**1. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.**

**import** csv

hypo **=** ['%','%','%','%','%','%'];

**with** open('trainingdata.csv') **as** csv\_file:

readcsv **=** csv**.**reader(csv\_file, delimiter**=**',')

print(readcsv)

data **=** []

print("\nThe given training examples are:")

**for** row **in** readcsv:

print(row)

**if** row[len(row)**-**1]**.**upper() **==** "YES":

data**.**append(row)

<\_csv.reader object at 0x0000001781862BA8>

The given training examples are:

['sky', 'airTemp', 'humidity', 'wind', 'water', 'forecast', 'enjoySport']

['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']

In [2]:

print("\nThe positive examples are:");

**for** x **in** data:

print(x);

print("\n");

**OUTPUT**:

The positive examples are:

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']

['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes']

['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']

**PROGRAM**

TotalExamples **=** len(data);

i**=**0;

j**=**0;

k**=**0;

print("The steps of the Find-s algorithm are :\n",hypo);

list **=** [];

p**=**0;

d**=**len(data[p])**-**1;

**for** j **in** range(d):

list**.**append(data[i][j]);

hypo**=**list;

i**=**1;

**for** i **in** range(TotalExamples):

**for** k **in** range(d):

**if** hypo[k]**!=**data[i][k]:

hypo[k]**=**'?';

k**=**k**+**1;

**else**:

hypo[k];

print(hypo);

i**=**i**+**1;

**OUTPUT**

The steps of the Find-s algorithm are :

['%', '%', '%', '%', '%', '%']

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'

['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']

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

In [4]:

print("\nThe maximally specific Find-s hypothesis for the given training examples is :");

list**=**[];

**for** i **in** range(d):

list**.**append(hypo[i]);

print(list);

**OUTPUT**

The maximally specific Find-s hypothesis for the given training examples is :

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

**DATABASE TABLE:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **sky** | **airTemp** | **humidity** | **wind** | **water** | **forecast** | **enjoySport** |
| 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 |

**2.Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.**

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.neighbors **import** KNeighborsClassifier

**from** sklearn.metrics **import** classification\_report

**from** sklearn.metrics **import** confusion\_matrix

**import** pandas **as** pd

**import** numpy **as** np

**from** sklearn **import** datasets

iris**=**datasets**.**load\_iris()

iris\_data**=**iris**.**data

iris\_labels**=**iris**.**target

print(iris\_data)

x\_train, x\_test, y\_train, y\_test**=**(train\_test\_split(iris\_data, iris\_labels, test\_size**=**0.20))

classifier**=**KNeighborsClassifier(n\_neighbors**=**6)

classifier**.**fit(x\_train, y\_train)

y\_pred**=**classifier**.**predict(x\_test)

print("accuracy is")

print(classification\_report(y\_test, y\_pred))

**OUTPUT**

accuracy is

precision recall f1-score support

0 1.00 1.00 1.00 9

1 1.00 0.93 0.96 14

2 0.88 1.00 0.93 7

accuracy 0.97 30

macro avg 0.96 0.98 0.97 30

weighted avg 0.97 0.97 0.97 30

**3. 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 new sample.**

**import** numpy **as** np

**import** math

**import** csv

**def** read\_data(filename):

**with** open(filename, 'r') **as** csvfile:

datareader **=** csv**.**reader(csvfile, delimiter**=**',')

headers **=** next(datareader)

metadata **=** []

traindata **=** []

**for** name **in** headers:

metadata**.**append(name)

**for** row **in** datareader:

traindata**.**append(row)

**return** (metadata, traindata)

**class** Node:

**def** \_\_init\_\_(self, attribute):

self**.**attribute **=** attribute

self**.**children **=** []

self**.**answer **=** ""

**def** \_\_str\_\_(self):

**return** self**.**attribute

**def** subtables(data, col, delete):

dict **=** {}

items **=** np**.**unique(data[:, col])

count **=** np**.**zeros((items**.**shape[0], 1), dtype**=**np**.**int32)

**for** x **in** range(items**.**shape[0]):

**for** y **in** range(data**.**shape[0]):

**if** data[y, col] **==** items[x]:

count[x] **+=** 1

**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):

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 **\*** 1.0)

**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**)

total\_size **=** data**.**shape[0]

entropies **=** np**.**zeros((items**.**shape[0], 1))

intrinsic **=** np**.**zeros((items**.**shape[0], 1))

**for** x **in** range(items**.**shape[0]):

ratio **=** dict[items[x]]**.**shape[0]**/**(total\_size **\*** 1.0)

entropies[x] **=** ratio **\*** entropy(dict[items[x]][:, **-**1])

intrinsic[x] **=** ratio **\*** math**.**log(ratio, 2)

total\_entropy **=** entropy(data[:, **-**1])

iv **=** **-**1 **\*** sum(intrinsic)

**for** x **in** range(entropies**.**shape[0]):

total\_entropy **-=** entropies[x]

**return** total\_entropy **/** iv

**def** create\_node(data, metadata):

**if** (np**.**unique(data[:, **-**1]))**.**shape[0] **==** 1:

node **=** Node("")

node**.**answer **=** np**.**unique(data[:, **-**1])[0]

**return** node

gains **=** np**.**zeros((data**.**shape[1] **-** 1, 1))

**for** col **in** range(data**.**shape[1] **-** 1):

gains[col] **=** gain\_ratio(data, col)

split **=** np**.**argmax(gains)

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):

s **=** ""

**for** x **in** range(size):

s **+=** " "

**return** s

**def** print\_tree(node, level):

**if** node**.**answer **!=** "":

print(empty(level), node**.**answer)

**return**

print(empty(level), node**.**attribute)

