```python
#Brendan Neal
#ENPM661
#This Script Will Initialize the Robot dimensions and all Obstacle dimensions in this
Project

import numpy as np


#Robot Information
RobotRadius = 0
Clearance_Temp = 5 #mm
Clearance = RobotRadius + Clearance_Temp

##-----------------------------------Original
Obstacles-------------------------------##
BotRec_X_BL_Corner = 100
BotRec_Y_BL_Corner = 0

BotRec_X_BR_Corner = 150
BotRec_Y_BR_Corner = 0

BotRec_X_TL_Corner = 100
BotRec_Y_TL_Corner = 100

BotRec_X_TR_Corner = 150
BotRec_Y_TR_Corner = 100

Bottom_Rectangle_Points = np.array([[BotRec_X_TL_Corner, BotRec_Y_TL_Corner],
                                    [BotRec_X_BL_Corner, BotRec_Y_BL_Corner],
                                    [BotRec_X_BR_Corner, BotRec_Y_BR_Corner],
                                    [BotRec_X_TR_Corner, BotRec_Y_TR_Corner]])

#Top Rectangle

TopRec_X_BL_Corner = 100
TopRec_Y_BL_Corner = 150

TopRec_X_BR_Corner = 150
TopRec_Y_BR_Corner = 150

TopRec_X_TL_Corner = 100
TopRec_Y_TL_Corner = 250

TopRec_X_TR_Corner = 150
TopRec_Y_TR_Corner = 250

Top_Rectangle_Points = np.array([[TopRec_X_TL_Corner, TopRec_Y_TL_Corner],
                                    [TopRec_X_BL_Corner, TopRec_Y_BL_Corner],
                                    [TopRec_X_BR_Corner, TopRec_Y_BR_Corner],
                                    [TopRec_X_TR_Corner, TopRec_Y_TR_Corner]])


#Middle Hexagon
#Theory: A hexagon is made up of 6 equalateral triangles.
CenterX = 300
```

```
CenterY = 125

X_Top_Right_Corner = int(CenterX + 75*np.cos(np.deg2rad(30)))
Y_Top_Right_Corner = int(CenterY + 75*np.sin(np.deg2rad(30)))

X_Bottom_Right_Corner = int(CenterX + 75*np.cos(np.deg2rad(-30)))
Y_Bottom_Right_Corner = int(CenterY + 75*np.sin(np.deg2rad(-30)))

X_TopMost_Corner = int(CenterX )
Y_TopMost_Corner = int(CenterY + 75)

X_BottomMost_Corner = int(CenterX)
Y_BottomMost_Corner = int(CenterY - 75)

X_Top_Left_Corner = int(CenterX - 75*np.cos(np.deg2rad(30)))
Y_Top_Left_Corner = int(CenterY + 75*np.sin(np.deg2rad(30)))

X_Bottom_Left_Corner = int(CenterX - 75*np.cos(np.deg2rad(-30)))
Y_Bottom_Left_Corner = int(CenterY + 75*np.sin(np.deg2rad(-30)))

Hexagon_Points = np.array([[X_BottomMost_Corner, Y_BottomMost_Corner],
                           [X_Bottom_Right_Corner, Y_Bottom_Right_Corner],
                           [X_Top_Right_Corner, Y_Top_Right_Corner],
                           [X_TopMost_Corner, Y_TopMost_Corner],
                           [X_Top_Left_Corner, Y_Top_Left_Corner],
                           [X_Bottom_Left_Corner, Y_Bottom_Left_Corner]
                           ], dtype=np.int32)



#Triangle
Bot_Tri_CornerX = 460
Bot_Tri_CornerY = 25

Top_Tri_CornerX = 460
Top_Tri_CornerY = 225

Right_Tri_CornerX = 510
Right_Tri_CornerY = 125

Triangle_Points = np.array([[Bot_Tri_CornerX, Bot_Tri_CornerY],
                            [Top_Tri_CornerX, Top_Tri_CornerY],
                            [Right_Tri_CornerX, Right_Tri_CornerY]])


##-----------------------------Extended Obstacle
Spaces------------------------------##

#Bottom Rectangle

BotRec_X_BL_Corner_Ext = 100 - Clearance
