**Practical No.: 01**

**Aim:** Implement the Breadth First Search algorithm to solve a given problem

**Date: Sign:**

Breadth First Search (BFS) is an algorithm used to traverse or search a graph or tree data structure. It

explores all the vertices of a graph in breadth-first order, starting from a specified vertex known as the

"source" vertex. BFS is often used to find the shortest path between two vertices in an unweighted

graph or to visit all the vertices in a connected component of a graph.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

**The BFS algorithm works as follows:**

a) Start by initializing a queue and a set to keep track of visited vertices.

b) En-queue the source vertex into the queue and mark it as visited.

c) Repeat the following steps until the queue becomes empty:

d) De-queue a vertex from the front of the queue.

e) Process the vertex (print it, store it, or perform any other desired operation).

f) En-queue all the unvisited neighbors of the vertex into the queue and mark them as visited.

g) The algorithm terminates when the queue becomes empty, indicating that all reachable

vertices have been processed.

**Applications of BFS algorithm**

The applications of breadth-first-algorithm are given as follows -

* BFS can be used to find the neighboring locations from a given source location.
* In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
* BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
* BFS is used to determine the shortest path and minimum spanning tree.
* BFS is also used in Cheney's technique to duplicate the garbage collection.

**Practical No.: 01**

**Aim:** Implement the Breadth First Search algorithm to solve a given problem

**Date: Sign:**

**Source Code:**

from collections import deque

def bfs(graph,start):

visited = set()

queue = deque([start])

while queue:

vertex= queue.popleft()

if vertex not in visited:

visited.add(vertex)

print(vertex)

neighbours=graph[vertex]

for neighbour in neighbours:

if neighbour not in visited:

queue.append(neighbour)

graph={

'A':['B','C'],

'B':['A','D','E'],

'C':['A','F'],

'D':['B'],

'E':['B','F'],

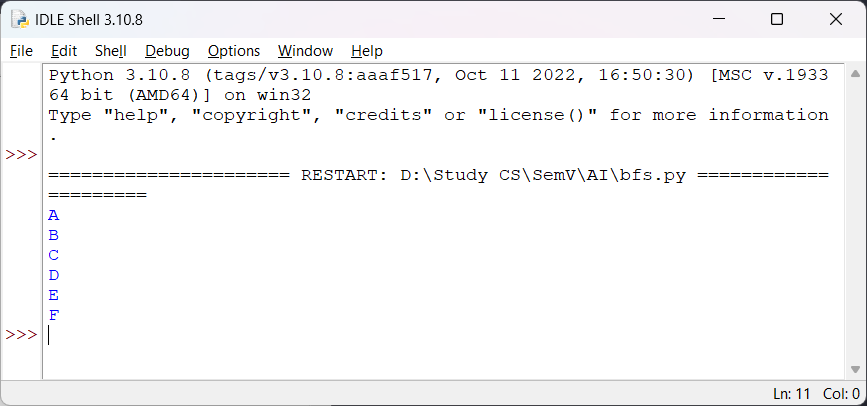
'F':['C','E'],

}

start\_vertex='A'

bfs(graph,start\_vertex)

**Output:**



**Practical No.: 02**

**Aim:** Implement the Iterative Depth First Search algorithm to solve the same problem.

**Date: Sign:**

Iterative deepening search or more specifically iterative deepening depth-first search is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found.

Interative depth First Traversal (or Search) for a graph is same as Depth First Traversal (DFS) for a tree. The only point is that, unlike trees, graphs can have cycles, thus a node may be visited twice. Use a boolean visited array to prevent processing a node multiple times.

**The iterative DFS algorithm works as follows:**

a) Start by initializing a stack and a set to keep track of visited vertices.

b) Push the source vertex onto the stack and mark it as visited.

c) Repeat the following steps until the stack becomes empty:

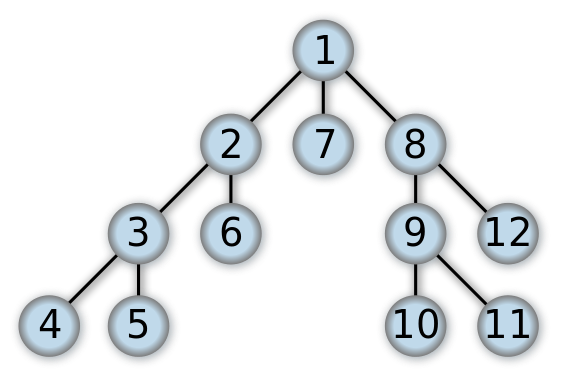
d) Pop a vertex from the top of the stack.

e) Process the vertex (print it, store it, or perform any other desired operation).

f) Retrieve all the unvisited neighbors of the vertex.

g) For each unvisited neighbor, push it onto the stack and mark it as visited.

h) The algorithm terminates when the stack becomes empty, indicating that all reachable vertices have been processed.



