Practical No. 1: Breadth First Search & Iterative Depth First

Aim:

1) **To implement the Breadth First Search algorithm to solve a given problem.**

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])  # corrected to use queue instead of redeclaring deque
    while queue:
        vertex = queue.popleft()  # corrected deque to queue
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=" ")
            neighbors = graph[vertex]
            for neighbor in neighbors:
                if neighbor not in visited:  # fixed the typo 'visted' to 'visited'
                    queue.append(neighbor)

# Graph and BFS call outside the bfs function
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)
```

2) **To implement the Iterative Depth First Search algorithm to solve the same problem.**

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u) # Assuming an undirected graph
    def iterative_dfs(self, start, end):
        if start == end:
            return [start]
        visited = set()
        stack = [(start, [start])]
        while stack:
            current_vertex, path = stack.pop()
            visited.add(current_vertex)
```

```
            for neighbor in self.graph[current_vertex]:
                if neighbor not in visited:
                    if neighbor == end:
                        return path + [neighbor]
                    stack.append((neighbor, path + [neighbor]))
        return None # No path found
# Example usage:
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)
    g.add_edge(3, 6)
    g.add_edge(3, 7)
    g.add_edge(4, 8)
    g.add_edge(4, 9)
    g.add_edge(5, 10)
    g.add_edge(5, 11)
    g.add_edge(6, 12)
    g.add_edge(6, 13)
    g.add_edge(7, 14)
    g.add_edge(7, 15)
    start_node = 1
    end_node = 9
    shortest_path = g.iterative_dfs(start_node, end_node)
    if shortest_path:
        print(f"Shortest path from {start_node} to {end_node}: {shortest_path}")
    else:
        print(f"No path found from {start_node} to {end_node}")
```

Practical No.2: A* Search and Recursive Best-First Search A* Search:

Aim:

1) Implement the A* Search algorithm for solving a path finding problem.

2) Implement the Recursive Best-First Search algorithm for the same problem.

3) Compare the performance and effectiveness of both algorithms.

Code:

```
import heapq

# Define the map of Romania with distances between cities

romania_map = {

    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

    'Zerind': {'Arad': 75, 'Oradea': 71},
```

```
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucharest': 90},
    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}
class Node:
    def __init__(self, city, cost, parent=None):
        self.city = city
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost
def astar_search(graph, start, goal):
    open_list = []
    closed_set = set()
    heapq.heappush(open_list, start)
```

```
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.city == goal.city:
            path = []
            while current_node:
                path.append(current_node.city)
                current_node = current_node.parent
            return path[::-1]  # Reverse the path to get it from start to goal

        closed_set.add(current_node.city)

        for neighbor, distance in graph[current_node.city].items():
            if neighbor not in closed_set:
                new_cost = current_node.cost + distance
                new_node = Node(neighbor, new_cost, current_node)
                heapq.heappush(open_list, new_node)

    return None  # No path found if open_list is empty
# Test the A* search
start_city = 'Arad'
goal_city = 'Bucharest'
start_node = Node(start_city, 0)
goal_node = Node(goal_city, 0)
path = astar_search(romania_map, start_node, goal_node)
if path:
    print("Path found:", path)
else:
    print("No path found")
```

Practical No.3: Decision Tree Learning

Aim:

1) Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset

2) Evaluate the accuracy and effectiveness of the decision tree on test data

3) Visualize and interpret the generated decision tree.

Code:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier, plot_tree

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from sklearn.datasets import load_iris


# Load the Iris dataset directly from sklearn

iris = load_iris()


# Convert to a pandas DataFrame for easier manipulation

data = pd.DataFrame(data=iris.data, columns=iris.feature_names)


# Add the target (species) to the DataFrame

data['species'] = iris.target


# Defining the features and target variable

X = data.drop('species', axis=1)

y = data['species']


# Splitting the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Creating and training the Decision Tree model
```

```
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Making predictions on the test set
y_pred = clf.predict(X_test)

# Calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Visualizing the Decision Tree
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=X.columns, class_names=iris.target_names)
plt.title("Decision Tree Visualization")
plt.show()
```

Practical No.4: Feedforward Backpropagation Neural Network Feedforward Neural Network (FNN)

Aim:1. Implement the Feed Forward Backpropagation algorithm to train a neural network.

2. Use a given dataset to train the neural network for a specific task.

3. Evaluate the performance of the trained network on test data

Code:

```
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```

```
# Neural Network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.uniform(-1, 1, (input_size, hidden_size))
        self.weights_hidden_output = np.random.uniform(-1, 1, (hidden_size, output_size))

    def forward(self, inputs):
        self.hidden_input = np.dot(inputs, self.weights_input_hidden)
        self.hidden_output = sigmoid(self.hidden_input)
        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output)
        self.predicted_output = sigmoid(self.output_input)
        return self.predicted_output

    def backward(self, inputs, target, learning_rate):
        # Error in output layer
        error = target - self.predicted_output
        delta_output = error * sigmoid_derivative(self.predicted_output)

        # Error propagated to hidden layer
        error_hidden = delta_output.dot(self.weights_hidden_output.T)
        delta_hidden = error_hidden * sigmoid_derivative(self.hidden_output)

        # Update weights for hidden-to-output and input-to-hidden
        self.weights_hidden_output += np.outer(self.hidden_output, delta_output) * learning_rate
        self.weights_input_hidden += np.outer(inputs, delta_hidden) * learning_rate

    def train(self, training_data, targets, epochs, learning_rate):
        for epoch in range(epochs):
            for i in range(len(training_data)):
                inputs = training_data[i]
                target = targets[i]
```

```
            self.forward(inputs)
            self.backward(inputs, target, learning_rate)

    def predict(self, inputs):
        return self.forward(inputs)

# XOR training data
training_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])
# Define network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.1
epochs = 10000
# Initialize and train the neural network
nn = NeuralNetwork(input_size, hidden_size, output_size)
nn.train(training_data, targets, epochs, learning_rate)
# Test the network with training data
for i in range(len(training_data)):
    inputs = training_data[i]
    prediction = nn.predict(inputs)
    print(f"Input: {inputs}, Predicted Output: {prediction}")
```

Practical No. 5: Support Vector Machines(SVM) SVM

Aim:

1) Implement the SVM algorithm for binary classification.

2) Train an SVM model using a given dataset and optimize its parameters.

3) Evaluate the performance of the SVM model on test data and analyze the results.

Code:

```
import pandas as pd

from sklearn.svm import SVC

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

import os


# Specify the full path to the Iris dataset (replace with the correct path to your Iris.csv file)

file_path = 'C:/Users/Neeraj/Desktop/Tycs/Artificial Intelligence/practical AI file/Iris.csv'


# Check if the file exists

if os.path.exists(file_path):

   # Load the dataset

   data = pd.read_csv(file_path)


   # Print first few rows to verify correct data loading

   print(data.head())


   # Verify the column names (replace 'species' with the actual target column name in the dataset)

   print(data.columns)


   # Assuming the target column is named 'species', and others are features

   X = data.drop('species', axis=1)  # Features

   y = data['species']  # Target


   # Split the dataset into training and testing sets (80% train, 20% test)
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Initialize the Support Vector Classifier with a linear kernel
    svm_classifier = SVC(kernel='linear')

    # Train the classifier
    svm_classifier.fit(X_train, y_train)

    # Predict on the test set
    y_pred = svm_classifier.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.2f}")
else:
    print(f"File not found: {file_path}")
```

Practical No. 6:Adaboost Ensembale Learning.

AIM:

1. Implement the Adaboost algorithm to create an ensemble of weak classifiers.

2. Train the ensemble model on a given dataset and evaluate its performance

3. Compare the results with individual weak classifiers.

Code:

```
import pandas as pd
from sklearn import model_selection
from sklearn.ensemble import AdaBoostClassifier
# Load dataset
url="https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indiansdiabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pd.read_csv(url, names=names)
# Split dataset into input (X) and output (Y)
```

```
array = dataframe.values

X = array[:, 0:8]

Y = array[:, 8]

# Set parameters

seed = 7

num_trees = 30

# Define KFold cross-validator with shuffle

kfold = model_selection.KFold(n_splits=10, random_state=seed, shuffle=True)

# Define AdaBoost model with the SAMME algorithm

model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed, algorithm='SAMME')

# Evaluate the model using cross-validation

results = model_selection.cross_val_score(model, X, Y, cv=kfold)

# Print the results

print(results)
```

