```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 11 14:12:46 2019

@author: diehl
"""

import numpy as np
import matplotlib.pyplot as plt

class DPSolver_Inscribe():
    def __init__(self, d, n):

        self.n = n
        self.numLayers = n-1                             #Number of interior laye
        self.numNodes = d - self.numLayers               #Number of nodes in each
        self.circlePoints = self.discretizeCircle(d) #Listed sequentially aro

        self.numStates = (self.numLayers*self.numNodes + 2)
        self.statespace = self.constructStateSpace() #np array (Nx2) of all p

        self.values = np.zeros(self.statespace.shape[0]) #Array to hold optim
        self.policy = np.zeros(self.statespace.shape[0]) #Array to hold optim

    #Returns 2xd array with x,y coordinates of circle points in rows
    def discretizeCircle(self,d):
        theta = np.linspace(0,2*np.pi,d,endpoint = False)
        points = np.zeros((d,2))
        points[:,0] = 1.0*np.cos(theta) #x coord
        points[:,1] = 1.0*np.sin(theta) #y coord
        return points

    #Construct our statespace using the circle point coordinates
    def constructStateSpace(self):
        statespace = np.zeros((self.numStates,2))

        #Initial and end points are the same
        statespace[0,:]  = self.circlePoints[0,:]
        statespace[-1,:] = self.circlePoints[0,:]

        #Construct each layer
        for i in range(self.numLayers):
            beginInd = (1+self.numNodes*i)
            endInd = beginInd+self.numNodes

            statespace[beginInd:endInd,:] = self.circlePoints[(1+i):(1+self.n
        return statespace


    #Returns list of possible actions given the current state
    #In this case, it will return a list of state indices we can go to next
    def actionSpace(self, stateIndex):

        #If we are on the last layer, we can only go to the end
        if stateIndex > (self.numLayers-1)*(self.numNodes):
            actions = [self.numStates-1]

        #If we are at the beginning, we can go to the first layer
        elif stateIndex == 0:
            actions = range( (1), (1+self.numNodes) )

        #Otherwise we can go to any point in the next layer that is
        #ahead in the circle
        else:

            #Find which layer we are in using integer division. This
            #will tell us how many segments we have left (if we are in
            #layer 1, we have only completed one segment)
            i = int(stateIndex-1)/int(self.numNodes)

            #Determine which node in the layer we are. We only want to select
            #nodes in the next layer with equal or greater node number as cur
            #node. This will ensure we only move forward around circle.
            nodeNum = (stateIndex-1) - (i)*self.numNodes

            #Return the nodes from the next layer that we can move to
            beginInd = (1+self.numNodes*(i+1))
            endInd = beginInd+(self.numNodes)
            actions = range( beginInd+nodeNum, endInd)

        return actions

    #returns cost of taking a given action from a given state
    #Given an array of state and action (each with two elements)
    def computeCost(self, state, action):
        #action is actually the next state we will go to.
        #Compute length between these points
        return np.sqrt((state[0] - action[0])**2 + (state[1] - action[1])**2)


    def solve(self):

        #Start from end and work to beginning
        sequence = range(self.statespace.shape[0])
        sequence.reverse()

        for stateIndex in sequence:

            if stateIndex == self.numStates-1:
                #Give policy a -1 to indicate we are at the final position
                self.policy[stateIndex] = -1

            else:
                x = self.statespace[stateIndex]

                #Obtain the set of actions available in this state
                #(Returns a set of stateIndices)
                actions = self.actionSpace(stateIndex)

                #Array to hold value of state for each available action
                values = np.zeros(len(actions))

                #Find the value of the state for each action
                for j in range(len(actions)):
                    action = self.statespace[actions[j]]
                    values[j] = (self.computeCost(x,action) + self.values[act

                #Record the optimal value
                self.values[stateIndex] = np.max(values)

                #Record the action that gave the highest value and update our
                #(In this case return the index of the state we would like to
                self.policy[stateIndex] = actions[np.argmax(values)]
        #EndFor



#%%
if __name__ == "__main__":

    n = 3   #Sides of polygon
    d = 6#Our resolution

    DP = DPSolver_Inscribe(d,n)

    ss =  DP.statespace

    #%% Solve the DP problem
    DP.solve()


    #%% Get our policy and reconstruct the path
    policy = DP.policy

    pathx = [DP.statespace[0,0]]
    pathy = [DP.statespace[0,1]]

    i = 0
    nextPointIndex = 0
    while policy[nextPointIndex] != -1:
        nextPointIndex = int(policy[nextPointIndex])
        if nextPointIndex == -1:
            pathx.append(DP.statespace[-1,0])
            pathy.append(DP.statespace[-1,1])
        else:
            pathx.append(DP.statespace[nextPointIndex,0])
            pathy.append(DP.statespace[nextPointIndex,1])


    #%% Plotting
    plt.clf()

    #Plot path
    plt.plot(pathx, pathy,'r')

    #Plot circle discretization
    plt.scatter(DP.circlePoints[:,0],DP.circlePoints[:,1])

    #Plot circle
    theta = np.linspace(0,2*np.pi,200)
    xcirc = 1.0*np.cos(theta) #x coord
    ycirc = 1.0*np.sin(theta) #y coord
    plt.plot(xcirc, ycirc)

    plt.title('Dynamic Programming - Inscribed polygon')
    plt.xlabel('x')
    plt.ylabel('y')
plt.axis('square')
```