# Depth First Iterative Deepening Search (DFID)

```
from collections import defaultdict

graph = defaultdict(list)


def addEdge(u, v):
    graph[u].append(v)


def dfs(start, goal, depth):
    print(start, end=" ")
    if start == goal:
        return True
    if depth <= 0:
        return False
    for i in graph[start]:
        if dfs(i, goal, depth - 1):
            return True
    return False


def dfid(start, goal, maxDepth):
    print("Start node: ", start, "Goal node: ", goal)
    for i in range(maxDepth):
        print("\nDFID at level : ", i + 1)
        print("Path Taken : ", end=' ')
        isPathFound = dfs(start, goal, i)
    if isPathFound:
        print("\nGoal node found!")
        return
    else:
        print("\nGoal node not found!")

goal = defaultdict(list)
addEdge('A', 'B')
addEdge('A', 'C')
addEdge('A', 'D')
addEdge('B', 'E')
addEdge('B', 'F')
addEdge('E', 'I')
addEdge('E', 'J')
addEdge('D', 'G')
addEdge('D', 'H')
addEdge('G', 'K')
addEdge('G', 'L')
dfid('A', 'L', 4)
```

# Best First Search

```
SuccList ={ 'S':[['A',3],['B',6],['C',5]], 'A':[['E',8],['D',9]],'B':[['G',14],['F',12]], 'C':[['H',7]], 'H':[['J',6],['I',5]],'I': [['M',2],['L',10],['K',1]]} #Graph(Tree) List

Start= input("Enter Source node >> ").upper()
Goal= input('Enter Goal node >> ').upper()
Closed = list()
SUCCESS = True
FAILURE = False
State = FAILURE


def GOALTEST(N):
    if N == Goal:
        return True
    else:
        return False

def MOVEGEN(N):
    New_list=list()
    if N in SuccList.keys():
          New_list=SuccList[N]

    return New_list

def APPEND(L1,L2):
    New_list=list(L1)+list(L2)
    return New_list

def SORT(L):
    L.sort(key = lambda x: x[1])
    return L
def BestFirstSearch():
    OPEN=[[Start,5]]
    CLOSED=list()
    global State
    global Closed
    i=1
    while (len(OPEN) != 0) and (State != SUCCESS):
        print("\n<<<<<<<<<<---({})--->>>>>>>>>>\n".format(i))
        N= OPEN[0]
        print("N=",N)
        del OPEN[0] #delete  the node we picked
        if GOALTEST(N[0])==True:
            State = SUCCESS
            CLOSED = APPEND(CLOSED,[N])
            print("CLOSED=",CLOSED)
        else:
            CLOSED = APPEND(CLOSED,[N])
            print("CLOSED=",CLOSED)
            CHILD = MOVEGEN(N[0])
```

```
            print("CHILD=",CHILD)
            for val in OPEN:
                if val in CHILD:
                    CHILD.remove(val)
            for val in CLOSED:
                if val in CHILD:
                    CHILD.remove(val)
            OPEN = APPEND(CHILD,OPEN)
            print("Unsorted OPEN=",OPEN)
            SORT(OPEN)
            print("Sorted OPEN=",OPEN)
            Closed=CLOSED
            i+=1
    return State
result=BestFirstSearch()
print("Best First Search Path >>>> {} <<<{}>>>".format(Closed, result))
```