Practical No. 7: Naive Bayes Classifier Naïve Bayes Classifier

Aim:

1. **To implement the Naïve Bayes' algorithm for classification.**
   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)
   ```
2. **Train a Naïve Bayes' model using a given dataset and calculate class probabilities.**
   Code:
   ```
   import pandas as pd
   from sklearn.model_selection import train_test_split
   from sklearn.naive_bayes import GaussianNB
   from sklearn.metrics import accuracy_score
   from sklearn.preprocessing import LabelEncoder
   ```

```
# Load dataset
dataset = pd.read_csv('argfrc.csv')

# Check the column names
print(dataset.columns)

# Handle missing values (if any)
dataset = dataset.dropna()

# Select features and target
X = dataset[['Argentina', 'France']].values  # Ensure these columns exist and are numeric
y = dataset['Result'].values

# If 'Result' is categorical, encode it
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

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

# Initialize and train the Gaussian Naive Bayes model
clf = GaussianNB()
clf.fit(X_train, y_train)

# Make predictions and calculate accuracy
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Output the accuracy
print("Accuracy:", accuracy)
```

Practical No. 8: K - Nearest Neighbors (K-NN) K-NN:

Aim: 1. Implement the K-NN algorithm for classification or regression.

2. Apply the K-NN algorithm to a given dataset and predict the class or value for test data.

3. Evaluate the accuracy or error of the predictions and analyze the results

Code:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score

# Load the dataset from the CSV file
```

```
df = pd.read_csv('C:/Users/Neeraj/Downloads/Iris.csv')

# Extract relevant columns for the feature matrix (exclude 'ID' and 'Target' columns)

X = df.drop(['ID', 'Target'], axis=1).values

# Target variable for classification

y = df['Target'].values

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the value of k for K-NN

k = 3

# Classification with K-NN

clf = KNeighborsClassifier(n_neighbors=k)

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

# Evaluate classification accuracy

classification_accuracy = accuracy_score(y_test, y_pred)

print("Classification Accuracy:", classification_accuracy)
```

Practical No. 9: Association Rule Mining

The following is the Python code to implement Association Rule Mining (To run this code, MLxtend must beinstalled (pip install mlxtend) and Pandas must be installed (pip install pandas).

Code:

```
from mlxtend.frequent_patterns import apriori

from mlxtend.frequent_patterns import association_rules

import pandas as pd

dataset = [

 ['milk','bread','nuts'],

 ['milk','bread'],

 ['milk','eggs','nuts'],

 ['milk','bread','eggs'],

 ['bread','nuts'],

 ]
```

```
df=pd.DataFrame(dataset)

df_encoded = pd.get_dummies(df,prefix= '',prefix_sep='')

frequent_itemsets= apriori(df_encoded, min_support = 0.5, use_colnames=True)

print("Frequent itemsets:")

print(frequent_itemsets)

rules=association_rules(frequent_itemsets,metric="lift",min_threshold=1.0)

print("\nAssociation Rules:")

print(rules)
```

Practical No. 9: Association Rule Mining

The following is the Python code to implement Association Rule Mining (To run this code, MLxtend must beinstalled (pip install mlxtend) and Pandas must be installed (pip install pandas).

Code:

```
from mlxtend.frequent_patterns import apriori

from mlxtend.frequent_patterns import association_rules

import pandas as pd

dataset = [

 ['milk','bread','nuts'],

 ['milk','bread'],

 ['milk','eggs','nuts'],

 ['milk','bread','eggs'],

 ['bread','nuts'],

 ]

df=pd.DataFrame(dataset)

df_encoded = pd.get_dummies(df,prefix= '',prefix_sep='')

frequent_itemsets= apriori(df_encoded, min_support = 0.5, use_colnames=True)

print("Frequent itemsets:")

print(frequent_itemsets)

rules=association_rules(frequent_itemsets,metric="lift",min_threshold=1.0)

print("\nAssociation Rules:")

print(rules)
```