**Practical No.: 02**

**Aim:** Implement the Iterative Depth First Search algorithm to solve the same problem.

**Date: Sign:**

**Source Code:**

import queue as Q

from RMP import dict\_gn

start='Arad'

goal='Bucharest'

result=''

def DLS(city, visitedstack, startlimit, endlimit):

global result

found=0

result=result+city+' '

visitedstack.append(city)

if city==goal:

return 1

if startlimit==endlimit:

return 0

for eachcity in dict\_gn[city].keys():

if eachcity not in visitedstack:

found=DLS(eachcity, visitedstack, startlimit+1, endlimit)

if found:

return found

def IDDFS(city, visitedstack, endlimit):

global result

for i in range(0, endlimit):

print("Searching at Limit: ",i)

found=DLS(city, visitedstack, 0, i)

if found:

print("Found")

break

else:

print("Not Found! ")

print(result)

print("-----")

result=' '

visitedstack=[]

def main():

visitedstack=[]

IDDFS(start, visitedstack, 9)

print("IDDFS Traversal from ",start," to ", goal," is: ")

print(result)

main()

RMP.py

dict\_gn=dict(

Arad=dict(Zerind=75,Timisoara=118,Sibiu=140),

Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211),

Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146),

Drobeta=dict(Mehadia=75,Craiova=120),

Eforie=dict(Hirsova=86),

Fagaras=dict(Sibiu=99,Bucharest=211),

Giurgiu=dict(Bucharest=90),

Hirsova=dict(Eforie=86,Urziceni=98),

Iasi=dict(Neamt=87,Vaslui=92),

Lugoj=dict(Mehadia=70,Timisoara=111),

Mehadia=dict(Lugoj=70,Drobeta=75),

Neamt=dict(Iasi=87),

Oradea=dict(Zerind=71,Sibiu=151),

Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138),

Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146),

Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151),

Timisoara=dict(Lugoj=111,Arad=118),

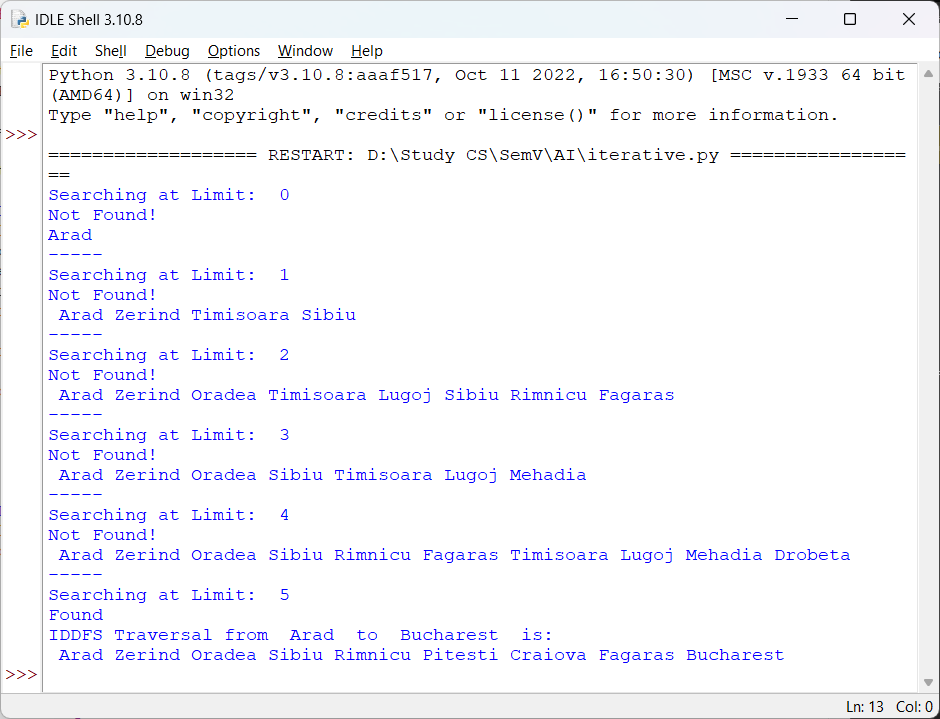
Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142),

Vaslui=dict(Iasi=92,Urziceni=142),

Zerind=dict(Oradea=71,Arad=75)

)

**Output:**



**Practical No.: 03**

**Aim:** Implement the A\* Search algorithm for solving a pathfinding problem

**Date: Sign:**

A\* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.).

A\* (A-star) is a widely used search algorithm that combines the best features of both Dijkstra's algorithm and heuristic search. It is commonly applied to solve the path-finding problem in graphs or grids, where the goal is to find the shortest path from a start node to a goal node. The A\* algorithm works by maintaining two main values for each node: the cost to reach the nodefrom the start node (known as g-value), and an estimate of the cost from the node to the goal node (known as h-value). It uses a priority queue, typically implemented as a min-heap, to prioritize the nodes for exploration based on their f-value, which is the sum of the g-value and h-value.

