Program 01 AStar

```python
graph = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'D': [('G', 1)],
    'E': [('D', 6)],
    'G': None
}
heuristicValues = {
    'A': 11,
    'B': 6,
    'C': 99,
    'D': 1,
    'E': 7,
    'G': 0,
}
parents = {}
openList = []
closedList = []
g = {}


def h(n):
    return heuristicValues.get(n, None)


def getNeighbors(n):
    return graph.get(n, None)


def aStar(startNode, stopNode):
    parents[startNode] = startNode
    g[startNode] = 0
    openList.append(startNode)
    while len(openList) > 0:  # missed while loop
        n = None
        for v in openList:
            if n is None or g[n] + h(n) > g[v] + h(v):
                n = v

        if n is None:
            return n

        if n == stopNode:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(n)
            path.reverse()
            return path
```

```
        for node, weight in getNeighbors(n):
            parents[node] = n

            if node not in [openList, closedList]:
                openList.append(node)
                g[node] = g[n] + weight

            else:
                if g[node] > g[n] + weight:
                    g[node] = g[n] + weight
                    if node in closedList:
                        closedList.remove(node)
                        openList.append(node)

        openList.remove(n)
        closedList.append(n)


res = aStar('A', 'G')

if res is None:
    print('No solution exists')
else:
    print('Path found: \n {}'.format(res))
```

program 02 AOStar

```
graph = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]],
}
h = {
    'A': 1,
    'B': 6,
    'C': 12,
    'D': 10,
    'E': 4,
    'F': 4,
    'G': 5,
    'H': 7,
    'I': 1,
    'J': 1
}
status = {}
parents = {}

def getS(n):
    return status.get(n, 0)

def getH(n):
    return h.get(n, None)

def getNeighbors(n):
    return graph.get(n, [])

def getMinimumCostChild(n):
    minimumCost = 999
    minimumCostChild = []

    for listOfTuples in getNeighbors(n):
        cost = 0
        childNodes = []
        for node, weight in listOfTuples:
            cost += weight
            childNodes.append(node)

        if cost < minimumCost:
            minimumCost = cost
            minimumCostChild = childNodes.copy()

    return 0 if minimumCost == 999 else minimumCost, minimumCostChild

solution = {}

def aoStar(n, backTracking):
    if getS(n) >= 0:
```

```python
        minimumCost, childrenList = getMinimumCostChild(n)
        h[n] = minimumCost
        status[n] = len(childrenList)
        solved = True

        for child in childrenList:
            parents[child] = n
            if getS(child) != -1:
                solved = False

        if solved:
            status[n] = -1
            solution[n] = childrenList

        if n != startNode:
            aoStar(parents[n], True)

        if not backTracking:
            for child in childrenList:
                status[child] = 0
                aoStar(child, False)

startNode = 'A'
aoStar(startNode, False)
print(solution)
```

Program 03 Candidate elimination

```
import pandas as pd

dataset = pd.read_csv('./enjoysports.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

POSITIVE = 'yes'

specificH = X[0].copy()
generalH = [['?' for i in range(len(X[0]))] for j in range(len(X[0]))]

for i, row in enumerate(X):
    if y[i] == POSITIVE:
        for j, item in enumerate(row):
            if specificH[j] != item:
                specificH[j] = '?'

    else:
        for j, item in enumerate(row):
            if specificH[j] != item:
                generalH[j][j] = specificH[j]

for i, row in enumerate(generalH):
    for j, item in enumerate(row):
        if item not in specificH:
            generalH[i][j] = '?'

for i in range(generalH.count(['?' for i in range(len(X[0]))])):
    generalH.remove(['?' for i in range(len(X[0]))])

print(specificH)
print(generalH)
```

program 04 ID3

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X / np.amax(X, axis=0)
y = y / 100

inputLayers = 2
hiddenLayers = 3
outputLayers = 1

hW = np.random.uniform(size=(inputLayers, hiddenLayers))
oW = np.random.uniform(size=(hiddenLayers, outputLayers))
hB = np.random.uniform(size=(1, hiddenLayers))
oB = np.random.uniform(size=(1, outputLayers))

epochs = 999999
lr = 0.01

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

def derivativeSignmoid(x):
    return x*(1-x)

for i in range(epochs):
    hiddenLInput = np.dot(X, hW) + hB
    hiddenLOutput = sigmoid(hiddenLInput)

    outputLInput = np.dot(hiddenLOutput, oW) + oB
    outputLOutput = sigmoid(outputLInput)

    outputGrad = derivativeSignmoid(outputLOutput)
    hiddenGrad = derivativeSignmoid(hiddenLOutput)

