PROGRAM 1 (A*)

```
class Node():
    """A node class for A* Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position

def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the
given end in the given maze"""
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    # Initialize both open and closed list
    open_list = []
    closed_list = []
    # Add the start node
    open_list.append(start_node)
    # Loop until you find the end
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)
        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path
        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -
```

```
1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares
            # Get node position
            node_position = (current_node.position[0] +
new_position[0], current_node.position[1] + new_position[1])
            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0]
< 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or
node_position[1] < 0:
                continue
            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:

                continue
            # Create new node
            new_node = Node(current_node, node_position)
            # Append
            children.append(new_node)
        # Loop through children
        for child in children:
            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue
            # Create the f, g, and h values
            child.g = current_node.g + 1
            child.h = ((child.position[0] - end_node.position[0]) **
2) + ((child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h
            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:

                    continue
            # Add the child to the open list
            open_list.append(child)


def main():
    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
    start = (0, 0)
    end = (7, 6)
```

```
    path = astar(maze, start, end)
    print(path)
if __name__ == '__main__':
    main()
```

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
```

Candidate elimation

```
import csv
a = []
print("\n The Given Training Data Set \n")

with open('ws.csv', 'r') as csvFile:
    reader = csv.reader(csvFile)
    for row in reader:
        a.append (row)
        print(row)
num_attributes = len(a[0])-1 # we don't want last col which is target
concet ( yes/no)

print("\n The initial value of hypothesis: ")
S = ['0'] * num_attributes
G = ['?'] * num_attributes
print ("\n The most specific hypothesis S0 : [0,0,0,0,0,0]\n")
print (" \n The most general hypothesis G0 : [?,?,?,?,?,?]\n")

for j in range(0,num_attributes):
       S[j] = a[0][j];

# Comparing with Remaining Training Examples of Given Data Set

print("\n Candidate Elimination algorithm  Hypotheses Version Space
Computation\n")
temp=[]

for i in range(0,len(a)):
    if a[i][num_attributes]=='Yes':
        for j in range(0,num_attributes):
            if a[i][j]!=S[j]:
                S[j]='?'

        for j in range(0,num_attributes):
            for k  in range(0,len(temp)):
                if temp[k][j] != '?' and temp[k][j] != S[j]:
                    del temp[k] #remove it if it's not matching with
the specific hypothesis
```

```
        print(" For Training Example No :{0} the hypothesis is S{0} 
".format(i+1),S)

        if (len(temp)==0):
            print(" For Training Example No :{0} the hypothesis is 
G{0} ".format(i+1),G)
        else:
            print(" For Training Example No :{0} the hypothesis is 
G{0}".format(i+1),temp)

    if a[i][num_attributes]=='No':
        for j in range(0,num_attributes):
             if S[j] != a[i][j] and S[j]!= '?':  #if not  matching 
with the specific Hypothesis take it seperately and store it
                  G[j]=S[j]
                  temp.append(G) # this is the version space to store 
all Hypotheses
                  G = ['?'] * num_attributes

        print(" For Training Example No :{0} the hypothesis is S{0} 
".format(i+1),S)
        print(" For Training Example No :{0} the hypothesis is 
G{0}".format(i+1),temp)
```

```
 The Given Training Data Set

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes']
['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No']
['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']

 The initial value of hypothesis:

 The most specific hypothesis S0 : [0,0,0,0,0,0]


 The most general hypothesis G0 : [?,?,?,?,?,?]


 Candidate Elimination algorithm  Hypotheses Version Space Computation

 For Training Example No :1 the hypothesis is S1   ['Sunny', 'Warm', 
'Normal', 'Strong', 'Warm', 'Same']
 For Training Example No :1 the hypothesis is G1  ['?', '?', '?', '?', 
'?', '?']
 For Training Example No :2 the hypothesis is S2   ['Sunny', 'Warm', 
'?', 'Strong', 'Warm', 'Same']
 For Training Example No :2 the hypothesis is G2  ['?', '?', '?', '?', 
```

