TMW1: Depth First Iterative Deepening Search (DFID)

```
#implementation

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)
```

OUTPUT:

```
Start node:  A Goal node:  L
```

DFID at level : 1

Path Taken : A

DFID at level : 2

Path Taken : A B C D

DFID at level : 3

Path Taken : A B E F C D G H

DFID at level : 4

Path Taken : A B E I J F C D G K L

Goal node found!

TMW2: Best First Search

```
#Implementation of 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]]} #Gra
ph(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) #append movegen elements to OPEN
            print("Unsorted OPEN=",OPEN)
            SORT(OPEN)
            print("Sorted OPEN=",OPEN)
            Closed=CLOSED
            i+=1
    return State
#code by <<<Sahil Gaonkar>>>
result=BestFirstSearch()
print("Best First Search Path >>>> {} <<<{}>>>".format(Closed, result))
```

OUTPUT:

```
Enter Source node >> S
Enter Goal node >> G

<<<<<<<<<<---(1)--->>>>>>>>>>

N= ['S', 5]
CLOSED= [['S', 5]]
CHILD= [['A', 3], ['B', 6], ['C', 5]]
Unsorted OPEN= [['A', 3], ['B', 6], ['C', 5]]
Sorted OPEN= [['A', 3], ['C', 5], ['B', 6]]

<<<<<<<<<<---(2)--->>>>>>>>>>

N= ['A', 3]
CLOSED= [['S', 5], ['A', 3]]
CHILD= [['E', 8], ['D', 9]]
Unsorted OPEN= [['E', 8], ['D', 9], ['C', 5], ['B', 6]]
Sorted OPEN= [['C', 5], ['B', 6], ['E', 8], ['D', 9]]

<<<<<<<<<<---(3)--->>>>>>>>>>

N= ['C', 5]
CLOSED= [['S', 5], ['A', 3], ['C', 5]]
CHILD= [['H', 7]]
Unsorted OPEN= [['H', 7], ['B', 6], ['E', 8], ['D', 9]]
Sorted OPEN= [['B', 6], ['H', 7], ['E', 8], ['D', 9]]
```

```
<<<<<<<<<<---(4)--->>>>>>>>>>

N= ['B', 6]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6]]
CHILD= [['G', 14], ['F', 12]]
Unsorted OPEN= [['G', 14], ['F', 12], ['H', 7], ['E', 8], ['D', 9]]
Sorted OPEN= [['H', 7], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]

<<<<<<<<<<---(5)--->>>>>>>>>>

N= ['H', 7]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7]]
CHILD= [['J', 6], ['I', 5]]
Unsorted OPEN= [['J', 6], ['I', 5], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]
Sorted OPEN= [['I', 5], ['J', 6], ['E', 8], ['D', 9], ['F', 12], ['G', 14]]

<<<<<<<<<<---(6)--->>>>>>>>>>

N= ['I', 5]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5]]
CHILD= [['M', 2], ['L', 10], ['K', 1]]
Unsorted OPEN= [['M', 2], ['L', 10], ['K', 1], ['J', 6], ['E', 8], ['D', 9], ['F',
12], ['G', 14]]
Sorted OPEN= [['K', 1], ['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F',
12], ['G', 14]]

<<<<<<<<<<---(7)--->>>>>>>>>>

N= ['K', 1]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1]]
CHILD= []
Unsorted OPEN= [['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12],
['G', 14]]
Sorted OPEN= [['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G',
14]]

<<<<<<<<<<---(8)--->>>>>>>>>>

N= ['M', 2]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2]]
CHILD= []
Unsorted OPEN= [['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]
Sorted OPEN= [['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]

<<<<<<<<<<---(9)--->>>>>>>>>>

N= ['J', 6]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6]]
```

```
CHILD= []
Unsorted OPEN= [['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]
Sorted OPEN= [['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]

<<<<<<<<<<---(10)--->>>>>>>>>>

N= ['E', 8]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6], ['E', 8]]
CHILD= []
Unsorted OPEN= [['D', 9], ['L', 10], ['F', 12], ['G', 14]]
Sorted OPEN= [['D', 9], ['L', 10], ['F', 12], ['G', 14]]

<<<<<<<<<<---(11)--->>>>>>>>>>

N= ['D', 9]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6], ['E', 8], ['D', 9]]
CHILD= []
Unsorted OPEN= [['L', 10], ['F', 12], ['G', 14]]
Sorted OPEN= [['L', 10], ['F', 12], ['G', 14]]

<<<<<<<<<<---(12)--->>>>>>>>>>

N= ['L', 10]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10]]
CHILD= []
Unsorted OPEN= [['F', 12], ['G', 14]]
Sorted OPEN= [['F', 12], ['G', 14]]

<<<<<<<<<<---(13)--->>>>>>>>>>

N= ['F', 12]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12]]
CHILD= []
Unsorted OPEN= [['G', 14]]
Sorted OPEN= [['G', 14]]

<<<<<<<<<<---(14)--->>>>>>>>>>

N= ['G', 14]
CLOSED= [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7], ['I', 5], ['K', 1],
['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12], ['G', 14]]
Best First Search Path >>>> [['S', 5], ['A', 3], ['C', 5], ['B', 6], ['H', 7],
['I', 5], ['K', 1], ['M', 2], ['J', 6], ['E', 8], ['D', 9], ['L', 10], ['F', 12]]
<<<True>>>
```