    EO = y - outputLOutput
    d_output = EO * outputGrad

    EH = d_output.dot(oW.T)
    d_hidden = EH * hiddenGrad

    oW += hiddenLOutput.T.dot(d_output) * lr
    hW += X.T.dot(d_hidden) * lr

print(y)

print(outputLOutput)
```

Program 05 EM

```
import numpy as np
import pandas as pd

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)

X = X / np.amax(X, axis=0)
y = y / 100

inputLN = 2
hiddenLN = 3
outputLN = 1

epochs = 999999
lr = 0.1

hiddenLW = np.random.uniform(size=(inputLN, hiddenLN))
outputLW = np.random.uniform(size=(hiddenLN, outputLN))
hiddenLB = np.random.uniform(size=(1, hiddenLN))
outputLB = np.random.uniform(size=(1, outputLN))

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

def derivativeSigmoid(n):
    return n * (1 - n)

for i in range(epochs):
    hiddenLOut = np.dot(X, hiddenLW) + hiddenLB
    hiddenLAct = sigmoid(hiddenLOut)

    outputLOut = np.dot(hiddenLAct, outputLW) + outputLB
    outputLAct = sigmoid(outputLOut)

    outputGrad = derivativeSigmoid(outputLAct)
    hiddenGrad = derivativeSigmoid(hiddenLAct)

    EO = y - outputLAct
    d_output = EO * outputGrad

    EH = d_output.dot(outputLW.T)
    d_hidden = EH * hiddenGrad

    outputLW += hiddenLAct.T.dot(d_output) * lr
    hiddenLW += X.T.dot(d_hidden) + lr

print(X)   print(y)   print(outputLAct)
```

Program 07 KMeans

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np

iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

model = KMeans(n_clusters=3)
model.fit(X)

plt.figure(figsize=(14,14))
colormap = np.array(['red', 'lime', 'black'])

plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)

plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[gmm_y], s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('Observation: The GMM using EM algorithm based clustering matched the true labels
more closely than the Kmeans.')
```

Program 08 KNN

```
import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split as tts
from sklearn.metrics import classification_report as cr, confusion_matrix as cm,
accuracy_score as acs
from sklearn.neighbors import KNeighborsClassifier

dataset = datasets.load_iris()
X = dataset.data
y = dataset.target

X_train, X_test, y_train, y_test = tts(X, y, test_size=0.1, random_state=0)

classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)

[print(f'Label {index} - {item}') for index, item in enumerate(dataset.target_names)]

y_pred = classifier.predict(X_test)
print(np.concatenate((y_test.reshape(len(y_test),1), y_pred.reshape(len(y_pred),1)),1))

print(f'Accuracy: {acs(y_test, y_pred)*100}%')
print(f'Classification Report:\n{cr(y_test, y_pred)}')
print(f'Confusion Matrix\n{cm(y_test, y_pred)}')
```

program 09 LocalWeightedRegression

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

def kernel(point, xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m)))

    for j in range(m):
        diff = point - X[j]
        weights[j, j] = np.exp(diff * diff.T / (-2.0 * k**2))

    return weights

def localWeight(point, xmat, ymat, k):
    wt = kernel(point, xmat, k)
    W = (X.T * (wt*X)).I * (X.T * wt * ymat.T)
    return W

def localWeightRegression(xmat, ymat, k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)

    for i in range(m):
        ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)

    return ypred

data = pd.read_csv('tips.csv')
colA = np.array(data.total_bill)
colB = np.array(data.tip)
mcolA = np.mat(colA)
mcolB = np.mat(colB)
m = np.shape(mcolB)[1]
one = np.ones((1, m), dtype = int)

X = np.hstack((one.T, mcolA.T))
print(X.shape)

ypred = localWeightRegression(X, mcolB, 0.8)

xsort = X.copy()
xsort.sort(axis=0)
plt.scatter(colA, colB, color='blue')
plt.plot(xsort[:, 1], ypred[X[:, 1].argsort(0)], color='yellow',linewidth=5)
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.show()
```