**for** value, n **in** node**.**children:

print(empty(level **+** 1), value)

print\_tree(n, level **+** 2)

metadata, traindata **=** read\_data("tennisdata.csv")

data **=** np**.**array(traindata)

node **=** create\_node(data, metadata)

print\_tree(node, 0)

**OUTPUT**

Outlook

Overcast

b'Yes'

Rainy

Windy

b'False'

b'Yes'

b'True'

b'No'

Sunny

Humidity

b'High'

b'No'

b'Normal'

b'Yes'

**4. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.**

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) # X = (hours sleeping, hours studying)

y = np.array(([92], [86], [89]), dtype=float) # y = score on test

# scale units

X = X/np.amax(X, axis=0) # maximum of X array

y = y/100 # max test score is 100

class Neural\_Network(object):

def \_\_init\_\_(self):

# Parameters

self.inputSize = 2

self.outputSize = 1

self.hiddenSize = 3

# Weights

self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer

self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer

def forward(self, X):

#forward propagation through our network

self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights

self.z2 = self.sigmoid(self.z) # activation function

self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights

o = self.sigmoid(self.z3) # final activation function

return o

def sigmoid(self, s):

return 1/(1+np.exp(-s)) # activation function

def sigmoidPrime(self, s):

return s \* (1 - s) # derivative of sigmoid

def backward(self, X, y, o):

# backward propgate through the network

self.o\_error = y - o # error in output

self.o\_delta = self.o\_error\*self.sigmoidPrime(o) # applying derivative of sigmoid to

self.z2\_error = self.o\_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed to output error

self.z2\_delta = self.z2\_error\*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error

self.W1 += X.T.dot(self.z2\_delta) # adjusting first set (input --> hidden) weights

self.W2 += self.z2.T.dot(self.o\_delta) # adjusting second set (hidden --> output) weights

def train (self, X, y):

o = self.forward(X)

self.backward(X, y, o)

NN = Neural\_Network()

print ("\nInput: \n" + str(X))

print ("\nActual Output: \n" + str(y))

print ("\nPredicted Output: \n" + str(NN.forward(X)))

print ("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X))))) # mean sum squared loss)

NN.train(X, y)

**5. Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.**

# import necessary libraries

import pandas as pd

from sklearn import tree

from sklearn.preprocessing import LabelEncoder

from sklearn.naive\_bayes import GaussianNB

# load data from CSV

data = pd.read\_csv('tennisdata.csv')

print("THe first 5 values of data is :\n",data.head())

# obtain Train data and Train output

X = data.iloc[:,:-1]

print("\nThe First 5 values of train data is\n",X.head())

y = data.iloc[:,-1]

print("\nThe first 5 values of Train output is\n",y.head())

# Convert then in numbers

le\_outlook = LabelEncoder()

X.Outlook = le\_outlook.fit\_transform(X.Outlook)

le\_Temperature = LabelEncoder()

X.Temperature = le\_Temperature.fit\_transform(X.Temperature)

le\_Humidity = LabelEncoder()

X.Humidity = le\_Humidity.fit\_transform(X.Humidity)

le\_Windy = LabelEncoder()

X.Windy = le\_Windy.fit\_transform(X.Windy)

print("\nNow the Train data is :\n",X.head())

le\_PlayTennis = LabelEncoder()

y = le\_PlayTennis.fit\_transform(y)

print("\nNow the Train output is\n",y)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size=0.20)

classifier = GaussianNB()

classifier.fit(X\_train,y\_train)

from sklearn.metrics import accuracy\_score

print("Accuracy is:",accuracy\_score(classifier.predict(X\_test),y\_test))

**OUTPUT**

Accuracy is: 0.6666666666666666

**DATABASE TABLE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Outlook** | **Temperature** | **Humidity** | **Windy** | **PlayTennis** |
| Sunny | Hot | High | FALSE | No |
| Sunny | Hot | High | TRUE | No |
| Overcast | Hot | High | FALSE | Yes |
| Rainy | Mild | High | FALSE | Yes |
| Rainy | Cool | Normal | FALSE | Yes |
| Rainy | Cool | Normal | TRUE | No |
| Overcast | Cool | Normal | TRUE | Yes |
| Sunny | Mild | High | FALSE | No |
| Sunny | Cool | Normal | FALSE | Yes |
| Rainy | Mild | Normal | FALSE | Yes |
| Sunny | Mild | Normal | TRUE | Yes |
| Overcast | Mild | High | TRUE | Yes |
| Overcast | Hot | Normal | FALSE | Yes |
| Rainy | Mild | High | TRUE | No |

**( naïve Bayesian Classifier (using API) )**

**5 a. Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.**

import pandas as pd

msg = pd.read\_csv('document.csv', names=['message', 'label'])

print("Total Instances of Dataset: ", msg.shape[0])

msg['labelnum'] = msg.label.map({'pos': 1, 'neg': 0})

X = msg.message

y = msg.labelnum

from sklearn.model\_selection import train\_test\_split

Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y)

from sklearn.feature\_extraction.text import CountVectorizer

count\_v = CountVectorizer()

Xtrain\_dm = count\_v.fit\_transform(Xtrain)

Xtest\_dm = count\_v.transform(Xtest)

df = pd.DataFrame(Xtrain\_dm.toarray(),columns=count\_v.get\_feature\_names())

print(df[0:5])

from sklearn.naive\_bayes import MultinomialNB

clf = MultinomialNB()

clf.fit(Xtrain\_dm, ytrain)

pred = clf.predict(Xtest\_dm)

for doc, p in zip(Xtrain, pred):

p = 'pos' if p == 1 else 'neg'

print("%s -> %s" % (doc, p))

from sklearn.metrics import accuracy\_score, confusion\_matrix, precision\_score, recall\_score

print('Accuracy Metrics: \n')

print('Accuracy: ', accuracy\_score(ytest, pred))

print('Recall: ', recall\_score(ytest, pred))

print('Precision: ', precision\_score(ytest, pred))

print('Confusion Matrix: \n', confusion\_matrix(ytest, pred))