**The A\* algorithm follows these steps:**

a) Initialize the open list, closed list, and set the g-value of the start node to 0.

b) Calculate the h-value for each node in the graph or grid based on a heuristic function. The

heuristic function estimates the cost from each node to the goal node. Common heuristic

functions include Euclidean distance, Manhattan distance, or any other admissible and

consistent heuristic.

c) Enqueue the start node to the open list with its f-value as the priority.

d) Repeat the following steps until the open list becomes empty or the goal node is reached:

* Dequeue the node with the lowest f-value from the open list. This node becomes the

current node.

* If the current node is the goal node, the algorithm terminates, and the path has been

found.

* Add the current node to the closed list to mark it as visited.
* Explore the neighboring nodes of the current node:
* Calculate the tentative g-value for each neighbor by adding the cost to reach

the neighbor from the current node to the g-value of the current node.

* ii. If the neighbor is not in the closed list or its tentative g-value is lower than its

current g-value:

1) Update the g-value of the neighbor to the new lower value.

2) Calculate the f-value of the neighbor by adding its g-value and h-value.

3) If the neighbor is not in the open list, enqueue it with its f-value as the

priority.

4) If the neighbor is already in the open list, update its priority if the new fvalue is lower.

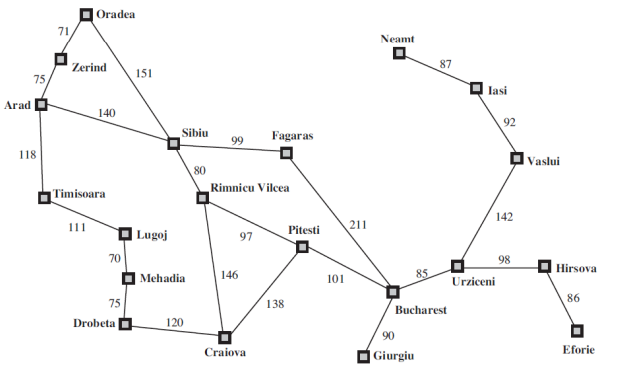
5) Set the parent of the neighbor to the current node.

e) If the open list becomes empty before reaching the goal node, there is no path available.

f) Once the goal node is reached, reconstruct the path by following the parent pointers from the goal node to the start node

The A\* algorithm is both complete (able to find a solution if one exists) and optimal (guaranteed to find the shortest path) under certain conditions. The heuristic used must be admissible, meaning it never overestimates the actual cost to reach the goal node. Additionally, the heuristic must be consistent (or monotonic), meaning the estimated cost from a node to its successor plus the heuristic value of the successor is always less than or equal to the estimated cost from the current node to the goal node.

A\* is widely used in various applications such as path planning, robotics, navigation systems, and game AI due to its efficiency and optimality. It efficiently explores the search space by prioritizing nodes that are most likely to lead to the goal while considering the actual cost from the start node.



**Practical No.: 03**

**Aim:** Implement the A\* Search algorithm for solving a pathfinding problem

**Date: Sign:**

**Source Code:**

import queue as Q

from RMP import dict\_gn

from RMP import dict\_hn

start='Arad'

goal='Bucharest'

result=''

def get\_fn(citystr):

cities=citystr.split(" , ")

hn=gn=0

for ctr in range(0, len(cities)-1):

gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]

hn=dict\_hn[cities[len(cities)-1]]

return(hn+gn)

def expand(cityq):

global result

tot, citystr, thiscity=cityq.get()

if thiscity==goal:

result=citystr+" : : "+str(tot)

return

for cty in dict\_gn[thiscity]:

cityq.put((get\_fn(citystr+" , "+cty), citystr+" , "+cty, cty))

expand(cityq)

def main():

cityq=Q.PriorityQueue()

thiscity=start

cityq.put((get\_fn(start),start,thiscity))

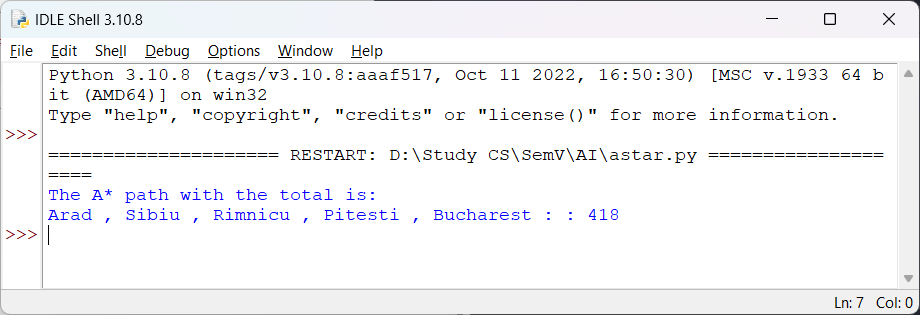
expand(cityq)

print("The A\* path with the total is: ")

print(result)

main()

**Output:**



**Practical No.: 04**

**Aim:** Implement the Recursive Best-First Search algorithm for the same problem

**Date: Sign:**

Recursive Best-First Search (RBFS) is an informed search algorithm used for finding the shortest path from a start node to a goal node in a graph or tree. RBFS is a memory-bounded variant of the A\* algorithm that avoids the need for storing the entire search tree explicitly.