```
'?', '?']
 For Training Example No :3 the hypothesis is S3  ['Sunny', 'Warm', 
'?', 'Strong', 'Warm', 'Same']
 For Training Example No :3 the hypothesis is G3 [['Sunny', '?', '?', 
'?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', 
'?', '?', 'Same']]
 For Training Example No :4 the hypothesis is S4   ['Sunny', 'Warm', 
'?', 'Strong', '?', '?']
 For Training Example No :4 the hypothesis is G4 [['Sunny', '?', '?', 
'?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

decision tree id3

```
import sys
import numpy as np
from numpy import *
import csv

class Node:
    def __init__(self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""

def read_data(filename):
    """ read csv file and return header and data  """
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        metadata = next(datareader)
        traindata=[]
        for row in datareader:
            traindata.append(row)

    return (metadata, traindata)



def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col]) # get unique values in a 
particular column

    count = np.zeros((items.shape[0], 1), dtype=np.int32)   #number of 
row = number of values

    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1
```

```python
    #count has the data of number of times each value is present in

    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]),
dtype="|S32")

        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1

        if delete:
           dict[items[x]] = np.delete(dict[items[x]], col, 1)
    return items, dict



def entropy(S):
    """ calculate the entropy """
    items = np.unique(S)
    if items.size == 1:
        return 0

    counts = np.zeros((items.shape[0], 1))
    sums = 0

    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size)

    for count in counts:
        sums += -1 * count * math.log(count, 2)
    return sums

def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)
    #item is the unique value and dict is the data corresponding to it
    total_size = data.shape[0]
    entropies = np.zeros((items.shape[0], 1))

    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0]/(total_size)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])


    total_entropy = entropy(data[:, -1])


    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]
```

```
    return total_entropy


def create_node(data, metadata):

    if (np.unique(data[:, -1])).shape[0] == 1: #to check how many rows
in last col(yes,no column). shape[0] gives no. of rows
        ''' if there is only yes or only no then reutrn a node
containing the value '''
        node = Node("")
        node.answer = np.unique(data[:, -1])
        return node

    gains = np.zeros((data.shape[1] - 1, 1))  # data.shape[1] - 1
returns the no of columns in the dataset, minus one to remove last
column
    #size of gains= number of attribute to calculate gain
    #gains is one dim array (size=4) to store the gain of each
attribute

    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)

    split = np.argmax(gains) # argmax returns the index of the max
value


    node = Node(metadata[split])
    metadata = np.delete(metadata, split, 0)


    items, dict = subtables(data, split, delete=True)

    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))

    return node

def empty(size):
    """ To generate empty space needed for shaping the tree"""
    s = ""
    for x in range(size):
        s += "   "
    return s

def print_tree(node, level):
    if node.answer != "":
```

```
        print(empty(level), node.answer.item(0).decode("utf-8"))
        return

    print(empty(level), node.attribute)

    for value, n in node.children:
        print(empty(level + 1), value.tobytes().decode("utf-8"))
        print_tree(n, level + 2)


metadata, traindata = read_data("tennis.csv")
data = np.array(traindata) # to convert the traindata to numpy array
node = create_node(data, metadata)
print_tree(node, 0)
```

```
 outlook
    overcast
       yes
    rainy
       windy
          Strong
             no
          Weak
             yes
    sunny
       humidity
          high
             no
          normal
             yes
```

ANN

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6])) # Hours Studied, Hours Slept
y = np.array(([92], [86], [89])) # Test Score

y = y/100 # max test score is 100


#Sigmoid Function
def sigmoid(x): #this function maps any value between 0 and 1
    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
```

```
epoch=1 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons of output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bias_hidden=np.random.uniform(size=(1,hiddenlayer_neurons)) #bias
matrix to the hidden layer
weight_hidden=np.random.uniform(size=(hiddenlayer_neurons,output_neuro
ns)) #weight matrix to the output layer
bias_output=np.random.uniform(size=(1,output_neurons)) # matrix to the
output layer
print(weight_hidden,"W")
print(weight_hidden.T,"WT")

for i in range(epoch):
    #Forward Propogation
    hinp1=np.dot(X,wh)
    hinp= hinp1 + bias_hidden #bias_hidden GRADIENT DISCENT
    hlayer_activation = sigmoid(hinp)

    outinp1=np.dot(hlayer_activation,weight_hidden)
    outinp= outinp1+ bias_output
    output = sigmoid(outinp)
    print(output,"output")
    #Backpropagation
    EO = y-output #Compare prediction with actual output and calculate
the gradient of error (Actual – Predicted)

    outgrad = derivatives_sigmoid(output) #Compute the slope/ gradient
of hidden and output layer neurons

    d_output = EO * outgrad #Compute change factor(delta) at output
layer, dependent on the gradient of error multiplied by the slope of
output layer activation
#     print(weight_hidden,weight_hidden.T,"T")
    EH = d_output.dot(weight_hidden.T)  #At this step, the error will
propagate back into the network which means error at hidden layer. we
will take the dot product of output layer delta with weight parameters
of edges between the hidden and output layer (weight_hidden.T).

    hiddengrad = derivatives_sigmoid(hlayer_activation) #how much
hidden layer weight contributed to error
    d_hiddenlayer = EH * hiddengrad