BotRec_Y_BL_Corner_Ext = 0

BotRec_X_BR_Corner_Ext = 150 + Clearance
BotRec_Y_BR_Corner_Ext = 0
```

```
BotRec_X_TL_Corner_Ext = 100 - Clearance
BotRec_Y_TL_Corner_Ext = 100 + Clearance

BotRec_X_TR_Corner_Ext = 150 + Clearance
BotRec_Y_TR_Corner_Ext = 100 + Clearance

Bottom_Rectangle_Points_OBS = np.array([[BotRec_X_TL_Corner_Ext,
BotRec_Y_TL_Corner_Ext],
                                    [BotRec_X_BL_Corner_Ext, BotRec_Y_BL_Corner_Ext],
                                    [BotRec_X_BR_Corner_Ext, BotRec_Y_BR_Corner_Ext],
                                    [BotRec_X_TR_Corner_Ext, BotRec_Y_TR_Corner_Ext]])

#Top Rectangle

TopRec_X_BL_Corner_Ext = 100 - Clearance
TopRec_Y_BL_Corner_Ext = 150 - Clearance

TopRec_X_BR_Corner_Ext = 150 + Clearance
TopRec_Y_BR_Corner_Ext = 150 - Clearance

TopRec_X_TL_Corner_Ext = 100 - Clearance
TopRec_Y_TL_Corner_Ext = 250

TopRec_X_TR_Corner_Ext = 150 + Clearance
TopRec_Y_TR_Corner_Ext = 250

Top_Rectangle_Points_OBS = np.array([[TopRec_X_TL_Corner_Ext, TopRec_Y_TL_Corner_Ext],
                                    [TopRec_X_BL_Corner_Ext, TopRec_Y_BL_Corner_Ext],
                                    [TopRec_X_BR_Corner_Ext, TopRec_Y_BR_Corner_Ext],
                                    [TopRec_X_TR_Corner_Ext, TopRec_Y_TR_Corner_Ext]])


#Middle Hexagon
#Theory: A hexagon is made up of 6 equalateral triangles.
CenterX = 300
CenterY = 125

X_Top_Right_Corner_Ext = int(CenterX + 75*np.cos(np.deg2rad(30)) + Clearance)
Y_Top_Right_Corner_Ext = int(CenterY + 75*np.sin(np.deg2rad(30)))

X_Bottom_Right_Corner_Ext = int(CenterX + 75*np.cos(np.deg2rad(-30)) + Clearance)
Y_Bottom_Right_Corner_Ext = int(CenterY + 75*np.sin(np.deg2rad(-30)))

X_TopMost_Corner_Ext = int(CenterX )
Y_TopMost_Corner_Ext = int(CenterY + 75 + Clearance)

X_BottomMost_Corner_Ext = int(CenterX )
Y_BottomMost_Corner_Ext = int(CenterY - 75 - Clearance)

X_Top_Left_Corner_Ext = int(CenterX - 75*np.cos(np.deg2rad(30)) - Clearance)
Y_Top_Left_Corner_Ext = int(CenterY + 75*np.sin(np.deg2rad(30)))

X_Bottom_Left_Corner_Ext = int(CenterX - 75*np.cos(np.deg2rad(-30)) - Clearance)
Y_Bottom_Left_Corner_Ext = int(CenterY + 75*np.sin(np.deg2rad(-30)))
```

```
Hexagon_Points_OBS = np.array([[X_BottomMost_Corner_Ext, Y_BottomMost_Corner_Ext],
                              [X_Bottom_Right_Corner_Ext, Y_Bottom_Right_Corner_Ext],
                              [X_Top_Right_Corner_Ext, Y_Top_Right_Corner_Ext],
                              [X_TopMost_Corner_Ext, Y_TopMost_Corner_Ext],
                              [X_Top_Left_Corner_Ext, Y_Top_Left_Corner_Ext],

                              [X_Bottom_Left_Corner_Ext, Y_Bottom_Left_Corner_Ext]
                              ], dtype=np.int32)



#Triangle
Bot_Tri_CornerX_Ext = 460 - Clearance
Bot_Tri_CornerY_Ext = 25 - 3*Clearance #Needed to Tack on Extra to get the proper side
extension.

Top_Tri_CornerX_Ext = 460 - Clearance
Top_Tri_CornerY_Ext = 225 + 3*Clearance #Needed to Tack on Extra to get the proper
side extension.

Right_Tri_CornerX_Ext= 510 + 2*Clearance #Needed to Tack on Extra to get the proper
side extension.