RBFS utilizes a heuristic function that estimates the cost from each node to the goal node. It maintains a priority queue of nodes to be explored, sorted based on their f-values, which is the sum of the cost to reach the node from the start node (g-value) and the estimated cost from the node to the goal node (hvalue).

**The RBFS algorithm follows these steps:**

a) Initialize the recursive function RBFS with the current node, goal node, and a limit value

(initially set to infinity).

b) Check if the current node is the goal node. If it is, return the path containing only the current node, indicating success.

c) Generate the successors of the current node and calculate their f-values using a heuristic

function.

d) If there are no successors, return None to indicate failure.

e) Loop over the successors in order of their f-values:

* Recursively call RBFS on the successor with the minimum f-value, the goal

node, and the minimum of the current limit and the f-value of the next best

successor. This recursive call effectively explores the subtree rooted at the

selected successor.

* ii. If the recursive call returns a path, return the path concatenated with the current

node, indicating success.

* If the recursive call returns None, update the limit to the maximum f-value

among the successors. This limit represents the threshold beyond which RBFS

will not explore further.

* If all successors have been explored and none of them resulted in a path, return

None to indicate failure.

* The RBFS algorithm continues to recursively explore the graph/tree, using the

minimum limit from the failed recursive calls as the new limit for subsequent

iterations. This allows RBFS to "backtrack" and reconsider previously rejected

nodes if a better path is discovered.

RBFS terminates when a path is found or when it exhaustively explores all nodes without finding a path. The algorithm is memory-efficient since it only keeps track of the current path and the best fvalue encountered so far.

RBFS is especially useful in scenarios where memory is limited, and it prioritizes exploring promisingnodes based on their f-values, leading to efficient search in large state spaces.

**Practical No.: 04**

**Aim:** Implement the Recursive Best-First Search algorithm for the same problem

**Date: Sign:**

**Source Code:**

import queue as Q

from RMP import dict\_gn

from RMP import dict\_hn

start='Arad'

goal='Bucharest'

result=''

def get\_fn(citystr):

cities=citystr.split(',')

hn=gn=0

for ctr in range(0,len(cities)-1):

gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]

hn=dict\_hn[cities[len(cities)-1]]

return(hn+gn)

def printout(cityq):

for i in range(0,cityq.qsize()):

print(cityq.queue[i])

def expand(cityq):

global result

tot,citystr,thiscity=cityq.get()

nexttot=999

if not cityq.empty():

nexttot,nextcitystr,nextthiscity=cityq.queue[0]

if thiscity==goal and tot<nexttot:

result=citystr+'::'+str(tot)

return

print("Expanded city------------------------------",thiscity)

print("Second best f(n)------------------------------",nexttot)

tempq=Q.PriorityQueue()

for cty in dict\_gn[thiscity]:

tempq.put((get\_fn(citystr+','+cty),citystr+','+cty,cty))

for ctr in range(1,3):

ctrtot,ctrcitystr,ctrthiscity=tempq.get()

if ctrtot<nexttot:

cityq.put((ctrtot,ctrcitystr,ctrthiscity))

else:

cityq.put((ctrtot,citystr,thiscity))

break

printout(cityq)

expand(cityq)

def main():

cityq=Q.PriorityQueue()

thiscity=start

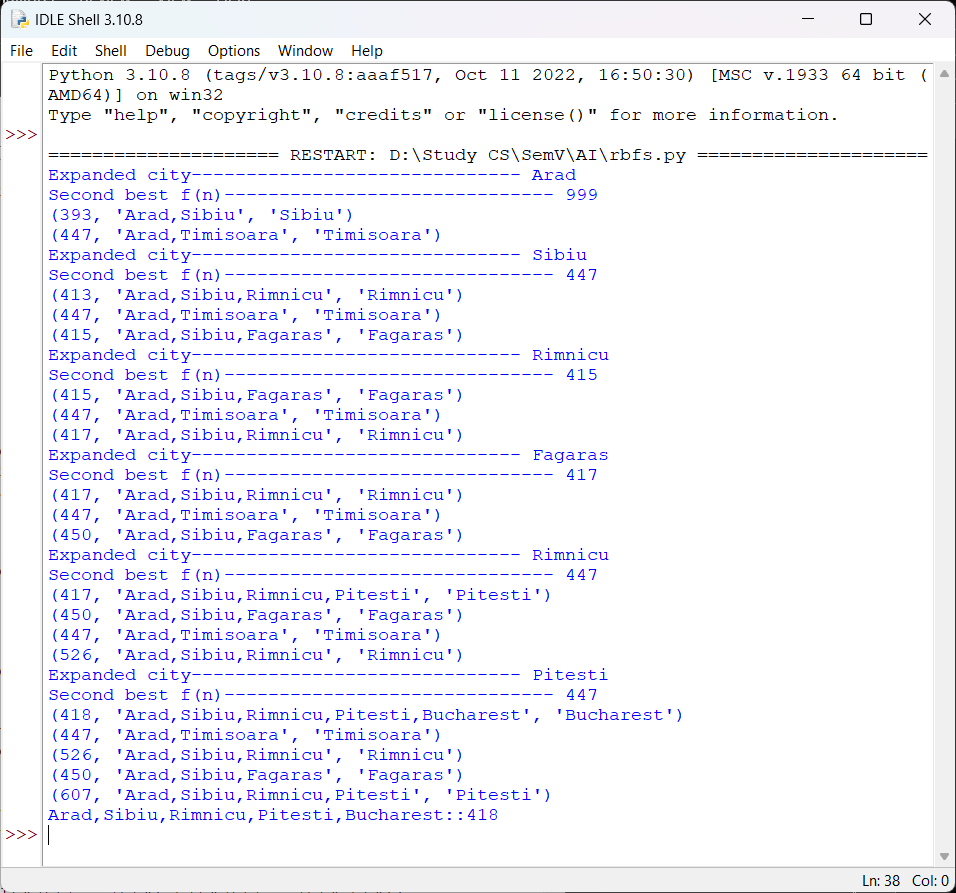
cityq.put((get\_fn(start),start,thiscity))

expand(cityq)

print(result)

main()

**Output:**



**Practical No.: 05**

**Aim:** Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

**Date: Sign:**

A Decision Tree is a supervised machine learning algorithm used for both classification and

regression tasks. It's a tree-like model that makes decisions based on a series of conditions or

rules learned from the training data. Each internal node in the tree represents a decision rule

based on a feature, and each leaf node represents a class label (in classification) or a predicted

value (in regression). Decision trees are characterized by their simplicity, interpretability, and

ability to handle both categorical and numerical data.

**Decision trees work in the following way :**

1. Root Node: At the root of the tree, the algorithm selects the feature that best separates the

data based on a criterion This feature becomes the root node.

2. Internal Nodes: The algorithm recursively selects features to split the data into subsets at each internal node. These splits are determined to minimize impurity (for classification) or reduce error (for regression).

3. Leaf Nodes: The process continues until a stopping criterion is met, such as a maximum

depth of the tree or a minimum number of data points in a node. The final nodes are called leaf nodes and represent the predicted class or value.

**Decision trees have several advantages that make them a popular choice for various machine learning and data analysis tasks. advantages of decision trees include:**

1. Interpretability: Decision trees provide a clear and intuitive representation of the decisionmaking process. It's easy to understand and explain the logic of a decision tree to non-technical stakeholders. This makes decision trees valuable in domains where model interpretability is essential, such as healthcare and finance.

2. Handling Mixed Data Types: Decision trees can handle both categorical and numerical data without the need for extensive data pre-processing. Other algorithms may require encoding categorical variables or scaling numerical features.

3. No Assumptions About Data Distribution: Decision trees do not assume that the data follows a particular statistical distribution, making them versatile for various types of data.

4. Non-Linearity: Decision trees can capture non-linear relationships between features and the target variable. They can model complex decision boundaries, which is especially useful when the relationship between variables is not linear.

5. Feature Importance: Decision trees can naturally identify feature importance by evaluating

which features are used for splitting nodes higher in the tree. This information can guide feature selection and dimensionality reduction efforts.

6. Robustness to Outliers: Decision trees are relatively robust to outliers in the data. Outliers

may affect individual branches of the tree but tend to have a limited impact on the overall

structure.

7. Scalability: Decision trees are computationally efficient, and the training time complexity is generally linear in the number of training examples and features. This makes them suitable for both small and large datasets.

**Practical No.: 05**

**Aim:** Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

**Date: Sign:**

**Source Code:**

from sklearn import datasets

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

iris = datasets.load\_iris()

X = iris.data

y = iris.target

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

clf = DecisionTreeClassifier()

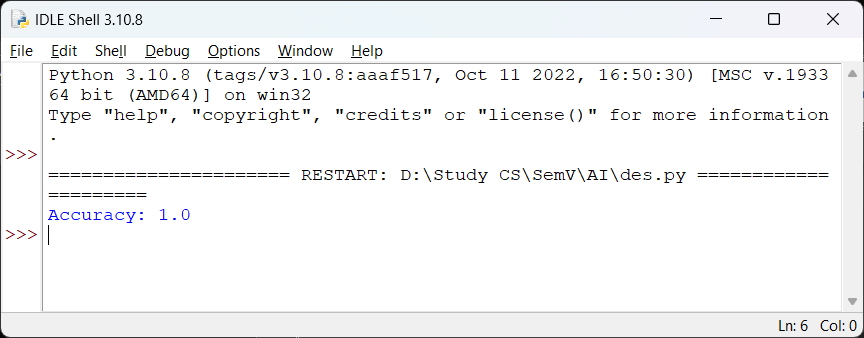
clf.fit(X\_train, y\_train)

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

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy}")

**Output:**



**Practical No.: 06**

**Aim:** Implement the Feed Forward Backpropagation algorithm to train a neural network

**Date: Sign:**

Feedforward neural networks, also known as feedforward neural networks (FNNs) or

multilayer perceptrons (MLPs), are a fundamental type of artificial neural network used in

machine learning and deep learning. They are designed to model complex relationships

between inputs and outputs by stacking multiple layers of interconnected neurons (also

called nodes or units).

**Here are the key characteristics of feedforward neural networks:**

1. Feedforward Structure: FNNs have a strict feedforward structure, meaning information

flows in one direction, from the input layer through one or more hidden layers to the

output layer. There are no feedback loops or recurrent connections in this architecture.

2. Layers: An FNN typically consists of three main types of layers:

- Input Layer: This layer contains neurons that represent the features or input data. Each

neuron corresponds to a specific input feature.

- Hidden Layers: These intermediate layers, which can be one or more, perform complex

transformations on the input data. Each neuron in a hidden layer is connected to all

neurons in the previous layer and feeds its output to the next layer.

- Output Layer: The final layer produces the network's predictions or outputs. The

number of neurons in the output layer depends on the problem; for regression tasks, it

may be one neuron, while for classification tasks, it can be one neuron per class.

3. Activation Functions: Non-linear activation functions (e.g., ReLU, sigmoid, or tanh) are

applied to the output of each neuron in the hidden layers. These functions introduce nonlinearity into the network, enabling it to capture complex patterns in the data.

4. Weights and Biases: Every connection between neurons has an associated weight that

determines the strength of the connection. Additionally, each neuron has a bias term that

helps shift the activation function. These weights and biases are learned during training to

optimize the network's performance.

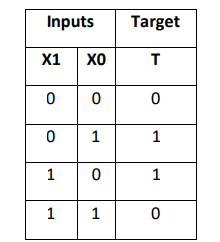
5. Training: FNNs are trained using supervised learning methods, such as gradient descent

and backpropagation. The network is presented with labelled training data, and it adjusts

its weights and biases to minimize the difference between its predictions and the actual

target values.

For the present case we implement the XOR- operation using Feedforward Backpropagation Neural Network. The XOR-operation for a 2-input variable is as follows



**Practical No.: 06**

**Aim:** Implement the Feed Forward Backpropagation algorithm to train a neural network

**Date: Sign:**

**Source Code:**

import numpy as np

class NeuralNetwork():

def \_\_init\_\_(self):

np.random.seed()

self.synaptic\_weights=2\*np.random.random((3,1))-1

def sigmoid(self, x):

return 1/(1+np.exp(-x))

def sigmoid\_derivative(self,x):

return x\*(1-x)

def train(self,training\_inputs,training\_outputs,training\_iterations):

for iteration in range(training\_iterations):

output=self.think(training\_inputs)

error=training\_outputs-output

adjustments=np.dot(training\_inputs.T,error\*self.sigmoid\_derivative(output))

self.synaptic\_weights+=adjustments

def think(self,inputs):

inputs=inputs.astype(float)

output=self.sigmoid(np.dot(inputs,self.synaptic\_weights))

return output

if \_\_name\_\_=="\_\_main\_\_":

neural\_network=NeuralNetwork()

print("Beginning randomly generated weights: ")

print(neural\_network.synaptic\_weights)

training\_inputs=np.array([[0,0,1],[1,1,1],[1,0,1],[0,1,1]])

training\_outputs=np.array([[0,1,1,0]]).T

neural\_network.train(training\_inputs,training\_outputs,15000)

print("Ending weights after training: ")

print(neural\_network.synaptic\_weights)

user\_input\_one=str(input("User Input One: "))

user\_input\_two=str(input("User Input Two: "))

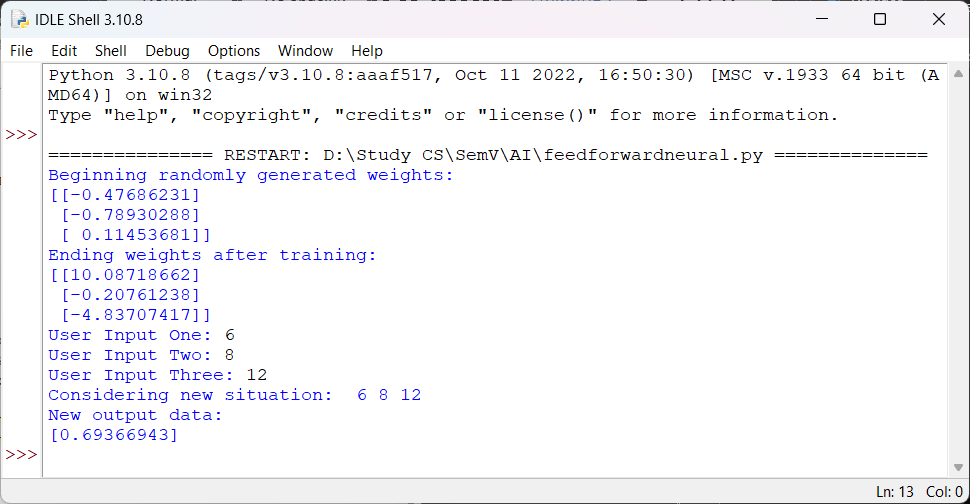
user\_input\_three=str(input("User Input Three: "))

print("Considering new situation: ",user\_input\_one,user\_input\_two,user\_input\_three)

print("New output data: ")

print(neural\_network.think(np.array([user\_input\_one,user\_input\_two,user\_input\_three])))

**Output:**



**Practical No.: 07**

**Aim** Implement the SVM algorithm for binary classification

**Date: Sign:**

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression. Though we say regression problems as well it’s best suited for classification. The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane. It becomes difficult to imagine when the number of features exceeds three.

