#1 A* SEARCH ALGO

In [ ]:

```
def aStarAlgo(start_node, stop_node):
        open_set = set(start_node)
        closed_set = set()
        g = {}
        parents = {}
        g[start_node] = 0
        parents[start_node] = start_node
        while len(open_set) > 0:
            n = None
            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):

                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight

                    else:
                        if g[m] > g[n] + weight:
                            #update g(m)
                            g[m] = g[n] + weight
                            #change parent of m to n
                            parents[m] = n
                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)
            if n == None:
                print('Path does not exist!')
                return None
            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
            open_set.remove(n)
            closed_set.add(n)

        print('Path does not exist!')
        return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }
        return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

#2 AO* SEARCH ALGO

In [ ]:

```
def aoStar(graph, H, startNode):     # AO* algorithm with input data graph structure, heuristics and start node
```

```
    openList  = list()  # Openlist for nodes to be explored
    closeList = list()  # Closlist for nodes already processed
    G = dict()          # F(X) = G(X) + H(X)
    S = dict()          # SOLVED status
    P = dict()          # PARENT of a node
    U = dict()          # UPDATED status for heuristic value

    openList.append([startNode])   # Initialization of openlist with startnode
    H[startNode] = 0               # Initialization for start node
    G[startNode] = 0
    S[startNode] = False
    P[startNode] = startNode
    U[startNode] = False


    while S[startNode]==False:     # As long as startnode is not solved, loop!!
        print("---------------------------------------------------------------------------------------")
        print(openList)
        print(closeList)           # Printing status on each iteration
        print(H)
        print(S)
        print("----------------------------------------------------------------------------------------")

        bestNodeList=None                                # Compute node with lowest f(x) on AO Graph
        bestNodeCost=0
        for nodeList in openList:                        # Each element is a list
            currentNodeListHCost=0
            currentNodeListGCost=0
            for node in nodeList:                        # Compute G(X) and H(X) for each node in the list
                currentNodeListHCost = currentNodeListHCost + H[node]
                currentNodeListGCost = currentNodeListGCost + 1     # Weight between nodes is valued one, but can
            currentNodeListGCost=currentNodeListGCost + G[P[nodeList[0]]]
            if bestNodeList==None or bestNodeCost>(currentNodeListGCost+currentNodeListHCost):
                bestNodeList=nodeList
                bestNodeCost=(currentNodeListGCost+currentNodeListHCost)
            print((currentNodeListGCost+currentNodeListHCost),":",nodeList)
        openList.remove(bestNodeList)                    # Move the best node(list) to close list of
        closeList.append(bestNodeList)


        for node in bestNodeList:                        # Process each node in the best node(list) for expansion
            if graph.get(node,None) == None:             # Expand each node with its child nodes
                S[node]=True                             # If node itself is child node, set the node as solved us
            elif S[node]==True:
                continue
            else:
                for childNodeList in graph[node]:        # If child nodes lists are availabile, place each of them
                    if childNodeList not in openList and childNodeList not in closeList:
                        openList.append(childNodeList)

                        for child in childNodeList:      # Initialize data of newly added node in the child node l
                            U[child] = False
                            S[child] = False
                            P[child] = node
                            G[child] = G[node] + 1       # Weight is set to 1, but can vary by weight matrix
                    else:
                        for child in childNodeList:      # Update all the data of node already in closelist
                            if G[child] > G[node] + 1:
                                U[child] = False
                                S[child] = False
                                G[child] = G[node] + 1
                                P[child] = node

                    if childNodeList in closeList:       # If a node is updated, move back to open list
                        closeList.remove(childNodeList)
                        openList.append(childNodeList)

        solved=True                                      # Checking all the node in the best node list are solved
        HeuristicCost=0
        for node in bestNodeList:
            solved=solved & S[node]
            HeuristicCost = HeuristicCost + H[node]

        if solved == True:                               # If all nodes in the best node list are solved, update i
            for node in bestNodeList:
                if U[P[node]] == False:                  # If parent's heuristic is not updated, updated it now an
                    U[P[node]] = True
                    S[P[node]] = True
                    H[P[node]] = HeuristicCost + len(bestNodeList)
                    break

                elif H[P[node]] > (HeuristicCost + len(bestNodeList)):  # If parent's heuristic is updated earlie
                    H[P[node]]  = (HeuristicCost + len(bestNodeList))   # update it with latest best value
                    S[P[node]]  = True
                    break

            for andedOrNodes in AOList:                                 # Check all parent nodes in a anded list
                for node in bestNodeList:
                    if P[node] in andedOrNodes:
```

