PRACTICAL 1: Breadth First Search (BFS)

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)

            neighbors=graph[vertex]
            for neighbor in neighbors:
                if neighbor not in visited:
                    queue.append(neighbor)

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

OUTPUT:

```
A
B
C
D
E
F
>>>
```

PRACTICAL 2: Depth First Search (DFS)

CODE:

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

def dfs(start):
    visited=set()
    stack=[start]

    while stack:
        vertex=stack.pop()
        if vertex not in visited:
            print(vertex,end=" ")
            visited.add(vertex)
            stack.extend(reversed(graph[vertex]))
```

```
dfs('A')
```

OUTPUT:

PRACTICAL 3:Recursive DFS

CODE:

```
graph={'A':['B','C'],
       'B':['D','E'],
       'C':['F'],
       'D':[],
       'E':['F'],
       'F':[]
       }
visited=set()
def dfs(vertex):
    if vertex not in visited:
        print(vertex,end=" ")
        visited.add(vertex)
        for neighbor in graph[vertex]:
            dfs(neighbor)
dfs('A')
```

OUTPUT:

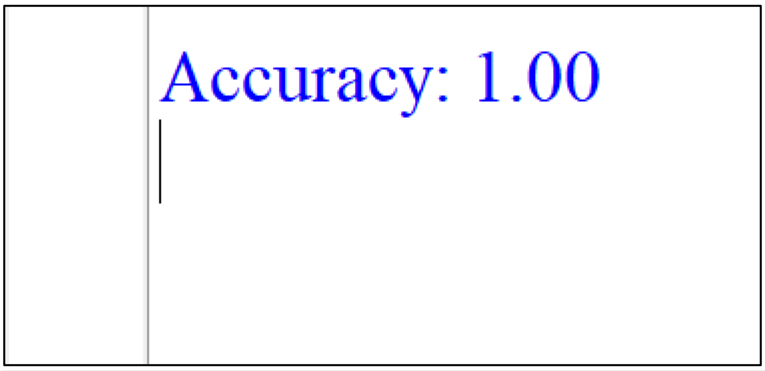
PRACTICAL 4:Decision Tree Learning

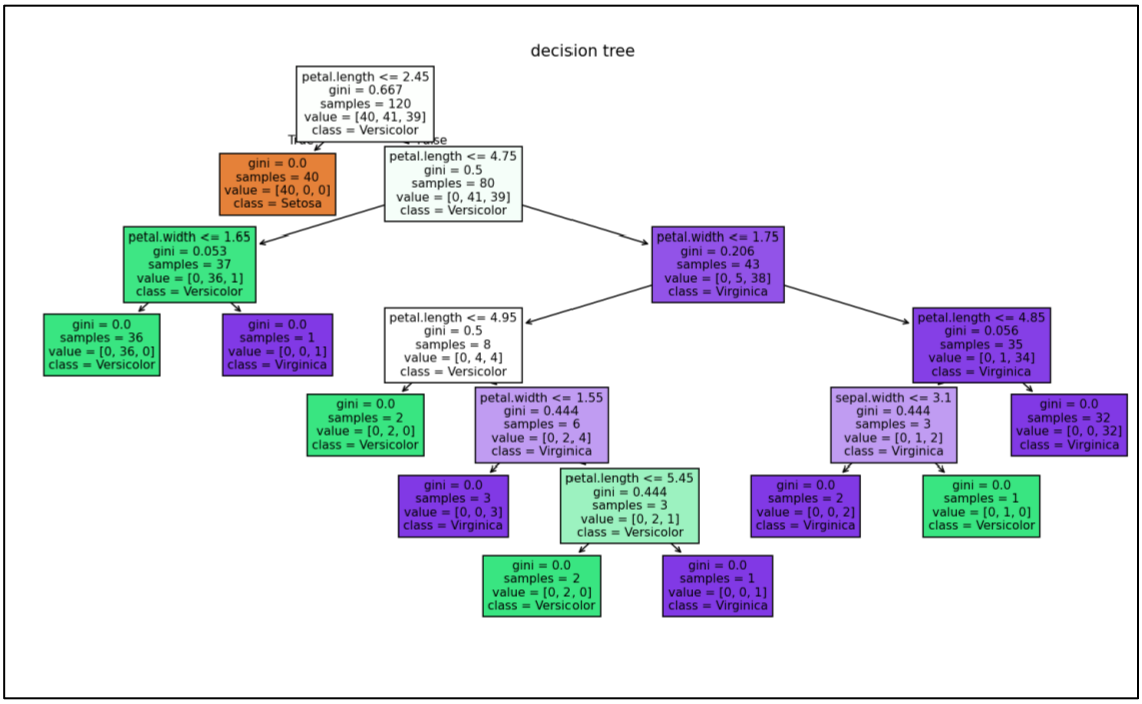
CODE:

```
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

data=pd.read_csv("data1.csv")

X=data.drop(columns=["variety"])
Y=data['variety']

X_train, X_test, Y_train, Y_test=train_test_split(X,Y,test_size=0.2,random_state=42)
clf=DecisionTreeClassifier(random_state=42)
clf.fit(X_train,Y_train)
Y_pred=clf.predict(X_test)

accuracy=accuracy_score(Y_test,Y_pred)
print(f"Accuracy:{accuracy: .2f}")
plt.figure(figsize=(12,8))
```

```
plot_tree(clf,filled=True,feature_names=X.columns,class_names=Y.unique().astype(str))
plt.title("decision tree ")
plt.show()
```

OUTPUT:





PRACTICAL 5:Water Jug Problem

CODE:

```
jugA,jugB=0,0
capA,capB=4,3
tar=2
jugB=capB
print(f"step 1:jugA={jugA},jugB={jugB}")
jugA,jugB=jugB,0
print(f"step 2:jugA={jugA},jugB={jugB}")
jugB=capB
print(f"step 3:jugA={jugA},jugB={jugB}")
transfer=capA-jugB
jugA+=transfer
jugB-=transfer
print(f"step 4:jugA={jugA},jugB={jugB}")
```

OUTPUT:

```
step 1:jugA=0,jugB=3
step 2:jugA=3,jugB=0
step 3:jugA=3,jugB=3
step 4:jugA=4,jugB=2
>>>
```

PRACTICAL 6:Tower Of Hanoi

CODE:

```
def tower_of_hanoi(n,source,destination,auxillary):
    if n==1:
        print(f"move disk 1 from {source}->{destination}")
```

```
        return
    tower_of_hanoi(n-1,source,auxillary,destination)
    print(f"move disk {n} from {source}->{destination}")
    tower_of_hanoi(n-1,auxillary,destination,source)
tower_of_hanoi(3,'A','C','B')
```

OUTPUT:

```
move disk 1 from A->C
move disk 2 from A->B
move disk 1 from C->B
move disk 3 from A->C
move disk 1 from B->A
move disk 2 from B->C
move disk 1 from A->C
>>>
```

PRACTICAL 7: A* Algorithm

CODE:

```
import heapq
graph={
    'S':{'B':4, 'C':3},
    'B':{'F':5, 'E':12},
    'C':{'D':7, 'E':10},
    'D':{'E':2},
    'E':{'G':5},
    'F':{'G':16},
    'G':{}
    }
h={'S':14,  'B':12, 'C':11, 'D':6, 'E':4, 'F':11, 'G':0}
```

```
def a_star(start,goal):
    pq=[(h[start],0,start,[])]
    while pq:
        f,g,node,path=heapq.heappop(pq)
        path=path+[node]
        if node==goal:
            return path
        for nbr,cost in graph[node].items():
            heapq.heappush(pq,(g+cost+h[nbr],g+cost,nbr,path))
print("shortest path:",a_star('S','G'))
```

OUTPUT:

```
shortest path: ['S', 'C', 'D', 'E', 'G']
>>>
```