Here's some essential theory about SVMs:

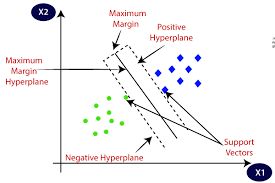
1. Linear Separability: SVMs are primarily designed for binary classification problems. They work by finding the optimal hyperplane that best separates two classes in the feature space. This hyperplane is chosen to maximize the margin between the two classes. When the classes are linearly separable, SVMs can find the hyperplane with the maximum margin.

2. Margin: The margin is the distance between the hyperplane and the nearest data point from either class. SVM aims to maximize this margin because a larger margin generally indicates a better separation and better generalization to unseen data.

3. Support Vectors: Support vectors are the data points that are closest to the hyperplane anddirectly influence its position and orientation. These are the critical data points that determine the margin. The SVM algorithm focuses on these support vectors during training.

4. Kernel Trick: SVMs can be extended to handle non-linearly separable data by using a kernel function. A kernel function transforms the original feature space into a higher-dimensional space, where the data may become linearly separable. Common kernel functions include the linear, polynomial, radial basis function (RBF/Gaussian), and sigmoid kernels.

5. C Parameter: The C parameter is a regularization parameter in SVM that balances the tradeoff between maximizing the margin and minimizing classification errors. A smaller C value results in a larger margin but may allow some training points to be misclassified, while a larger C value tries to classify all training points correctly but may result in a smaller margin.



**Practical No.: 07**

**Aim** Implement the SVM algorithm for binary classification

**Date: Sign:**

**Source Code:**

import numpy as np

from sklearn import datasets

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

X, y = datasets.make\_classification(n\_samples=100, n\_features=2, n\_informative=2, n\_redundant=0, random\_state=42)

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

svm\_classifier = SVC(kernel='linear')

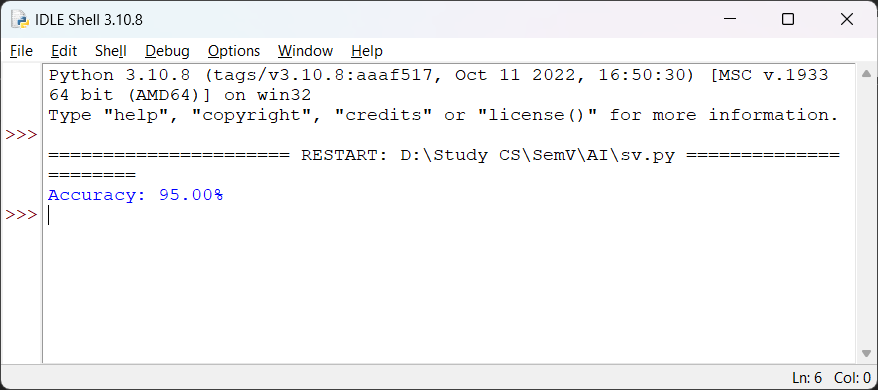
svm\_classifier.fit(X\_train, y\_train)

y\_pred = svm\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

**Output:**



**Practical No.: 08**

**Aim:** Implement the Naive Bayes' algorithm for classification

**Date: Sign:**

Naïve Bayes is a family of probabilistic machine learning algorithms based on Bayes'

Theorem. It's particularly useful for classification tasks, where you want to predict the

class label of an input based on its features. Despite its "Naïve" assumption, Naïve Bayes

has been shown to work surprisingly well in many real-world scenarios, especially in text

classification and other high-dimensional datasets.

**1. Bayes' Theorem:**

Naïve Bayes is built upon Bayes' Theorem, which is a fundamental concept in probability

theory. Bayes' Theorem describes how to update the probability of a hypothesis (an

event) based on new evidence. It's written as:

Where:

P(A | B) is the probability of event A given event B.

P(B | A) is the probability of event B given event A.

P(A) is the prior probability of event A.

P(B) is the evidence or marginal probability of event B.

**2. Naïve Assumption:**

The "Naïve" in Naïve Bayes refers to the assumption that the features (attributes) used to

predict the class are conditionally independent of each other given the class label. In other

words, the presence or absence of a particular feature doesn't affect the presence or

absence of any other feature. This simplifies the calculations significantly and makes the

algorithm computationally efficient.

**3. Training:**

During the training phase, Naïve Bayes estimates the probabilities needed for

classification. It calculates the prior probabilities of each class based on the training data

and then calculates the likelihood of each feature given each class.