    #update the weights
    weight_hidden += hlayer_activation.T.dot(d_output) *lr# dot
```

```
product of nextlayererror and currentlayerop
    bias_hidden += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr

    wh += X.T.dot(d_hiddenlayer) *lr
    bias_output += np.sum(d_output, axis=0,keepdims=True) *lr

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

```
[[0.01519819]
 [0.95510313]
 [0.91549755]] W
[[0.01519819 0.95510313 0.91549755]] WT
[[0.9406462 ]
 [0.93398837]
 [0.94082293]] output
Input:
[[2 9]
 [1 5]
 [3 6]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.9406462 ]
 [0.93398837]
 [0.94082293]]
```

Navie Bias

```
import numpy as np
import math
import csv
import pdb
def read_data(filename):
    with open(filename,'r') as csvfile:
        datareader = csv.reader(csvfile)
        metadata = next(datareader)
        traindata=[]
        for row in datareader:
            traindata.append(row)

    return (metadata, traindata)

def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    testset = list(dataset)
```

```
    i=0
    while len(trainSet) < trainSize:
        trainSet.append(testset.pop(i))
    print(trainSet)
    return [trainSet, testset]

def classify(data,test):

    total_size = data.shape[0]
    print("training data size=",total_size)
    print("test data size=",test.shape[0])

    countYes = 0
    countNo = 0
    probYes = 0
    probNo = 0
    print("target    count    probability")

    for x in range(data.shape[0]):
        if data[x,data.shape[1]-1] == 'yes':
            countYes +=1
        if data[x,data.shape[1]-1] == 'no':
            countNo +=1

    probYes=countYes/total_size
    probNo= countNo / total_size

    print('Yes',"\t",countYes,"\t",probYes)
    print('No',"\t",countNo,"\t",probNo)


    prob0 =np.zeros((test.shape[1]-1))
    prob1 =np.zeros((test.shape[1]-1))
    accuracy=0
    print("instance prediction  target")

    for t in range(test.shape[0]):
        for k in range (test.shape[1]-1):
            count1=count0=0
            for j in range (data.shape[0]):
                #how many times appeared with no
                if test[t,k] == data[j,k] and data[j,data.shape[1]-
1]=='no':
                    count0+=1
                #how many times appeared with yes
                if test[t,k]==data[j,k] and data[j,data.shape[1]-
1]=='yes':
                    count1+=1
            prob0[k]=count0/countNo
```

```
            prob1[k]=count1/countYes

        probno=probNo
        probyes=probYes
        for i in range(test.shape[1]-1):
            probno=probno*prob0[i]
            probyes=probyes*prob1[i]
        if probno>probyes:
            predict='no'
        else:
            predict='yes'

        print(t+1,"\t",predict,"\t    ",test[t,test.shape[1]-1])
        if predict == test[t,test.shape[1]-1]:
            accuracy+=1
    final_accuracy=(accuracy/test.shape[0])*100
    print("accuracy",final_accuracy,"%")
    return

metadata,traindata= read_data("tennis1.csv")
splitRatio=0.6
#split into training and testing
trainingset, testset=splitDataset(traindata, splitRatio)
training=np.array(trainingset)
testing=np.array(testset)

classify(training,testing)
```

```
[['sunny', 'hot', 'high', 'Weak', 'no'], ['sunny', 'hot', 'high',
'Strong', 'no'], ['overcast', 'hot', 'high', 'Weak', 'yes'], ['rainy',
'mild', 'high', 'Weak', 'yes'], ['rainy', 'cool', 'normal', 'Weak',
'yes'], ['rainy', 'cool', 'normal', 'Strong', 'no'], ['overcast',
'cool', 'normal', 'Strong', 'yes'], ['sunny', 'mild', 'high', 'Weak',
'no']]
training data size= 8
test data size= 6
target    count    probability
Yes    4    0.5
No     4    0.5
instance prediction  target
1      no       yes
2      yes      yes
3      no       yes
4      yes      yes
5      yes      yes
6      no       no
accuracy 66.66666666666666 %
```

AO STAR

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
#instantiate graph object with graph topology, heuristic values, start
node
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self): # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v): # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v): # return the status of a given node
        return self.status.get(v,0)

    def setStatus(self,v, val): # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0) # always return the heuristic value of
a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value # set the revised heuristic value of a given
node

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START
NODE:",self.start)

print("------------------------------------------------------------")
        print(self.solutionGraph)

print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v): # Computes the Minimum
Cost of child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v): # iterate over
all the set of child node/s
```