```
                        status  = True
                        for aoNode in andedOrNodes:                      # If all parent nodes in a anded list are
                            status = status & S[aoNode]
                        if status==True:                                 # move back the parent node list back to
                            if andedOrNodes not in openList:             # for backtracking the revised heuristic
                                openList.append(andedOrNodes)
    print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    print("Final Heuristic values of nodes:",H)
    print("Final Solved status of nodes:",S)
    print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    # Input data : heuristics, graph structure, anded/or list of nodes, start node

h1 = {'A': 0, 'B': 4, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G': 2, 'H': 0, 'I': 0, 'J': 0}  # First input heuristic da
graph1 = {                                                                              # Graph structure
    'A': [['C', 'D'], ['B']],
    'B': [['E'], ['F']],
    'C': [['G'],['H', 'I']],
    'D': [['J']]
}
AOList=[['A'],['B'],['C','D'],['E'],['F'],['G'],['H','I'],['J']]                    # Anded/Or condition list of n
aoStar(graph1, h1, 'A')                                                             # Invoke AO* algorithm with st

print("&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&")

h2 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
graph2 = {
    'A': [['B','C'], ['D']],
    'B': [['G'], ['H']],
    'C': [['J']],
    'D': [['E','F']],
    'G': [['I']]
}
AOList=[['A'],['B','C'],['D'],['G'],['H'],['J'],['E','F'],['I']]
aoStar(graph2, h2, 'A')
print("&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&")
h3 = {'A': 1, 'B': 6, 'C': 12, 'D': 11, 'E': 4, 'F': 4, 'G': 5, 'H': 7}  # Heuristic values of Nodes

graph3 = {                                   # Graph of Nodes and Edges
    'A': [['B','C'], ['D']],                 # Neighbors of Node 'A', B, C & D with repective weights
    'B': [['G'], ['H']],                     # Neighbors are included in a list of lists
    'D': [['E','F']]                         # Each sublist indicate a "OR" node or "AND" nodes
}
AOList=[['A'],['B','C'],['D'],['G'],['H'],['E','F']]
aoStar(graph3, h3, 'A')
```

#3 CANDIDATE ELIMINATION

In [ ]:

```
import numpy as np
import pandas as pd

data = pd.read_csv('EconomyCar.csv',header=None)
concepts = np.array(data.iloc[:,0:-1])
print("\nInstances are:\n",concepts)
target = np.array(data.iloc[:,-1])
print("\nTarget Values are: ",target)

def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and genearal_h")
    print("\nSpecific Boundary: ", specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print("\nGeneric Boundary: ",general_h)

    for i, h in enumerate(concepts):
        print("\nInstance", i+1 , "is ", h)
        if target[i] == "Yes":
            print("Instance is Positive ")
            for x in range(len(specific_h)):
                if h[x]!= specific_h[x]:
                    specific_h[x] ='?'
                    general_h[x][x] ='?'

        if target[i] == "No":
            print("Instance is Negative ")
            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] = '?'

        print("Specific Bundary after ", i+1, "Instance is ", specific_h)
        print("Generic Boundary after ", i+1, "Instance is ", general_h)
        print("\n")
```

```
    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 Specific_h: ", s_final, sep="\n")
print("Final General_h: ", g_final, sep="\n")
```

#4 K-MEANS/EM PY

In [ ]:

```
from sklearn import datasets
from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X_train,X_test,y_train,y_test = train_test_split(iris.data,iris.target)
model =KMeans(n_clusters=3)
model.fit(X_train,y_train)
metrics.accuracy_score(y_test,model.predict(X_test))

#-------Expectation and Maximization----------
from sklearn.mixture import GaussianMixture
model2 = GaussianMixture(n_components=3)
model2.fit(X_train,y_train)
model2.score
metrics.accuracy_score(y_test,model2.predict(X_test))
```

#5 KNN

In [ ]:

```
import pandas as pd

df_irisbd = pd.read_csv('iris.csv',header=None,index_col=None)
print(df_irisbd)
X = df_irisbd.iloc[:, :-1].values
y = df_irisbd.iloc[:, 4].values
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,train_size=0.8,random_state=100)
print(y_test)
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training sets
model.fit(X_train,y_train)

predicted= model.predict(X_test) # 0:Overcast, 2:Mild
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, predicted))
print(classification_report(y_test, predicted))
```

#6 Naive Bayesian Classifiers

In [ ]:

```
import pandas as pd
import random as rd
from sklearn.metrics import confusion_matrix
import math

lut=dict()
label_attribute="PlayTennis"
labels=["No","Yes"]

df=pd.read_csv("tennis.csv")
print(df)
total=len(df)

attribute_names=df.columns.tolist()
attribute_names.remove(label_attribute)
ldf=df.pivot_table(index=[label_attribute], columns=[label_attribute], aggfunc='size')
print(ldf)
lut[label_attribute]=ldf

for attribute in attribute_names:
    lut[attribute]=df.pivot_table(index=["PlayTennis"], columns=[attribute], aggfunc='size')
    lut[attribute].fillna(0,inplace=True)
```