For example, if you're classifying emails as "spam" or "not spam," you'd estimate the

probability of certain words appearing in spam emails versus non-spam emails.

**4. Prediction:**

When making a prediction for a new input, Naïve Bayes calculates the probability of the

input belonging to each class based on the features. It uses Bayes' Theorem to compute

the posterior probability for each class, and the class with the highest posterior probability

is chosen as the predicted class.

**Types of Naïve Bayes Classifiers:**

**There are several variants of Naïve Bayes classifiers, including:**

- Gaussian Naïve Bayes: Assumes that the features follow a Gaussian distribution.

- Multinomial Naïve Bayes: Used for discrete data, often in text classification (e.g.,

counting word occurrences).

- Bernoulli Naïve Bayes: Used for binary data, such as presence or absence of features.

- Categorical Naïve Bayes: Used for categorical data, where features have discrete values.

**Practical No.: 08**

**Aim:** Implement the Naive Bayes' algorithm for classification

**Date: Sign:**

**Source Code:**

from sklearn.datasets import load\_iris

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train,y\_test = train\_test\_split(X,y,test\_size=0.2,

random\_state=42)

clf =GaussianNB()

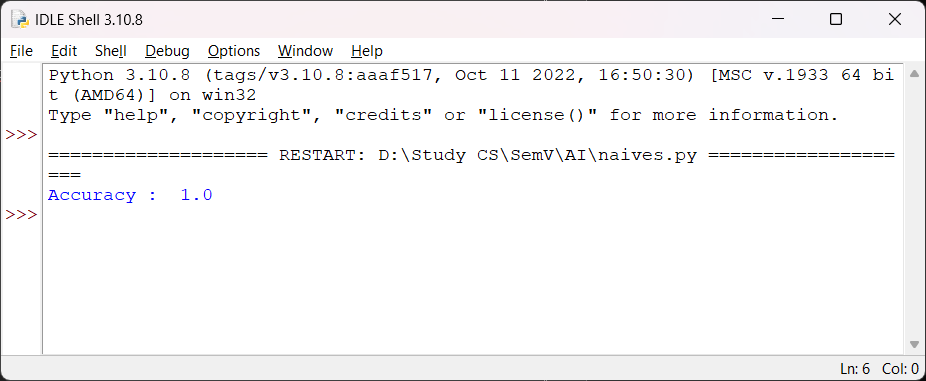
clf.fit(X\_train,y\_train)

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

accuracy = accuracy\_score(y\_test,y\_pred)

print('Accuracy : ',accuracy)

**Output:**



**Practical No.: 09**

**Aim:** Implement the K-NN algorithm for classification or regression.

**Date: Sign:**

K-Nearest Neighbors (K-NN) is a simple but powerful machine learning algorithm used for

both classification and regression tasks.

1. Intuition: K-NN is based on the idea that objects (data points) that are close to each

other in a feature space are more likely to belong to the same class or have similar values

(for regression).

2. How it works:

- Classification: Given a new data point, K-NN finds the K nearest data points in the

training dataset and assigns the class label that is most common among those K neighbors

to the new point.

- Regression: For regression tasks, K-NN calculates the average (or another aggregation)

of the target values of the K nearest neighbors and assigns this value to the new data

point.

3. Hyperparameter K: The choice of the hyperparameter K (the number of neighbors to

consider) is critical. A small K may lead to a noisy model (sensitive to outliers), while a

large K may lead to a biased model (smoothing over variations in the data). K is typically an

odd number to avoid ties in voting.

4. Distance Metric: K-NN uses a distance metric (e.g., Euclidean distance, Manhattan

distance, etc.) to measure the similarity between data points. The choice of distance

metric should be appropriate for your data and problem.

5. Scaling Features: It's important to scale features before applying K-NN, especially when

using distance-based metrics, to ensure that all features have equal influence on the

results.

**6. Pros:**

- Simple and easy to understand.

- No assumptions about the data distribution.

- Works well for both classification and regression tasks.

- Non-parametric (does not make assumptions about the functional form of

relationships).

**7. Cons:**

- Can be computationally expensive, especially for large datasets.

- Sensitive to the choice of K and the distance metric.

- Requires a sufficient amount of training data.

- May not perform well when the feature space is high-dimensional.

8. Use Cases:

- K-NN is often used for tasks such as recommendation systems, image classification, and

anomaly detection.

- It can be used as a baseline model for comparison with more complex algorithms.

9. Model Evaluation: Common evaluation metrics for K-NN include accuracy (for

classification) and mean squared error (for regression). Cross-validation is often used to

estimate the model's generalization performance.

**Practical No.: 09**

**Aim:** Implement the K-NN algorithm for classification or regression.

**Date: Sign:**

**Source Code:**

from sklearn.neighbors import KNeighborsClassifier

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

from sklearn.datasets import load\_iris

irisData = load\_iris()

X = irisData.data

y = irisData.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size = 0.2, random\_state=42)

knn = KNeighborsClassifier(n\_neighbors=7)

knn.fit(X\_train, y\_train)

print(knn.predict(X\_test))

**Output:**