```
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            if flag==True: # initialize Minimum Cost with the cost of 
first set of child node/s
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList # set 
the Minimum Cost child node/s
                flag=False
            else: # checking the Minimum Cost nodes with the current 
Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList # 
set the Minimum Cost child node/s
        return minimumCost, costToChildNodeListDict[minimumCost] # 
return Minimum Cost and Minimum Cost child node/s

    def aoStar(self, v, backTracking): # AO* algorithm for a start 
node and backTracking status flag
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)

print("-----------------------------------------------------------
-------------------------")
        if self.getStatus(v) >= 0: # if status node v >= 0, compute 
Minimum Cost nodes of v
            minimumCost, childNodeList = 
self.computeMinimumCostChildNodes(v)
            print(minimumCost, childNodeList)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True # check the Minimum Cost nodes of v are solved
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False
            if solved==True: # if the Minimum Cost nodes of v are 
solved, set the current node status as solved(-1)
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList # update the 
solution graph with the solved nodes which may be a part of solution
            if v!=self.start: # check the current node is the start 
node for backtracking the current node value
                self.aoStar(self.parent[v], True) # backtracking the 
current node value with backtracking status set to true
            if backTracking==False: # check the current call is not
```

```
for backtracking
                for childNode in childNodeList: # for each Minimum
Cost child node
                    self.setStatus(childNode,0) # set the status of
child node to 0(needs exploration)
                    self.aoStar(childNode, False) # Minimum Cost child
node is further explored with backtracking status as false
# print ("Graph - 1")
# h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H':
7, 'I': 7, 'J': 1}
# graph1 = {
#     'A': [[('B', 1), ('C', 1)], [('D', 1)]],
#     'B': [[('G', 1)], [('H', 1)]],
#     'C': [[('J', 1)]],
#     'D': [[('E', 1), ('F', 1)]],
#     'G': [[('I', 1)]]
# }

# G1= Graph(graph1, h1, 'A')
# G1.applyAOStar()
# G1.printSolution()


print ("Graph - 2")
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H':
7} # Heuristic values of Nodes
graph2 = { # Graph of Nodes and Edges
    'A': [[('B', 1), ('C', 1)], [('D', 1)]], # Neighbors of Node 'A',
B, C & D with repective weights
    'B': [[('G', 1)], [('H', 1)]], # Neighbors are included in a list
of lists
    'D': [[('E', 1), ('F', 1)]] # Each sublist indicate a "OR" node or
"AND" nodes
}

G2 = Graph(graph2, h2, 'A') # Instantiate Graph object with graph,
heuristic values and start Node
G2.applyAOStar() # Run the AO* algorithm
G2.printSolution() # Print the solution graph as output of the AO*
algorithm search
```

```
Graph - 2
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
----------------------------------------------------------------------
-------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
```

```
'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : D
----------------------------------------------------------------------
-------------------
10 ['E', 'F']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
----------------------------------------------------------------------
-------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : E
----------------------------------------------------------------------
-------------------
0 []
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : D
----------------------------------------------------------------------
-------------------
6 ['E', 'F']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : A
----------------------------------------------------------------------
-------------------
7 ['D']
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4,
'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : F
----------------------------------------------------------------------
-------------------
0 []
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0,
'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': [], 'F': []}
PROCESSING NODE : D
----------------------------------------------------------------------
-------------------
2 ['E', 'F']
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 2, 'E': 0, 'F': 0,
'G': 5, 'H': 7}
```

```
SOLUTION GRAPH : {'E': [], 'F': [], 'D': ['E', 'F']}
PROCESSING NODE : A
----------------------------------------------------------------------
-------------------
3 ['D']
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
------------------------------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
------------------------------------------------------------
```

K nearest neighbours

```
from sklearn import datasets
iris=datasets.load_iris()
iris_data=iris.data
iris_labels=iris.target

from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(iris_data,iris_labels,t
est_size=0.30)


from sklearn.neighbors import KNeighborsClassifier
classifier=KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train,y_train)
y_pred=classifier.predict(x_test)

from sklearn.metrics import classification_report,confusion_matrix
from sklearn import metrics
print('Confusion matrix is as follows')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Matrics')
print(classification_report(y_test,y_pred))
print("The final accuracy score is ",
metrics.accuracy_score(y_test,classifier.predict(x_test)))
```

```
Confusion matrix is as follows
[[19  0  0]
 [ 0 13  1]
 [ 0  0 12]]
Accuracy Matrics
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        19
          1       1.00      0.93      0.96        14
          2       0.92      1.00      0.96        12

avg / total       0.98      0.98      0.98        45
```

```
The final accuracy score is  0.9777777777777777
```