**OUTPUT**

I am sick and tired of this place -> pos

I do not like the taste of this juice -> neg

I love this sandwich -> neg

I can't deal with this -> pos

I do not like this restaurant -> neg

**Accuracy Metrics:**

Accuracy: 0.6

Recall: 0.5

Precision: 1.0

Confusion Matrix:

[[1 0]

[2 2]]

**AI PROGRAMS**

**6. implement an AI to implement BFS and DFS search**

**BFS Implementation**

**from** queue **import** Queue

graph **=** {

5 : [3, 7],

3 : [2, 4],

7 : [8, 9],

2 : [10],

4 : [8],

8 : [],

9 : [],

10 : []

}

**def** BFS(graph, starting\_node, target\_node):

BFS\_path **=** []

visited\_nodes **=** list()

queue **=** Queue()

queue**.**put(starting\_node)

visited\_nodes**.**append(starting\_node)

**while**(**not** queue**.**empty()):

current\_node **=** queue**.**get()

BFS\_path**.**append(current\_node)

**if**(current\_node **==** target\_node):

**return** BFS\_path

**for** neighbour\_node **in** graph[current\_node]:

**if**(neighbour\_node **not** **in** visited\_nodes):

queue**.**put(neighbour\_node)

visited\_nodes**.**append(neighbour\_node)

BFS\_sNode **=** int(input("Enter the source node: "))

BFS\_tNode **=** int(input("Enter the target node: "))

BFS\_path **=** BFS(graph, BFS\_sNode, BFS\_tNode)

print(BFS\_path)

**DFS Implementation**

DFS\_path **=** []

**def** DFS(graph, starting\_node, target\_node, path, visited **=** []):

path**.**append(starting\_node)

visited**.**append(starting\_node)

**if** starting\_node **==** target\_node:

**return** path

**for** neighbour\_node **in** graph[starting\_node]:

**if** neighbour\_node **not** **in** visited:

result **=** DFS(graph, neighbour\_node, target\_node, path, visited)

**if** result **is** **not** **None**:

**return** result

path**.**pop()

**return** **None**

DFS\_path **=** []

DFS\_sNode **=** int(input("Enter the source node: "))

DFS\_tNode **=** int(input("Enter the target node: "))

DFS\_path **=** DFS(graph, DFS\_sNode, DFS\_tNode, DFS\_path)

print(DFS\_path)

**OUTPUT**

Enter the source node: 5

Enter the target node: 8

[5, 3, 4, 8]

**7. implement an AI program on 8- puzzle problem using A\*.**

Implement A\* search algorithm and apply it to 8-puzzle problem. In addition to it provide problem formulation, operators, g-cost and two heuristic functions (h-cost) of the 8-puzzle problem.

# 8- PUZZLE PROBLEM:

The 8-puzzle consists of a 3x3 (3 by 3) grid area with 8 square blocks. Every single grid with that in the puzzle is known as a tile, and there is a number ranging from 1 to 8 for each tile, so that they are uniquely recognizable. The objective is to arrange the tiles according to the order specified. The only way the blocks can move is either horizontally or vertically into a blank square.

# PROBLEM FORMULATION:

Goal: Goal State is initially given. States: Integer locations of tiles. Actions: Move the blank tile UP, DOWN, LEFT or RIGHT. Performance: Number of total moves in the solution

# A\* ALGORITHM:

A \* search algorithm is used to solve this puzzle which illustrates a general artificial intelligence methodology. It is an informed search algorithm which is used in path findings and graph traversals. It is a combination of uniform cost search (UCS) and best first search (greedy), which avoids expanding expensive paths. A\* star uses admissible heuristics which is optimal as it avoids over-estimating the path to goal state. The evaluation function A\* uses for calculating the distance is:

f(n) = g(n) + h(n) where g(n) = cost so far to reach n, h(n) = estimated cost from n to goal, f(n) = estimated total cost of path through n to goal

# Heuristic Functions:

The heuristic function is a way to inform the search regarding the direction to a goal. It provides an information to estimate which neighboring node will lead to the goal. The two heuristic functions that we considered for solving 8-puzzle problem are:

● Manhattan Distance: The distance between two tiles measured along the axes of right angles. It is the sum of absolute values of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively, i.e. |i - l|+ |j - m|

● Misplaced Tile: This calculates the number of misplaced tiles in any given state.

