EXP 1: Depth First Search (DFS)

AIM: Write a program to implement Depth First Search (DFS) Algorithm in Python.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code)

PROGRAM:

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

visited = set()

def dfs(visited, graph, node):
   if node not in visited:
      print(node, end=" ")
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)

print("Following is the Depth-First Search:")
dfs(visited, graph, 'A')
```

OUTPUT:

```
Following is the Depth-First Search:
A B D E C F
```

CONCLUSION: The program successfully implemented the Depth-First Search algorithm, traversing the graph by exploring as far as possible along each branch before backtracking.

QUESTION:

1. Define Depth First Search (DFS). How is it different from Breadth First Search (BFS)?

- **DFS**: A graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack (implicitly or explicitly) to remember the next vertex to visit.
- **Difference from BFS**: DFS explores depth-wise (vertical) using a stack, while BFS explores breadth-wise (horizontal, layer by layer) using a queue. DFS might not find the shortest path, whereas BFS guarantees finding the shortest path in an unweighted graph.

2. Explain the working of the Depth First Search algorithm with the help of a suitable graph example and traversal order.

Working: Starting from an initial node, DFS visits an unvisited neighbor, then recursively calls itself for that neighbor. If no unvisited neighbors are found, it backtracks.

Example (from program):

1. Start at **A**. Visit **A**.
2. Go to A's first neighbor, **B**. Visit **B**.
3. Go to B's first neighbor, **D**. Visit **D**. **D** has no neighbors, so backtrack.
4. Go to B's next neighbor, **E**. Visit **E**. **E** has no neighbors, so backtrack.
5. **B** has no more unvisited neighbors, so backtrack to **A**.
6. Go to A's next neighbor, **C**. Visit **C**.
7. Go to C's first neighbor, **F**. Visit **F**. **F** has no neighbors, so backtrack.
8. **A** has no more unvisited neighbors. **Traversal Order: A → B → D → E → C → F**.

EXP 2: Greedy Best-First Search (Informed Search)

AIM: Write a program to implement Greedy Best-First Search in Python.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code)

PROGRAM:

```
import heapq

def greedy_best_first(graph, start, goal, heuristic):
    visited = set()
    pq = [(heuristic[start], start)]

    while pq:
        (h, node) = heapq.heappop(pq)
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            if node == goal:
                break
            for neighbor in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(pq, (heuristic[neighbor], neighbor))
```

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

heuristic = {'A': 5, 'B': 3, 'C': 2, 'D': 4, 'E': 1, 'F': 0}

print("Greedy Best-First Search Path:")
greedy_best_first(graph, 'A', 'F', heuristic)
```

OUTPUT:

```
Greedy Best-First Search Path:
A C F
```

CONCLUSION: The program successfully implemented the Greedy Best-First Search algorithm, finding a path from the start node 'A' to the goal node 'F' by prioritizing nodes with the lowest heuristic value.

QUESTION

1. Explain the working principle of Greedy Best-First Search. Why is it called an "Informed Search"?

- **Working Principle:** Greedy Best First Search expands the node that seems closest to the goal using a heuristic function h(n). It picks the node with the lowest h(n) value, ignoring the cost so far.
- **Why Informed Search:** It is called informed because it uses a heuristic to estimate the distance to the goal, helping it choose paths more wisely than BFS or DFS.

2. Why is it better than BFS and DFS algorithm?

- Greedy Best First Search is usually faster because it uses a heuristic to focus directly on the goal, skipping unpromising areas of the graph.
- While BFS and DFS search blindly without guidance, GBFS uses knowledge to choose promising paths, often reaching a solution more quickly.
- However, it does not always guarantee the shortest path, only a faster one.

EXP 3: Breadth First Search (BFS)

AIM: Write a program to implement Breadth First Search (BFS) Algorithm in Python.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code)

PROGRAM:

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

visited = []
queue = []

def bfs(visited, graph, node):
   visited.append(node)
   queue.append(node)

   while queue:
      m = queue.pop(0)
      print(m, end=" ")
      for neighbour in graph[m]:
         if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)

print("Following is the Breadth-First Search:")
bfs(visited, graph, 'A')
```

OUTPUT:

