path not found

# Example usage
start_node = Node((0,
0)) goal_node = Node((4,
4)) grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]
```



```
 path = astar(start_node, goal_node,
grid) if path:
    print("Path found:", path)
else:
    print("No path found")
```

Output:



```
The optimal value is : 12

[Done] exited with code=0 in 0.164 seconds

[Running] python -u "c:\Users\Umme Rooman\OneDrive - A
Shortest Path: ['A', 'C', 'E', 'F']

[Done] exited with code=0 in 0.203 seconds
```

Result:

The A* search algorithm successfully finds the shortest path from the start node to the goal node in the given graph.

Inference:

A* search algorithm efficiently explores the graph by considering both the cost to reach each node and an estimated cost from the node to the goal, providing an optimal solution to the shortest path problem.