**Program**

**import copy**

**class Node:**

**# Initialize new node with the data**

**def \_\_init\_\_(self,data,level,hval,parent):**

**self.state = data**

**# level : path cost**

**self.level = level**

**# hval : heuristic value(heuristic 1 or heuristic 2 which is manhatten distance)**

**self.hval = hval**

**# fval : total cost = levelCost+HeuristicValue = level+ hval**

**self.fval = level+hval**

**self.parent = parent**

**self.children = []**

**# get\_position function returns position of any number in a given state**

**# But we specifically use it to find the position of the 0 in the puzzle**

**def get\_position(self,mat,num):**

**for row in range(0,len(self.state)):**

**for col in range(0,len(self.state)):**

**if mat[row][col] == num:**

**return row,col**

**# creates possible moves by moving the 0 in either left, right, up or down directions**

**def generate\_child\_nodes(self):**

**self.zero = self.get\_position(self.state,0)**

**i,j = self.get\_position(self.state,0)**

**self.child\_nodes = []**

**maxrows = 2**

**maxcols = 2**

**left = j - 1**

**if(left >= 0):**

**self.child\_nodes.append((i, left))**

**down = i + 1**

**if(down <= maxrows):**

**self.child\_nodes.append((down, j))**

**up = i - 1**

**if(up >= 0):**

**self.child\_nodes.append((up, j))**

**right = j + 1**

**if(right <= maxcols):**

**self.child\_nodes.append((i, right))**

**# Move the 0 in the given direction and if the position is out of limits the return None**

**def shuffle(self,state1,state2):**

**temp\_matrix = copy.deepcopy(self.state)**

**i1 = state1[0]**

**j1 = state1[1]**

**i2 = state2[0]**

**j2 = state2[1]**

**mx = temp\_matrix[i1][j1]**

**temp\_matrix[i1][j1] = temp\_matrix[i2][j2]**

**temp\_matrix[i2][j2] = mx**

**return temp\_matrix**

**class Astar\_8\_puzzle:**

**def \_\_init\_\_(self):**

**self.opened = []**

**self.closed = []**

**# get\_position function returns position of any number in a given state**

**# But we specifically use it to find the position of the 0 in the puzzle**

**def get\_position(self,mat,num):**

**for row in range(0,len(mat)):**

**for col in range(0,len(mat)):**

**if mat[row][col] == num:**

**return row,col**

**# takes the input for the puzzle from the user and returns the input state as a 3\*3 matrix**

**def input\_matrix(self,title):**

**matrix = []**

**while True:**

**try:**

**print("\n",title,"State = ")**

**state = list(input("Enter the state matrix seperated by ',':").strip().split(','))[:9]**

**if len(state) != 9:**

**print("Please provide correct input length for the puzzle")**

**else:**

**for elem in range(0,9):**

**if(state[elem]!=''):**

**state[elem]=int(state[elem])**

**else:**

**state[elem]=0**

**break**

**except ValueError:**

**print("Please provide the CORRECT INPUT.\nEnter the state matrix seperated by ',':")**

**input\_matrix(title)**

**return [state[0:3],state[3:6],state[6:9]]**

**# Read the states {start,goal} and initialize node count to 0**

**def read\_states(self):**

**self.start\_state = self.input\_matrix("Start")**

**self.goal\_state = self.input\_matrix("Goal")**

**self.node\_count = 0**

**# get the heuristic {h1, h2} from the user**

**def heuristic(self):**

**while True:**

**self.heuristic\_type = input("\nPlease select a heuristic as h1 or h2:")**

**if self.heuristic\_type == "h1" or self.heuristic\_type == "h2":**

**break**

**else:**

**print("Please provide Valid Input!!!")**

**# h1 calculates the distance between current state and goal state**

**def h1(self,current\_state,goal\_state):**

**h1\_value = 0**

**for num in range(0,9):**

**current\_position = self.get\_position(current\_state,num)**

**goal\_position = self.get\_position(goal\_state,num)**

**if current\_position != goal\_position:**

**h1\_value += 1**

**return h1\_value**

**# h2 calculates manhatten distance**

**def h2(self,current\_state,goal\_state):**

**h2\_value = 0**

**for num in range(0,9):**

**current\_position = self.get\_position(current\_state,num)**

**goal\_position = self.get\_position(goal\_state,num)**

**m\_distance = abs(current\_position[0] - goal\_position[0]) + abs(current\_position[1] - goal\_position[1])**

**h2\_value += m\_distance**

**return h2\_value**

**# selected node with f,g and h values**

**def selected\_node(self,node,info = True):**

**curstate = node**

**if info == True:**

**print("cost g(x) = ", node.level, "heuristic h(x) = ",node.hval, "total cost f(x) = g(x)+h(x)",node.fval)**

**curstate = node.state**

**for i in curstate:**

**print(i)**

**print("\n\n")**

**def path(self,node):**

**self.selected\_node(node,False)**

**if node.parent == None:**

**return**

**self.path(node.parent)**

**# f calculates hueristic value => f(x) = h(x) + g(x)**

**def f(self,current\_state,goal\_state,h\_type):**

**if h\_type == 'h1':**

**return self.h1(current\_state,goal\_state)+current\_state.level**

**else:**

**return self.h2(current\_state,goal\_state)+current\_state.level**

**def Execution\_process(self):**

**self.read\_states()**

**self.heuristic()**

**if self.heuristic == "h1":**

**initial\_hval = self.h1(self.start\_state,self.goal\_state)**

**else:**

**initial\_hval = self.h2(self.start\_state,self.goal\_state)**

**print("Provide your start state matrix \n")**

**self.selected\_node(self.start\_state,False)**

**print("Provide the final/goal state matrix \n")**

**self.selected\_node(self.goal\_state,False)**

**print(self.start\_state)**

**# Initialize start state and append the start node to opened list**

**start\_state = Node(self.start\_state,0,initial\_hval,None)**

**self.opened.append(start\_state)**

**while True:**

**cur\_node = self.opened[0]**

**self.node\_count += 1**

**self.selected\_node(cur\_node)**

**if cur\_node.hval == 0:**

**print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**

**break**

**cur\_node.generate\_child\_nodes()**

**for node in cur\_node.child\_nodes:**

**temp\_node = cur\_node.shuffle(node,cur\_node.zero)**

**if self.heuristic == 'h1':**

**temp\_hval = self.h1(temp\_node,self.goal\_state)**

**else:**

**temp\_hval = self.h2(temp\_node,self.goal\_state)**

**cur\_node.children.append(Node(temp\_node,cur\_node.level+1,temp\_hval,cur\_node))**

**for node in cur\_node.children:**

**self.opened.append(node)**

**self.closed.append(cur\_node)**

**del self.opened[0]**

**self.opened.sort(key = lambda val:val.fval,reverse=False)**

**# if the program is unable to find solution after 150 iterations then we end it saying no solution is found**

**if self.node\_count > 150:**

**print("Unable to find solution after 150 iterations!!")**

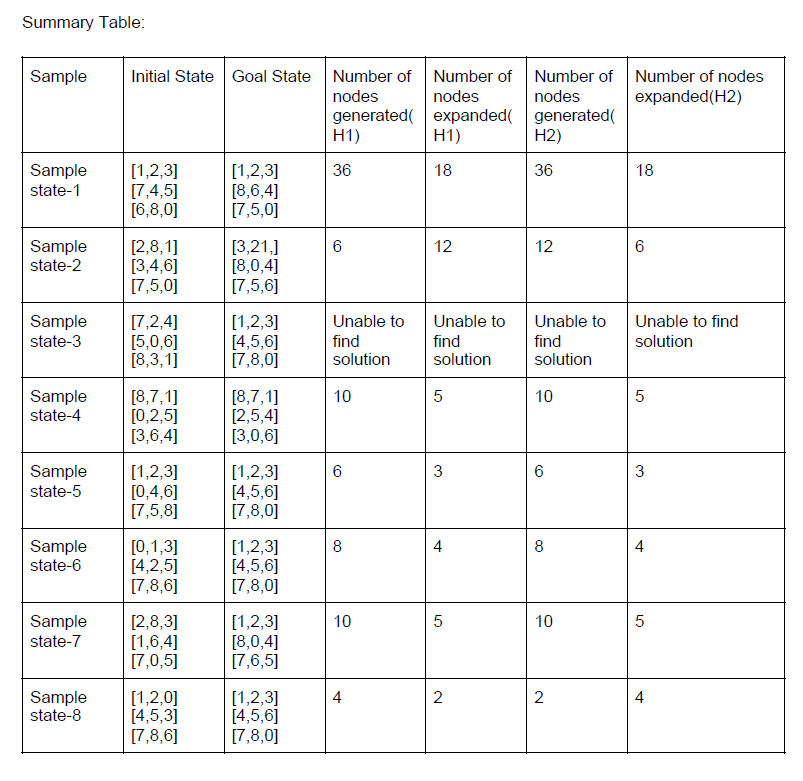
**break**

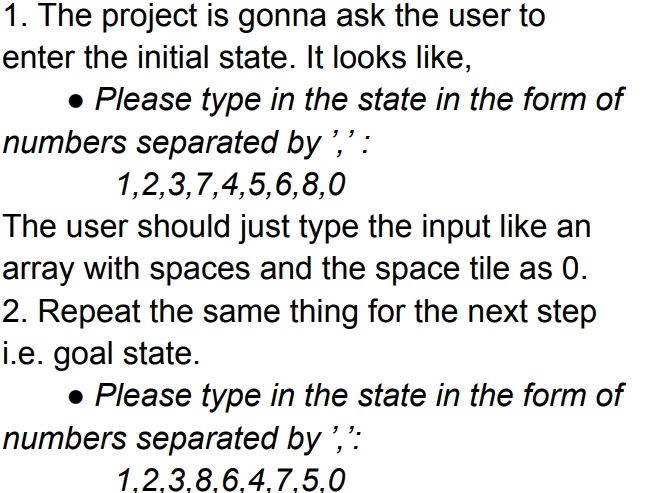
**puzzle = Astar\_8\_puzzle()**

**puzzle.Execution\_process()**

**print("Number of nodes expanded: ", len(puzzle.closed))**

**print("Number of nodes generated: ", len(puzzle.closed) + len(puzzle.closed))**

****

****

**OUTPUT**

Start State =

Enter the state matrix seperated by ',':1,2,3,4,5,6,8,0

Please provide correct input length for the puzzle

Start State =

Enter the state matrix seperated by ',':1,2,3,8,6,4,7,5,0

Goal State =

Enter the state matrix seperated by ',':1,2,3,4,5,6,7,8,0

Please select a heuristic as h1 or h2:h1

Provide your start state matrix

[1, 2, 3]

[8, 6, 4]

[7, 5, 0]

Provide the final/goal state matrix

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

[[1, 2, 3], [8, 6, 4], [7, 5, 0]]

cost g(x) = 0 heuristic h(x) = 6 total cost f(x) = g(x)+h(x) 6

[1, 2, 3]

[8, 6, 4]

[7, 5, 0]

Unable to find solution after 150 iterations!!

Number of nodes expanded: 151

Number of nodes generated: 302

1. **Implement an AI program for alpha beta pruning**

**class BinaryTree():**

**def \_\_init\_\_(self, rootid, value):**

**self.left = None**

**self.right = None**

**self.rootid = rootid**

**self.value = value**

**def getLeftChild(self):**

**return self.left**

**def getRightChild(self):**

**return self.right**

**def setNodeValue(self, value):**

**self.value = value**

**def getNodeValue(self):**

**return self.value**

**def getNodePlace(self):**

**return self.rootid**

**def insertRight(self, childid, newNode):**

**self.right = BinaryTree(childid, newNode)**

**def insertLeft(self, childid, newNode):**

**self.left = BinaryTree(childid, newNode)**

**def testTree():**

**myTree = BinaryTree("l\_00", inf)**

**myTree.insertLeft("l\_10", ninf)**

**myTree.insertRight("l\_11", ninf)**

**# layer 1**

**l10 = myTree.getLeftChild()**

**l10.insertLeft("l\_20", inf)**

**l10.insertRight("l\_21", inf)**

**l11 = myTree.getRightChild()**

**l11.insertLeft("l\_22", inf)**

**l11.insertRight("l\_23", inf)**

**# layer2**

**l20 = l10.getLeftChild()**

**l20.insertLeft("l\_30", ninf)**

**l20.insertRight("l\_31", ninf)**

**l21 = l10.getRightChild()**

**l21.insertLeft("l\_32", ninf)**

**l21.insertRight("l\_33", ninf)**

**l22 = l11.getLeftChild()**

**l22.insertLeft("l\_34", ninf)**

**l22.insertRight("l\_35", ninf)**

**l23 = l11.getRightChild()**

**l23.insertLeft("l\_36", ninf)**

**l23.insertRight("l\_37", ninf)**

**# layer 3**

**l30 = l20.getLeftChild()**

**l30.insertLeft("l\_40", 3)**

**l30.insertRight("l\_41", 10)**

**l31 = l20.getRightChild()**

**l31.insertLeft("l\_42", 2)**

**l31.insertRight("l\_43", 9)**

**l32 = l21.getLeftChild()**

**l32.insertLeft("l\_44", 10)**

**l32.insertRight("l\_45", 7)**

**l33 = l21.getRightChild()**

**l33.insertLeft("l\_46", 5)**

**l33.insertRight("l\_47", 9)**

**l34 = l22.getLeftChild()**

**l34.insertLeft("l\_48", 2)**

**l34.insertRight("l\_49", 5)**

**l35 = l22.getRightChild()**

**l35.insertLeft("l\_410", 6)**

**l35.insertRight("l\_411", 4)**

**l36 = l23.getLeftChild()**

**l36.insertLeft("l\_412", 2)**

**l36.insertRight("l\_413", 7)**

**l37 = l23.getRightChild()**

**l37.insertLeft("l\_414", 9)**

**l37.insertRight("l\_415", 1)**

**return myTree**

**inf = 99999999**

**ninf = -99999999**

**def maximizer(current,alpha,beta):**

**if current.getNodeValue() != inf and current.getNodeValue() != ninf:**

**return current.getNodeValue()**

**children = [current.getLeftChild(), current.getRightChild()]**

**comp = ninf**

**for i in children:**

**eval = minimizer(i,alpha,beta)**

**comp = max(comp, eval)**

**alpha=max(alpha,eval)**

**if beta <= alpha:**

**break**

**return comp**

**def minimizer(current,alpha,beta):**

**if current.getNodeValue() != inf and current.getNodeValue() != ninf:**

**return current.getNodeValue()**

**children = [current.getLeftChild(), current.getRightChild()]**

**comp = inf**

**for i in children:**

**eval = maximizer(i,alpha,beta)**

**comp = min(comp, eval)**

**beta=min(beta,eval)**

**if beta<=alpha:**

**break**

**return comp**

**def path(root, id, winner):**

**if (not root):**

**return False**

**id.append(root.rootid)**

**if root.getNodeValue() == winner:**

**return True**

**if (path(root.getLeftChild(), id, winner) or path(root.getRightChild(), id, winner)):**

**return True**

**id.pop(-1)**

**return False**

**def minimax(root, winner):**

**id = []**

**if path(root, id, winner)==True:**

**print("The winner node is:",winner)**

**print("The path is:")**

**for i in range(len(id) - 1):**

**print( id[i])**

**print(id[len(id) - 1])**

**else:**

**print("No Path")**

**#We call maximizer because our root node is maximizer for this example**

**minimax(testTree(), maximizer(testTree(),ninf,inf))**

**OUTPUT**

The winner node is: 3

The path is:

l\_00

l\_10

l\_20

l\_30

l\_40

1. **Implement an AI program forTic-Tac-Toe game in python**

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

for row in board:

if all(cell == player for cell in row):

return True

for col in range(3):

if all(row[col] == player for row in board):

return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_board\_full(board):

return all(all(cell != '-' for cell in row) for row in board)

def tic\_tac\_toe():

board = [['-' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

while True:

print\_board(board)

print(f"Player {current\_player}'s turn")

while True:

row = int(input("Enter row (0, 1, or 2): "))

col = int(input("Enter column (0, 1, or 2): "))

if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == '-':

break

else:

print("Invalid move. Try again.")

board[row][col] = current\_player

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

elif is\_board\_full(board):

print\_board(board)

print("It's a draw!")

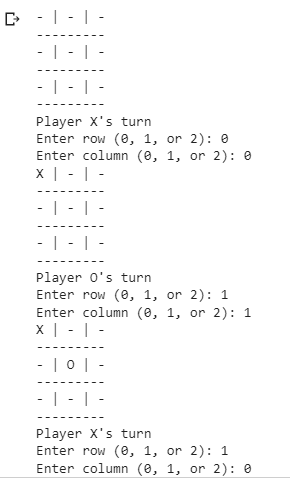
break

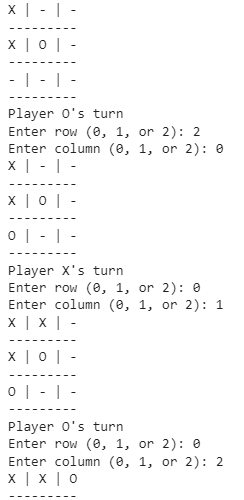
current\_player = 'O' if current\_player == 'X' else 'X'

if \_\_name\_\_ == "\_\_main\_\_":

tic\_tac\_toe()

**OUTPUT**







1. **Implement an AI program for Water Jug Problem.**

**PROGRAM:**

**import sys**

**visited\_states = set()**

**def state(x, y):**

**if (x, y) in visited\_states:**

**return**

**visited\_states.add((x, y))**

**if x < 4:**

**print(f"Fill the 4-Gallon Jug: ({x},{y}) --> (4,{y})")**

**sys.stdout.flush()**

**state(4, y)**

**if y < 3:**

**print(f"Fill the 3-Gallon Jug: ({x},{y}) --> ({x},3)")**

**sys.stdout.flush()**

**state(x, 3)**

**if x > 0:**

**print(f"Empty the 4-Gallon jug on ground: ({x},{y}) --> (0,{y})")**

**sys.stdout.flush()**

**state(0, y)**

**if y > 0:**

**print(f"Empty the 3-Gallon jug on ground: ({x},{y}) --> ({x},0)")**

**sys.stdout.flush()**

**state(x, 0)**

**pour\_amount = min(x, 3 - y)**

**if pour\_amount > 0:**

**new\_x, new\_y = x - pour\_amount, y + pour\_amount**

**print(f"Pour {pour\_amount} gallons from 4-Gallon jug to 3-gallon: ({x},{y}) --> ({new\_x},{new\_y})")**

**sys.stdout.flush()**

**state(new\_x, new\_y)**

**pour\_amount = min(y, 4 - x)**

**if pour\_amount > 0:**

**new\_x, new\_y = x + pour\_amount, y - pour\_amount**

**print(f"Pour {pour\_amount} gallons from 3-Gallon jug to 4-gallon: ({x},{y}) --> ({new\_x},{new\_y})")**

**sys.stdout.flush()**

**state(new\_x, new\_y)**

**def goal():**

**state(0, 0)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**goal()**

**OUTPUT:**

Fill the 4-Gallon Jug: (0,0) --> (4,0)

Fill the 3-Gallon Jug: (4,0) --> (4,3)

Empty the 4-Gallon jug on ground: (4,3) --> (0,3)

Fill the 4-Gallon Jug: (0,3) --> (4,3)

Empty the 3-Gallon jug on ground: (0,3) --> (0,0)

Pour 3 gallons from 3-Gallon jug to 4-gallon: (0,3) --> (3,0)

Fill the 4-Gallon Jug: (3,0) --> (4,0)

Fill the 3-Gallon Jug: (3,0) --> (3,3)

Fill the 4-Gallon Jug: (3,3) --> (4,3)

Empty the 4-Gallon jug on ground: (3,3) --> (0,3)

Empty the 3-Gallon jug on ground: (3,3) --> (3,0)

Pour 1 gallons from 3-Gallon jug to 4-gallon: (3,3) --> (4,2)

Fill the 3-Gallon Jug: (4,2) --> (4,3)

Empty the 4-Gallon jug on ground: (4,2) --> (0,2)

Fill the 4-Gallon Jug: (0,2) --> (4,2)

Fill the 3-Gallon Jug: (0,2) --> (0,3)

Empty the 3-Gallon jug on ground: (0,2) --> (0,0)

Pour 2 gallons from 3-Gallon jug to 4-gallon: (0,2) --> (2,0)

Fill the 4-Gallon Jug: (2,0) --> (4,0)

Fill the 3-Gallon Jug: (2,0) --> (2,3)

Fill the 4-Gallon Jug: (2,3) --> (4,3)

Empty the 4-Gallon jug on ground: (2,3) --> (0,3)

Empty the 3-Gallon jug on ground: (2,3) --> (2,0)

Pour 2 gallons from 3-Gallon jug to 4-gallon: (2,3) --> (4,1)

Fill the 3-Gallon Jug: (4,1) --> (4,3)

Empty the 4-Gallon jug on ground: (4,1) --> (0,1)

Fill the 4-Gallon Jug: (0,1) --> (4,1)

Fill the 3-Gallon Jug: (0,1) --> (0,3)

Empty the 3-Gallon jug on ground: (0,1) --> (0,0)

Pour 1 gallons from 3-Gallon jug to 4-gallon: (0,1) --> (1,0)

Fill the 4-Gallon Jug: (1,0) --> (4,0)

Fill the 3-Gallon Jug: (1,0) --> (1,3)

Fill the 4-Gallon Jug: (1,3) --> (4,3)

Empty the 4-Gallon jug on ground: (1,3) --> (0,3)

Empty the 3-Gallon jug on ground: (1,3) --> (1,0)

Pour 3 gallons from 3-Gallon jug to 4-gallon: (1,3) --> (4,0)

Empty the 4-Gallon jug on ground: (1,0) --> (0,0)

Pour 1 gallons from 4-Gallon jug to 3-gallon: (1,0) --> (0,1)

Empty the 3-Gallon jug on ground: (4,1) --> (4,0)

Pour 2 gallons from 4-Gallon jug to 3-gallon: (4,1) --> (2,3)

Empty the 4-Gallon jug on ground: (2,0) --> (0,0)

Pour 2 gallons from 4-Gallon jug to 3-gallon: (2,0) --> (0,2)

Empty the 3-Gallon jug on ground: (4,2) --> (4,0)

Pour 1 gallons from 4-Gallon jug to 3-gallon: (4,2) --> (3,3)

Empty the 4-Gallon jug on ground: (3,0) --> (0,0)

Pour 3 gallons from 4-Gallon jug to 3-gallon: (3,0) --> (0,3)

Empty the 3-Gallon jug on ground: (4,3) --> (4,0)

Empty the 4-Gallon jug on ground: (4,0) --> (0,0)

Pour 3 gallons from 4-Gallon jug to 3-gallon: (4,0) --> (1,3)

Fill the 3-Gallon Jug: (0,0) --> (0,3)

1. **Write a program to implement Monkey Banana Problem.**

**PROGRAM:**

# Action class to represent actions

class Action:

    def \_\_init\_\_(self, name, \*params):

        self.name = name

        self.params = params

# State class to represent states

class State:

    def \_\_init\_\_(self, monkey\_loc, monkey\_on, box\_loc, has\_banana):

        self.monkey\_loc = monkey\_loc

        self.monkey\_on = monkey\_on

        self.box\_loc = box\_loc

        self.has\_banana = has\_banana

# Define actions

grab = Action("grab")

climb = Action("climb")

push = Action("push")

walk = Action("walk")