```
Following is the Breadth-First Search:
A B C D E F
```

CONCLUSION: The program successfully implemented the Breadth-First Search algorithm, traversing the graph level by level, visiting all neighbors of a node before moving to the next level.

QUESTION

1. Explain the working of the Breadth First Search algorithm with the help of a suitable graph example and traversal order.

Working: Starting from an initial node, BFS explores all of the neighbor nodes at the present depth before moving on to the nodes at the next depth level. It uses a queue to manage the order of nodes to visit.

Example (from program):

1. Start at **A**. Enqueue **A**. Dequeue **A**. Visit **A**. Enqueue its neighbors **B, C**. Queue: [B, C].
2. Dequeue **B**. Visit **B**. Enqueue its neighbors **D, E**. Queue: [C, D, E].
3. Dequeue **C**. Visit **C**. Enqueue its neighbor **F**. Queue: [D, E, F].
4. Dequeue **D**. Visit **D**. **D** has no neighbors. Queue: [E, F].
5. Dequeue **E**. Visit **E**. **E** has no neighbors. Queue: [F].
6. Dequeue **F**. Visit **F**. **F** has no neighbors. Queue: [].
7. Queue is empty. **Traversal Order: A → B → C → D → E → F**.

2. Discuss at least three applications of BFS in computer science and artificial intelligence.

1. **Finding the Shortest Path:** In unweighted graphs, BFS finds the shortest path between two nodes (in terms of number of edges) by exploring layer by layer.
2. **Web Crawlers/Indexing:** BFS can be used by web crawlers to systematically explore web pages. Starting from a seed URL, it visits all linked pages at the current depth before moving to links found on those pages.
3. **Social Network Analysis:** Finding connections between people (e.g., "friends of friends") or determining the "degree of separation" between individuals in a social network.

EXP 4: Split Dataset into Train and Test Sets

AIM: Write a program to split any dataset into train and test sets.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code), Scikit-learn (sklearn) library, Pandas library

PROGRAM:

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Example dataset
data = {'X': [1,2,3,4,5,6,7,8],
        'Y': [2,4,6,8,10,12,14,16]}

df = pd.DataFrame(data)

# Split data
X_train, X_test, y_train, y_test = train_test_split(df[['X']], df['Y'], test_size=0.25, random_state=1)

print("Train Set:")
print(pd.concat([X_train, y_train], axis=1))

print("\nTest Set:")
print(pd.concat([X_test, y_test], axis=1))
```

OUTPUT:

```
Train Set:
   X   Y
1  2   4
6  7  14
0  1   2
4  5  10
3  4   8
5  6  12

Test Set:
   X   Y
7  8  16
2  3   6
```

CONCLUSION: The program successfully demonstrated how to split a given dataset into training and testing sets using sklearn.modelselection.traintest_split, which is crucial for evaluating machine learning models.

QUESTION

1. Explain the difference between training set and testing set. Why is it important to separate them?

1. **Training Set:** The subset of the dataset used to train the machine learning model. The model learns patterns and relationships from this data.
2. **Testing Set:** The subset of the dataset used to evaluate the performance of the trained model on unseen data. It assesses how well the model generalizes to new examples.
3. **Importance of Separation:** Separating them is crucial to prevent overfitting. If a model is evaluated on the same data it was trained on, it might perform unrealistically well (memorizing training data) but fail on new, unseen data. The test set provides an unbiased evaluation of the model's generalization ability.

2. What would happen if we train and test a machine learning model on the same dataset without splitting? Explain with reasons.

- If a model is trained and tested on the same dataset, it would likely show very high accuracy or performance metrics during evaluation. This is because the model has already "seen" and learned from all the data, effectively memorizing it. This phenomenon is called overfitting.
- Reason: The model would have no way to demonstrate its ability to generalize to new, unseen data, as all data is "seen." The evaluation results would be overly optimistic and not reflect its real-world performance, making the model unreliable for practical applications.

EXP 5: Decision Tree

AIM: Create and display a Decision Tree on a given dataset.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code), Scikit-learn (sklearn) library, Matplotlib library

PROGRAM:

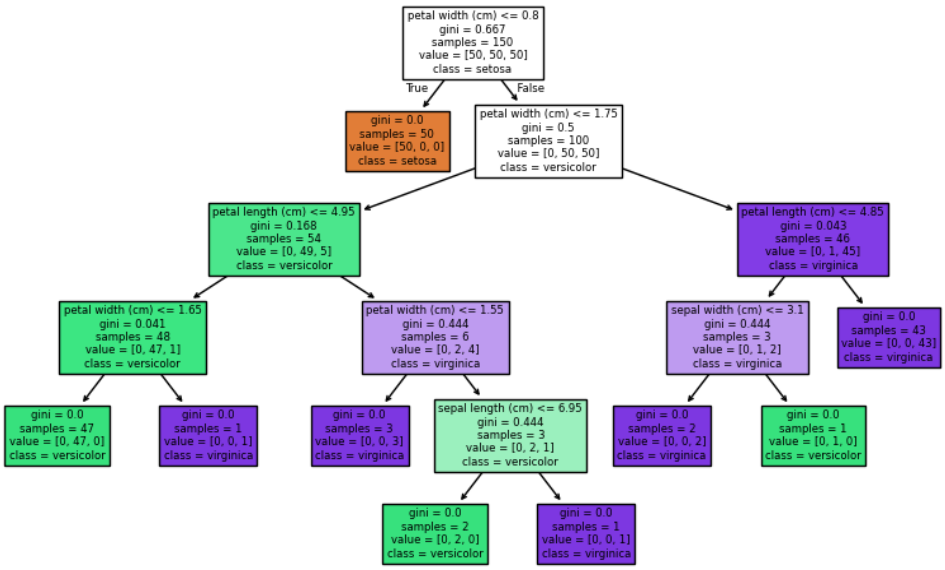
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