```
target=list()
prediction=list()

for index, row in df.iterrows():
    if rd.random()>0.5:
        result=list()
        for label in labels:
            posteriorProb=math.log(lut[label_attribute][label][label]/total)
            for attribute in attribute_names:
                value=row[attribute]
                posteriorProb=posteriorProb+math.log(lut[attribute][value][label]/lut[label_attribute][label][lab
            result.append(posteriorProb)

        target.append(row[label_attribute])
        print(result)
        maxindex=result.index(max(result))
        prediction.append(labels[maxindex])

print(confusion_matrix(target,prediction))
print(target)
print(prediction)
```

#7 ID3

In [ ]:

```
import pandas as pd
from pprint import pprint
from sklearn.feature_selection import mutual_info_classif

def id3(df, target_attribute, attribute_names, default_class=None):

    from collections import Counter
    cnt=Counter(x for x in df[target_attribute])
    if len(cnt)==1:
        return next(iter(cnt))

    elif df.empty or (not attribute_names):
         return default_class

    else:
        gainz = mutual_info_classif(df[attribute_names],df[target_attribute],discrete_features=True)
        index_of_max=gainz.tolist().index(max(gainz))
        best_attr=attribute_names[index_of_max]
        tree={best_attr:{}}
        remaining_attribute_names=[i for i in attribute_names if i!=best_attr]
        for attr_val, data_subset in df.groupby(best_attr):
            subtree=id3(data_subset, target_attribute, remaining_attribute_names,default_class)

            tree[best_attr][attr_val]=subtree

        return tree


df=pd.read_csv("tennis.csv")

attribute_names=df.columns.tolist()
print("List of attribut name")

attribute_names.remove("PlayTennis")

for colname in df.select_dtypes("object"):
    df[colname], _ = df[colname].factorize()

print(df)

tree= id3(df,"PlayTennis", attribute_names)
print("The tree structure")
pprint(tree)
```

#8 BACKPROPAGATION ALGORITHM

In [ ]:

```
import numpy as np
inputNeurons=2
hiddenlayerNeurons=2
outputNeurons=2

input  = np.random.randint(1,10,size=(1,inputNeurons))
output = np.array([1.0, 0.0])

hidden_weights=np.random.normal(scale=0.5, size=(inputNeurons,hiddenlayerNeurons))
output_weights=np.random.normal(scale=0.5, size=(hiddenlayerNeurons,outputNeurons))
```

```
def sigmoid (layer):
    return 1/(1 + np.exp(-layer))

def gradient(layer):
    return layer*(1-layer)

for i in range(2000):
    #Feedforward inputs
    L1in=np.dot(input,hidden_weights)
    L1out=sigmoid(L1in)

    L2in=np.dot(L1out,output_weights)
    L2out=sigmoid(L2in)

    #Backpropogate Error
    error = (L2out-output)
    gradient_L2=gradient(L2out)
    error_terms_L2=error*gradient_L2

    error=np.dot(error_terms_L2, output_weights.T)
    gradient_L1=gradient(L1out)
    error_terms_L1=error * gradient_L1

    gradient_output_weights = np.dot(L1out.T,error_terms_L2)
    gradient_hidden_weights = np.dot(input.T,error_terms_L1)

    output_weights = output_weights - 0.1*gradient_output_weights
    hidden_weights = hidden_weights - 0.1*gradient_hidden_weights

    print("**********************")
    print("iteration:",i,"<error>:",error)
    print("              <output>:",L2out)
```

#9 LOCALLY WEIGHTED REGRESSION ALGORITHM (LOWESS)

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)
y = np.log(np.abs((x ** 2) - 1) + 0.5)
x = x + np.random.normal(scale=0.05, size=1000)
plt.scatter(x, y, alpha=0.3)


def local_regression(x0, x, y, tau):
    x0 = np.r_[1, x0]
    x = np.c_[np.ones(len(x)), x]
    xw = x.T * radial_kernel(x0, x, tau)
    beta = np.linalg.pinv(xw @ x) @ xw @ y
    return x0 @ beta


def radial_kernel(x0, x, tau):
    return np.exp(np.sum((x - x0) ** 2, axis=1) / (-2 * tau ** 2))


def plot_lr(tau):
    domain = np.linspace(-5, 5, num=300)
    pred = [local_regression(x0, x, y, tau) for x0 in domain]
    plt.scatter(x, y, alpha=0.3)
    plt.plot(domain, pred, color="red")
    return plt


plot_lr(0.03).show()
```

In [ ]: