**Implement A\* Search algorithm.**

def aStarAlgo(start\_node, stop\_node):

 open\_set = set(start\_node)

 closed\_set = set()

 g = {} #store distance from starting node

 parents = {} # parents contains an adjacency map of all nodes

 #ditance of starting node from itself is zero

 g[start\_node] = 0

 #start\_node is root node i.e it has no parent nodes

 #so start\_node is set to its own parent node

 parents[start\_node] = start\_node

 while len(open\_set) > 0:

 n = None

 #node with lowest f() is found

 for v in open\_set:

 if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):   n = v

 if n == stop\_node or Graph\_nodes[n] == None:

 pass

 else:

 for (m, weight) in get\_neighbors(n):

 #nodes 'm' not in first and last set are added to first   #n is set its parent

 if m not in open\_set and m not in closed\_set:   open\_set.add(m)

 parents[m] = n

 g[m] = g[n] + weight

 #for each node m,compare its distance from start i.e g(m) to the   #from start through n node

 else:

 if g[m] > g[n] + weight:

 #update g(

 #change parent of m to n

 parents[m] = n  m)

 g[m] = g[n] + weight

 #if m in closed set,remove and add to open   if m in closed\_set:

 closed\_set.remove(m)

 open\_set.add(m)

 if n == None:

 print('Path does not exist!')

 return None

 # if the current node is the stop\_node

 # then we begin reconstructin the path from it to the start\_node   if n == stop\_node:

 path = []

 while parents[n] != n:

 path.append(n)

 n = parents[n]

 path.append(start\_node)

 path.reverse()

 print('Path found: {}'.format(path))

 return path

 # remove n from the open\_list, and add it to closed\_list

  # because all of his neighbors were inspected   open\_set.remove(n)   closed\_set.add(n)

 print('Path does not exist!')

 return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

 if v in Graph\_nodes:

 return Graph\_nodes[v]

 else:

 return None

#for simplicity we ll consider heuristic distances given  #and this function returns heuristic distance for all nodes  def heuristic(n):

 H\_dist = {

 'A': 11,

 'B': 6,

 'C': 99,

 'D': 1,

 'E': 7,

 'G': 0,

 }

 return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

 'A': [('B', 2), ('E', 3)],

 'B': [('C', 1),('G', 9)],

 'C': None,

 'E': [('D', 6)],

 'D': [('G', 1)],

}

aStarAlgo('A', 'G')

OUTPUT

***Path found: ['A', 'E', 'D', 'G']***

#### PROGRAM 2

**Implement AO\* Search algorithm.**

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)

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

h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

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()

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

**OUTPUT**

HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : B

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : G

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : B

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 10, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : I

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': []}

PROCESSING NODE : G

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I']}

PROCESSING NODE : B

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 12, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : C

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : J

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}

PROCESSING NODE : C

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

------------------------------------------------------------

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

------------------------------------------------------------

HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : E

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : F

-----------------------------------------------------------------------------------------

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

-----------------------------------------------------------------------------------------

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

-----------------------------------------------------------------------------------------

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

------------------------------------------------------------

{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}

------------------------------------------------------------

#### PROGRAM 3

#### For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

**DATA SET (1finds.csv)**

Example Sky AirTemp Humidity Wind Water Forecast EnjoySport

1 Sunny Warm Normal Strong Warm Same Yes

2 Sunny Warm High Strong Warm Same Yes

3 Rainy Cold High Strong Warm Change No

4 Sunny Warm High Strong Cool Change Yes

import numpy as np

import pandas as pd

data = pd.DataFrame(data=pd.read\_csv('Finds1.csv'))

concepts = np.array(data.iloc[:,0:-1])

target = np.array(data.iloc[:,-1])

def learn(concepts, target):

specific\_h = concepts[0].copy()

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

for i,h in enumerate(concepts):

if target[i] == "Yes":

for x in range(len(specific\_h)):

if h[x] != specific\_h[x]:

specific\_h[x] = '?'

general\_h[x][x] = '?'

if target[i] == "No":

for x in range(len(specific\_h)):

if h[x] != specific\_h[x]:

general\_h[x][x] = specific\_h[x]

else:

general\_h[x][x] = '?'

indices = [i for i,val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

general\_h.remove(['?', '?', '?', '?', '?', '?'])

return specific\_h,general\_h

s\_final, g\_final = learn(concepts,target)

print("Final S:", s\_final, sep="\n")

print("Final G:", g\_final, sep="\n")

**OUTPUT**

Final S:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final G:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

#### PROGRAM 4

#### Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a newsample.

**DATASET (**tennis.csv**)**



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

#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)

**OUTPUT**

outlook

overcast

yes

rainy

windy

Strong

no

Weak

yes

sunny

humidity

high

no

normal

yes