# Single Layer Perceptron

```
def OR():
    w1=0;w2=0;a=0.2;t=0
    X=[[0,0],[0,1],[1,0],[1,1]]
    Y=[0,1,1,1]
    while(True):
        Out=[]
        count = 0
        for i in X:
            step=(w1*i[0]+w2*i[1])
            if step<=t:
                O=0
                if O==Y[count]:
                    Out.append(O)
                    count+=1
                else:
                    w1=w1+(a*i[0]*1)
                    w2=w2+(a*i[1]*1)
                    print(w1,w2)
            else:
                O=1
                if O==Y[count]:
                    Out.append(O)
                    count+=1
                else:
                    w1 = w1 + (a * i[0] * 0)
                    w2 = w2 + (a * i[1] * 0)
                    print(w1,w2)
        print(" ------>")
        if Out[0:]==Y[0:]:
            print("Final Output of OR ::\n")
            print("Weights: w1={} and w2={} >>>> {}".format(w1,w2,Out))
            break
OR()
#AND
def AND():
    w1=0;w2=0;a=0.2;t=1
    X=[[0,0],[0,1],[1,0],[1,1]]
    Y=[0,0,0,1]
    while(True):
        Out=[]
        count = 0
        for i in X:
            step=(w1*i[0]+w2*i[1])
            if step<=t:
                O=0
                if O==Y[count]:
                    Out.append(O)
                    count+=1
                    print(w1,w2,Out)
                else:
                    print('Weights changed to..')
                    w1=w1+(a*i[0]*1)
                    w2=w2+(a*i[1]*1)
                    print("w1={} w2={}".format(round(w1,2),round(w2,2)))
                    print(" ------>")
            else:
                O=1
                if O==Y[count]:
```

```
                Out.append(O)
                count+=1
                print(w1,w2,Out)
            else:
                print("Weights Changed to..")
                w1 = w1 + (a * i[0] * 0)
                w2 = w2 + (a * i[1] * 0)
                print("w1={} w2={}".format(round(w1,2),round(w2,2)))
                print(" ------>")
        if Out[0:]==Y[0:]:
            print("\nFinal Output of AND::\n")
            print("Weights: w1={} and w2={} >>>> {}".format(round(w1,2),round(w2,2),Out))
            break
AND()
#NOT
def NOT():
    X=[0,1]
    Y=[1,0]
    weight=-1
    bias=1;Out=[]
    for i in X:
        j=weight*i+bias
        Out.append(j)
    print("\nFinal Output of NOT ::\n")

    for i in X:
        print("NOT Gate {}-->{}".format(X[i],Out[i]))
NOT()
```

**'This is termwork 4**

```
def minmax(node, tree, is_max_node):
    if len(tree[node][0]) == 0:
        print(f"Leaf node {node} has value {tree[node][1]}")
        return tree[node][1]
    children = tree[node][0]
    if is_max_node:
        value = -float('inf')
        print(f"Max node {node} with children {children}")
        for child in children:
            value = max(value, minmax(child, tree, False))
    else:
        value = float('inf')
        print(f"Min node {node} with children {children}")
        for child in children:
            value = min(value, minmax(child, tree, True))
    tree[node][1] = value
    print(f"Node {node} computed value {value}")
    return value
tree = {
    'A': [['B', 'C'], None],
    'B': [['D', 'E'], None],
    'C': [['F', 'G'], None],
    'D': [['H', 'I'], None],
    'E': [['J', 'K'], None],
    'F': [['L', 'M'], None],
    'G': [['N', 'O'], None],
    'H': [[], -1],
    'I': [[], 4],
    'J': [[], 3],
    'K': [[], 6],
    'L': [[], -3],
    'M': [[], -5],
```

```
    'N': [[], 0],
    'O': [[], 7]
}
result = minmax('A', tree, True)
print(f"\nThe most appropriate value for the root node 'A' is {result}")
def convert_tree(node_key, tree):
    node = {
        "value": f"{node_key} ({tree[node_key][1]})",
        "children": [convert_tree(child, tree) for child in tree[node_key][0]]
    }
    return node
converted_tree = convert_tree('A', tree)
def print_tree(node, prefix="", is_tail=True):
    print(prefix + ("└── " if is_tail else "├── ") + str(node["value"]))
    for i, child in enumerate(node["children"]):
        print_tree(child, prefix + ("    " if is_tail else "│   "),
                   i == len(node["children"]) - 1)
print("\nFinal tree with updated values:")
print_tree(converted_tree)
```

# Back Propagation (Multilayer Perceptron)

```
import numpy as np
def sigmoid (x):
   return 1/(1 + np.exp(-x))
def sigmoid_derivative(x):
   return x * (1 - x)
#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])
epochs = 10000
lr = 0.5
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1
#Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
hidden_bias =np.random.uniform(size=(1,hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))
print("Initial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)
for _ in range(epochs):

   hidden_layer_activation = np.dot(inputs,hidden_weights)
   hidden_layer_activation += hidden_bias
   hidden_layer_output = sigmoid(hidden_layer_activation)
   output_layer_activation =np.dot(hidden_layer_output,output_weights)
   output_layer_activation += output_bias
   predicted_output = sigmoid(output_layer_activation)
```

```
    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)
    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)
    output_weights +=hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True)* lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) *lr
print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)
print("\nOutput from neural network after epochs :" +str(epochs) )
print(*predicted_output)
```