iris = load_iris()
X, y = iris.data, iris.target

clf = DecisionTreeClassifier()
clf.fit(X, y)

plt.figure(figsize=(10,6))
plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)
plt.show()
```

OUTPUT:



CONCLUSION: The program successfully trained a Decision Tree Classifier on the Iris dataset and generated a visual representation of the tree, allowing for easy interpretation of the decision rules.

QUESTION

1. When we create a Decision Tree using scikit-learn in Python, which function is used to visualize the tree? Explain with its parameters.

- The function used to visualize a Decision Tree in scikit-learn is **sklearn.tree.plot_tree**.
- **Key Parameters:**
    - **decisiontree:** The trained DecisionTreeClassifier object.
    - **featurenames:** A list of strings for the feature names, which are displayed at the decision nodes.
    - **classnames:** A list of strings for the target class names, which are displayed at the leaf nodes.
    - **filled:** Boolean; if True, draws nodes in color to indicate the majority class for classification.
    - **rounded:** Boolean; if True, draws node boxes with rounded corners.
    - **fontsize:** Integer; font size for text labels in nodes.
    - **max_depth:** Integer; limits the depth of the plotted tree.

2. What is a Decision Tree in machine learning? Describe its basic structure (root node, decision nodes, leaf nodes) with an example.

- **Decision Tree:** A non-parametric supervised learning algorithm used for both classification and regression tasks. It builds a model in the form of a tree structure, where each internal node represents a "decision" based on a feature, each branch represents an outcome of that decision, and each leaf node represents the final prediction or class label.
- **Basic Structure:**
    - **Root Node:** The topmost node in the tree. It represents the initial decision based on the most significant feature that best splits the dataset.
    - **Decision Nodes (Internal Nodes):** Nodes that have one or more branches. They represent a test on a specific feature, with each branch corresponding to a possible value or range of values for that feature.
    - **Leaf Nodes (Terminal Nodes):** Nodes that do not have any further branches. They represent the final outcome or class prediction after all decisions have been made.
- **Example (simplified):**
    - **Root Node:** "Is it raining?" (Yes/No)
    - **Decision Node (if Yes):** "Do I have an umbrella?" (Yes/No)
    - **Leaf Node (if Yes, and Yes to umbrella):** "Go outside."
    - **Leaf Node (if Yes, and No to umbrella):** "Stay inside."
    - **Leaf Node (if No to raining):** "Go outside."

EXP 6: Simple Linear Regression

AIM: Write a program to implement Simple Linear Regression using Python.

REQUIREMENTS(Software/Hardware):- Python Interpreter, Text Editor/IDE(VS Code), Scikit-learn (sklearn) library, NumPy library

PROGRAM:

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Example data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

model = LinearRegression()
model.fit(X, y)

print("Slope (m):", model.coef_[0])
print("Intercept (c):", model.intercept_)
print("Prediction for x=6:", model.predict([[6]])[0])
```

OUTPUT:

```
Slope (m): 2.0
Intercept (c): 0.0
Prediction for x=6: 12.0
```

CONCLUSION: The program successfully implemented Simple Linear Regression, calculating the slope and intercept from the given data, and demonstrated how to use the trained model for making predictions.

QUESTION

1. What is the difference between Simple Linear Regression and Multiple Linear Regression? Give examples of both.

1. **Simple Linear Regression:** Predicts a dependent variable using a single independent variable. **Example**: Predicting a person's weight based on their height.
2. **Multiple Linear Regression:** Predicts a dependent variable using two or more independent variables. **Example**: Predicting a house price based on its size, number of bedrooms, and location.

2. What is Simple Linear Regression? Write its mathematical equation and explain each term.

- It shows the relationship between one independent variable (X) and one dependent variable (Y) using a straight line.
- **Equation:**

$$y=mx+b$$

Terms:

  - **y:** Dependent variable (output)
  - **x:** Independent variable (input)
  - **m:** Slope of the line (change in y for one unit change in x)
  - **b:** Intercept (value of y when x = 0)