1.Depth First Search Algorithm

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

def dfs(start):
   visited=set()
   stack=[start]
   while stack:
      node=stack.pop()
      if node not in visited:
         print(node,end="")
         visited.add(node)

         stack.extend(reversed(graph[node]))

dfs('A')
```

```
ABDEFC
```

2. Depth First Search Algorithm - Recursive

```
graph={
   'A':['B','C'],
```

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

```
ABDEFC
```

3. Decision Tree Classification

```
from queue import PriorityQueue
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={
```

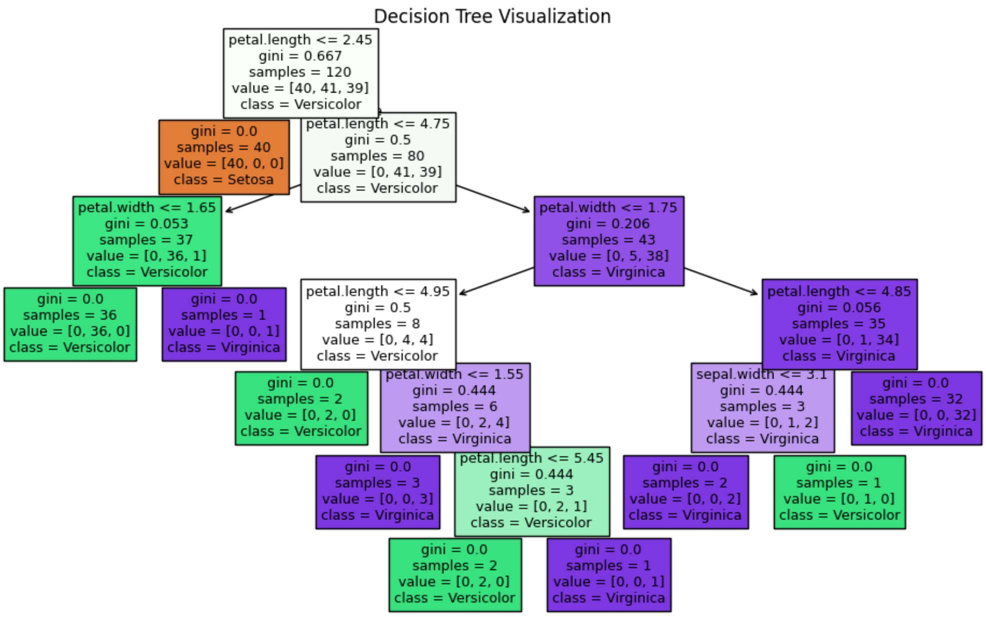
```
    'S':14,'B':12,'C':11,
    'D':6,'E':4,'F':11,
    'G':0
}

def a_start(start,goal):
    pq=PriorityQueue()
    pq.put((h[start],start))
    parent={start:None}
    g={start:0}

    while not pq.empty():
        f,node=pq.get()
        if node==goal:
            path=[]
            while node:
                path.append(node)
                node=parent[node]
            return path[::-1]

        for neighbor,cost in graph[node].items():
            newg=g[node]+cost
            if neighbor not in g or newg<g[neighbor]:
                g[neighbor]=newg
                f=newg+h[neighbor]
                pq.put((f,neighbor))
                parent[neighbor]=node

path=a_start('S','G')
print("Shortest path:",path)
```



4.Breadth First Search Algorithm

```
from collections import deque

def bfs(graph,start):

   visited=set()

   queue=deque([start])

   while queue:

      vertex=queue.popleft()

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

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

5. Water jug Problem

```
def water_jug_problem():
    jug1,jug2=0,0
    cap1,cap2=4,3
    steps=[]

    jug2=cap2
    steps.append((jug1,jug2))
    jug1,jug2=jug1,0
    steps.append((jug1,jug2))

    jug2=cap2
    steps.append((jug1,jug2))
    transfer=cap1-jug1
    jug1+=transfer
    jug2-=transfer
    steps.append((jug1,jug2))
```

```
    for i,(x,y) in enumerate(steps):
        print(f"steps{i+1}=jug1={x},jug2={y}")
water_jug_problem()
```

```
steps1=jug1=0,jug2=3
steps2=jug1=0,jug2=0
steps3=jug1=0,jug2=3
steps4=jug1=4,jug2=-1
```

6.Tower of Hanoi

```
def tower_of_hanoi(n,source,destination,auxilary):
    if n==1:
        print(f"Move disk1 from {source} to {destination}")
        return
    tower_of_hanoi(n-1,source,auxilary,destination)
    print(f"Move disk {n-1} from {source} to {destination}")
    tower_of_hanoi(n-1,auxilary,destination,source)
tower_of_hanoi(2,'A','C','B')
```

```
Move disk1 from A to B
Move disk 1 from A to C
Move disk1 from B to C
```

7.A* Algorithm

```
from queue import PriorityQueue
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_start(start,goal):
    pq=PriorityQueue()
    pq.put((h[start],start))
    parent={start:None}
    g={start:0}

    while not pq.empty():
        f,node=pq.get()
        if node==goal:
            path=[]
            while node:
                path.append(node)
                node=parent[node]
            return path[::-1]

        for neighbor,cost in graph[node].items():
            newg=g[node]+cost
            if neighbor not in g or newg<g[neighbor]:
                g[neighbor]=newg
                f=newg+h[neighbor]
```

```
            pq.put((f,neighbor))
            parent[neighbor]=node

path=a_start('S','G')
print("Shortest path:",path)
```

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