# Hebbian Learning

```
x1=[1,1]
x2=[1,-1]
x3=[-1,1]
x4=[-1,-1]
xilist=[x1,x2,x3,x4]
y=[1,-1,-1,-1]
w1=w2=bw=0
b=1
def heb_learn():
    global w1,w2,bw
    print("dw1\tdw2\tdb\tw1\tw2\tb")
    i=0
    for xi in xilist:
        dw1=xi[0]*y[i]
        dw2=xi[1]*y[i]
        db=y[i]
        w1=w1+dw1
        w2=w2+dw2
        bw+=db
        print(dw1,dw2,db,w1,w2,bw,sep='\t')
        i+=1
print("Learning...")
heb_learn()
print("Learning completed")
print("Output of AND gate using obtained w1,w2,bw:")
print("x1\tx2\ty")
for xi in xilist:
    print(xi[0],xi[1],1 if w1*xi[0]+w2*xi[1]+b*bw>0 else -1,sep='\t')
print("Final weights are: w1="+str(w1) +" w2=" +str(w2))
```

# Find S Algorithm

```
#implementation of Find S algorithm
import csv
a = []
with open('enjoysport.csv', 'r') as csvfile:
    next(csvfile)
    for row in csv.reader(csvfile):
        a.append(row)
    print(a)

print("\nThe total number of training instances are : ",len(a))

num_attribute = len(a[0])-1

print("\nThe initial hypothesis is : ")
hypothesis = ['0']*num_attribute
print(hypothesis)

for i in range(0, len(a)):
    if a[i][num_attribute] == 'yes':
        print ("\nInstance ", i+1, "is", a[i], " and is Positive Instance")
        for j in range(0, num_attribute):
            if hypothesis[j] == '0' or hypothesis[j] == a[i][j]:
                hypothesis[j] = a[i][j]
            else:
                hypothesis[j] = '?'
        print("The hypothesis for the training instance", i+1, " is: " , hypothesis, "\n")

    if a[i][num_attribute] == 'no':
        print ("\nInstance ", i+1, "is", a[i], " and is Negative Instance Hence Ignored")
        print("The hypothesis for the training instance", i+1, " is: " , hypothesis, "\n")

print("\nThe Maximally specific hypothesis for the training instance is ", hypothesis
```

**k means**

```
import csv
from math import sqrt
import random
from collections import defaultdict


def euclid(p1, p2):
    a = (p1[0] - p2[0])**2
    b = (p1[1] - p2[1])**2
    return sqrt(a + b)


def initialize_centroids(points, k):
    return random.sample(points, k)


def closest_centroid(point, centroids):
    closest = centroids[0]
    mind = euclid(point, centroids[0])
    for c in centroids[1:]:
        d = euclid(point, c)
        if d < mind:
            mind, closest = d, c
    return closest


def assign_clusters(points, centroids):
    clusters = defaultdict(list)
    for point in points:
        closest = closest_centroid(point, centroids)
        clusters[tuple(closest)].append(point)
    return clusters


def update_centroids(clusters):
    new_centroids = []
    for c, points in clusters.items():
        if points:
            new_centroid = [sum(p[dim] for p in points) / len(points)
                            for dim in range(len(points[0]))]
            new_centroids.append(new_centroid)
        else:
            new_centroids.append(list(c))
    return new_centroids


def has_converged(old, new, tol=1e-4):
    for i in range(len(old)):
        if euclid(old[i], new[i]) > tol:
            return False
    return True


def print_d(centroids, clusters):
    print('Centroids:', centroids)
    for centroid, cluster_points in clusters.items():
        print(f"Cluster around centroid {centroid}: ")
        for p in cluster_points:
            print(p)
        print('-----------------------')


def k_means(filename, k, max_iterations=5, tol=1e-4):
    points = []
    with open(filename, 'r') as file:
```

```
        reader = csv.reader(file)
        for row in reader:
            points.append([float(value) for value in row])

    centroids = initialize_centroids(points, k)

    for _ in range(max_iterations):
        clusters = assign_clusters(points, centroids)
        new_centroids = update_centroids(clusters)
        print("********\n")
        print_d(centroids, clusters)

        if has_converged(centroids, new_centroids, tol):
            break

        centroids = new_centroids

    return centroids, clusters


centroids, clusters = k_means('dataset.csv', 2)
print("********\n")
print_d(centroids, clusters)
```