# Define legal actions

def do(state):

    if state.monkey\_on == "onbox" and state.monkey\_loc == state.box\_loc and state.has\_banana == "hasnot":

        return [Action("grab")]

    elif state.monkey\_on == "onfloor" and state.monkey\_loc == state.box\_loc:

        return [Action("climb")]

    elif state.monkey\_on == "onfloor":

        return [Action("push", state.monkey\_loc, loc) for loc in ["middle", "atwindow"]]

    elif state.monkey\_on == "onbox":

        return [Action("walk", state.monkey\_loc, loc) for loc in ["atdoor", "atwindow"]]

# Define goal state

def is\_goal\_state(state):

    return state.has\_banana == "has"

# Generate plans

def generate\_plan(state, plan=[]):

    if is\_goal\_state(state):

        return plan

    possible\_actions = do(state)

    for action in possible\_actions:

        new\_state = apply\_action(state, action)

        new\_plan = plan + [action]

        result = generate\_plan(new\_state, new\_plan)

        if result:

            return result

    return []

# Apply action to a state

def apply\_action(state, action):

    if action.name == "grab":

        return State(state.monkey\_loc, state.monkey\_on, state.box\_loc, "has")

    elif action.name == "climb":

        return State(state.monkey\_loc, "onbox", state.box\_loc, state.has\_banana)

    elif action.name == "push":

        return State(action.params[1], state.monkey\_on, action.params[1], state.has\_banana)

    elif action.name == "walk":

        return State(action.params[1], state.monkey\_on, state.box\_loc, state.has\_banana)

# Start solving the problem

initial\_state = State("atdoor", "onfloor", "atwindow", "hasnot")

plan = generate\_plan(initial\_state)

if plan:

    print("Plan to get the banana:")

    for action in plan:

        print(action.name)

else:

    print("No plan found to get the banana.")

**OUTPUT:**

Plan to get the banana:

push

climb

grab

1. **Write a program in python to implement medical diagnosis prediction .**

from sklearn.tree import DecisionTreeClassifier

from sklearn.preprocessing import MultiLabelBinarizer, LabelEncoder

# Sample medical data

symptoms\_data = [

    (['fever', 'cough', 'fatigue', 'sore\_throat', 'headache'], 'diagnosis1'),

    (['fever', 'body\_ache', 'chills'], 'diagnosis2'),

    (['cough', 'runny\_nose', 'conjunctivitis'], 'diagnosis3'),

]

    # ... more data ...

# Separate symptoms and diagnoses

symptoms = [row[0] for row in symptoms\_data]

diagnoses = [row[1] for row in symptoms\_data]

# Create a label encoder for diagnoses

diagnosis\_label\_encoder = LabelEncoder()

encoded\_diagnoses = diagnosis\_label\_encoder.fit\_transform(diagnoses)

# Create a multi-label binarizer for symptoms

symptom\_multilabel\_binarizer = MultiLabelBinarizer()

encoded\_symptoms = symptom\_multilabel\_binarizer.fit\_transform(symptoms)

# Create a decision tree classifier

model = DecisionTreeClassifier()

# Train the model

model.fit(encoded\_symptoms, encoded\_diagnoses)

# Get symptom names from the multi-label binarizer

symptom\_names = symptom\_multilabel\_binarizer.classes\_

# Predict a new case using console input

new\_symptoms = input("Enter symptoms (comma-separated): ").split(",")

encoded\_new\_symptoms = symptom\_multilabel\_binarizer.transform([new\_symptoms])

predicted\_diagnosis = model.predict(encoded\_new\_symptoms)

predicted\_diagnosis\_label = diagnosis\_label\_encoder.inverse\_transform(predicted\_diagnosis)

print(f"Predicted Diagnosis: {predicted\_diagnosis\_label[0]}")

**OUTPUT**

Enter symptoms (comma-separated): fever,body\_ache,chills Predicted Diagnosis: diagnosis2

1. Implenet Block program

**PROGRAM**

class Block:

def \_\_init\_\_(self, name):

self.name = name

self.on\_top = None

def put\_on(self, other\_block):

if self.on\_top:

self.on\_top.on\_top = None

self.on\_top = other\_block

other\_block.on\_top = self

def clear(self):

return self.on\_top is None

def \_\_str\_\_(self):

return self.name

def achieve\_on(goal\_block, base\_block):

if not goal\_block.on\_top:

base\_block.put\_on(goal\_block)

def valid(state, goals):

return all(block.clear() for block in state) and all(goal\_block.on\_top == base\_block for goal\_block, base\_block in goals)

def achieve\_goal(state, goals):

for goal\_block, base\_block in goals:

achieve\_on(goal\_block, base\_block)

# Define the blocks

a = Block("a")

b = Block("b")

c = Block("c")

# Define initial state

initial\_state = [a, b, c]

# Define goal configuration

goal\_config = [(a, c), (b, a), (c, b)]

# Check if initial state satisfies the goals

if valid(initial\_state, goal\_config):

print("Initial state satisfies the goals.")

else:

print("Initial state does not satisfy the goals. Trying to achieve them...")

achieve\_goal(initial\_state, goal\_config)

# Print final state

print("Final state:")

for block in initial\_state:

print(f"{block}: On top of {block.on\_top}" if block.on\_top else f"{block}: On table")

**OUTPUT:**

Initial state does not satisfy the goals. Trying to achieve them... Final state: a: On table b: On top of c c: On top of b