TMW3: Single Layer Perceptron

```
#OR
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()
```

OUTPUT:

Weights: w1=0.2 and w2=0.2 >>>> [0, 1, 1, 1]

0 0 [0]

0 0 [0, 0]

0 0 [0, 0, 0]

Weights changed to..

w1=0.2 w2=0.2

------->

0.2 0.2 [0]

0.2 0.2 [0, 0]

0.2 0.2 [0, 0, 0]

Weights changed to..

w1=0.4 w2=0.4

------->

0.4 0.4 [0]

0.4 0.4 [0, 0]

0.4 0.4 [0, 0, 0]

Weights changed to..

w1=0.6 w2=0.6

------->

0.6000000000000001 0.6000000000000001 [0]

0.6000000000000001 0.6000000000000001 [0, 0]

0.6000000000000001 0.6000000000000001 [0, 0, 0]

0.6000000000000001 0.6000000000000001 [0, 0, 0, 1]

Final Output of AND::

Weights: w1=0.6 and w2=0.6 >>>> [0, 0, 0, 1]

Final Output of NOT ::

NOT Gate 0-->1

NOT Gate 1-->0

TMW 4: Back Propagation (Multilayer Perceptron)

```
import numpy as np
#np.random.seed(0)
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)
#Training algorithm
for _ in range(epochs):
#Forward Propagation
  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)
#Backpropagation
    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)
#Updating Weights and Biases
    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)
```

Output: After epoch 1

Initial hidden weights: [0.57739373 0.99731969] [0.23542431 0.76683569]

Initial hidden biases: [0.37407026 0.18114935]

Initial output weights: [0.0218607] [0.07345263]

Initial output biases: [0.04597635]

Final hidden weights: [0.57739202 0.9975624] [0.23545824 0.76717274]

Final hidden bias: [0.37401636 0.18106946]

Final output weights: [0.01274522] [0.06705193]

Final output bias: [0.03174794]

Output from neural network after epochs :1

[0.52472264] [0.52823899] [0.52944441] [0.53170537]

Output: After epoch 10,000

Initial hidden weights: [0.47929016 0.6120291] [0.37177763 0.62697496]

Initial hidden biases: [0.61356687 0.829318]

Initial output weights: [0.9328808] [0.31158112]

Initial output biases: [0.89154856]

Final hidden weights: [6.5248696 4.54422991] [6.52760347 4.54486153]

Final hidden bias: [-2.90354426 -6.97547132]

Final output weights: [9.58101852] [-10.31837822]

Final output bias: [-4.41780888]

Output from neural network after epochs :10000

[0.01927705] [0.98337029] [0.98336735] [0.01723606]

TMW5: Hebbian Learning

 # implementation

```
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))
```

Output:

```
dw1     dw2     db      w1      w2      b
1       1       1       1       1       1
-1      1       -1      0       2       0
1       -1      -1      1       1       -1
1       1       -1      2       2       -2
Learning completed
Output of AND gate using obtained w1,w2,bw:
x1      x2      y
1       1       1
1       -1      -1
-1      1       -1
-1      -1      -1
Final weights are: w1=2 w2=2
```