**A star**

```
import heapq

def astar(graph, start, goal):
    open_list = [(0, start)]
    parents = {}
    g_values = {node: float('inf') for node in graph}
    g_values[start] = 0
    f_values = {node: float('inf') for node in graph}
    f_values[start] = graph[start][1]
    iteration = 0
    while open_list:
        current_f, current_node = heapq.heappop(open_list)
        if current_node == goal:
            path = []
            while current_node in parents:
                path.append(current_node)
                current_node = parents[current_node]
            path.append(start)
            final_cost = g_values[goal]
            print(f"\nFinal Cost: {final_cost}")
            return path[::-1]
        for child, cost in graph[current_node][0].items():
            tentative_g = g_values[current_node] + cost
            if tentative_g < g_values[child]:
                parents[child] = current_node
                g_values[child] = tentative_g
                f_values[child] = tentative_g + graph[child][1]


                heapq.heappush(open_list, (f_values[child], child))
        iteration += 1
        print(f"\nIteration {iteration}:")
        print("Current Path:", reconstruct_path(parents, start, current_node))
        print(f"Evaluation Function Value for {
           current_node}: {f_values[current_node]}")
```

```
def reconstruct_path(parents, start, goal):
    path = [goal]
    while goal != start:
        goal = parents[goal]
        path.append(goal)
    return path[::-1]
start_node = 'A'
goal_node = 'G'
graph = {
    'A': [{'B': 5, 'C': 10}, 10],
    'B': [{'D': 5, 'E': 5}, 7],
    'C': [{'F': 5}, 7],
    'D': [{'G': 10}, 3],
    'E': [{'G': 7}, 2],
    'F': [{'G': 8}, 1],
    'G': [{}, 0]
}

print("\nA* Search Path:")
path = astar(graph, start_node, goal_node)
print("Final Path:", path)
```

Navy

```
import csv
import numpy as np
def load_data(file_path):
    data = []
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            data.append([float(value) for value in row])
    data = np.array(data)
    return data[:, :-1], data[:, -1]
def calculate_priors(y):
    classes, counts = np.unique(y, return_counts=True)
    priors = dict(zip(classes, counts / len(y)))
    return priors
def calculate_conditional_probabilities(X, y):
    conditional_probs = {}
    for cls in np.unique(y):
        X_cls = X[y == cls]
        feature_probs = []
        for col in range(X.shape[1]):
            values, counts = np.unique(X_cls[:, col], return_counts=True)
            probs = dict(zip(values, counts / len(X_cls)))
            feature_probs.append(probs)
        conditional_probs[cls] = feature_probs
    return conditional_probs
def classify(sample, priors, conditional_probs):
    probabilities = {}
    for cls in priors:
        prior = priors[cls]
        likelihood = 1.0
        for col, value in enumerate(sample):
            feature_probs = conditional_probs[cls][col]
            likelihood *= feature_probs.get(value, 1e-6)
```

```python
        probabilities[cls] = prior * likelihood
    return max(probabilities, key=probabilities.get), probabilities
def evaluate(X, y, priors, conditional_probs):
    predictions = [classify(sample, priors, conditional_probs)[0]
                   for sample in X]
    accuracy = np.mean(predictions == y)
    return accuracy
def predict(input_features, priors, conditional_probs):
    input_array = np.array(input_features, dtype=float)
    prediction, probabilities = classify(
        input_array, priors, conditional_probs)
    for cls in probabilities:
        print(f'P({cls}) = {probabilities[cls]:.6f}')
    return prediction
def main(file_path):
    X, y = load_data(file_path)
    X_discrete = np.floor(X)  # Simple discretization, adjust as needed
    priors = calculate_priors(y)
    conditional_probs = calculate_conditional_probabilities(X_discrete, y)
    accuracy = evaluate(X_discrete, y, priors, conditional_probs)
    print(f'Accuracy: {accuracy * 100:.2f}%')
    while True:
        try:
            input_str = input(
                "\nEnter new sample (comma-separated values) or 'exit' to quit: ")
            if input_str.lower() == 'exit':
                break
            input_features = [float(x) for x in input_str.split(',')]
            if len(input_features) != X.shape[1]:
                print(f"Please enter exactly {X.shape[1]} feature values.")
                continue
            input_features_discrete = np.floor(input_features)
            prediction = predict(input_features_discrete,
                                 priors, conditional_probs)
```

```
            print(f'Predicted class: {prediction}')
        except ValueError:
            print("Invalid input. Please enter valid numbers.")
if _name_ == "_main_":
    main('naive_bayes_dataset.csv')
```