Right_Tri_CornerY_Ext = 125

Triangle_Points_OBS = np.array([[Bot_Tri_CornerX_Ext, Bot_Tri_CornerY_Ext],
                               [Top_Tri_CornerX_Ext, Top_Tri_CornerY_Ext],
                               [Right_Tri_CornerX_Ext, Right_Tri_CornerY_Ext]])
```

```
#Brendan Neal
#ENPM661 Project 2
#Directory ID: bneal12


##---------Importing Functions----------##
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
import math
import timeit
import queue
from queue import PriorityQueue

##----Importing Variables from Other Files-----##
from ObstacleDefinitions import *

##-------Generating Node Class-----------##
class Node():
    #Initializing Function
    def __init__(self, state, parent, move, C2C):
        self.state = state
        self.parent = parent
        self.move = move
        self.C2C = C2C
    #---Methods for this Class---#
    def ReturnState(self): #Returns Node State X and Y
        return self.state

    def ReturnMove(self): #Returns the Move of that Node (change from prior node)
        return self.move

    def ReturnParent(self): #Returns the Parent Node
        return self.parent

    def ReturnParentState(self): #Returns the Parent Node's State
        if self.ReturnParent() is None:
            return None
        return self.ReturnParent().ReturnState()

    def ReturnCost(self): #Returns the Cost Leading up to the Node
        return self.C2C

    def __lt__(self, other): #OOP Definition for Less than. Required for Priority
Queue.
        return self.C2C < other.C2C

    ##-------------BACKTRACKING FUNCTION Integrated into Class--------##
    def ReturnPath(self):
        CompletedMoves = [] #Initialize Move Array
        NodePath = [] #Initialize the Node Path
        CurrentNode = self
        while(CurrentNode.ReturnMove() is not None): #For move that a Node has made
            CompletedMoves.append(CurrentNode.ReturnMove()) #Append the previous move
            NodePath.append(CurrentNode) #Append Node to Path
```

```python
            CurrentNode = CurrentNode.ReturnParent() #Backtrack to the Parent before repeating Process
        NodePath.append(CurrentNode) #Append the starting point after path is derived.
        NodePath.reverse() #Reverse Order to get front to back path
        CompletedMoves.reverse() #Reverse Order to get front to back path

        return CompletedMoves, NodePath


##------------------Defining my Check in Workspace? Function-------##
def CheckInWorkspace(CurrentX, CurrentY):
    WsX_Extended = 600-1 #Index the workspace back 1 slot for indexing
    WsY_Extended = 250-1 #Index the workspace back 1 slot for indexing
    if (CurrentX > WsX_Extended or int(CurrentX)<1 or int(CurrentY)<1 or CurrentY>WsY_Extended): #If outside of workspace
        return 1 # Logic will be used later!!!
    return 0

##-------------------Defining my Check in Obstacles? Function-------##
def CheckInObstacles(CurrentX, CurrentY, Rectangle1OS, Rectangle2OS, TriangleOS, HexOS):
    Rec1Check = cv.pointPolygonTest(Rectangle1OS, (CurrentX, CurrentY), False) #Check if in Rectangle1
    Rec2Check = cv.pointPolygonTest(Rectangle2OS, (CurrentX, CurrentY), False) #Check if in Rectangle2
    TriCheck = cv.pointPolygonTest(TriangleOS, (CurrentX, CurrentY), False) #Check if in Triangle
    HexCheck = cv.pointPolygonTest(HexOS, (CurrentX, CurrentY), False) #Check if in Hexagon
    #Return 1s or 0s used for Logic Later
    if Rec1Check > 0:
        return 1
    if Rec2Check > 0:
        return 1
    if TriCheck > 0:
        return 1
    if HexCheck > 0:
        return 1
    else:
        return 0

##---------------Defining my Action Set within one Function----------##
''' This function generates the possible moves and checks whether the move is valid by checking if the move leads into
 an obstacle, out of the workspace, or the move takes you to the previous parent all in one go.'''
def GeneratePossibleMoves(CurrentNode):
    x, y = CurrentNode.ReturnState() #Get Current State
    moves = ['Up','UpRight', 'Right', 'DownRight', 'Down', 'DownLeft','Left', 'UpLeft'] #Moves that the Node Could have Done
    possible_moves = ['Up','UpRight', 'Right', 'DownRight', 'Down', 'DownLeft','Left', 'UpLeft'] #Moves that the Node could do Next
    move_x = [x, x+1, x+1, x+1, x, x-1, x-1, x-1] #Mathematical Move the Node Could do in X
    move_y = [y+1, y+1, y, y-1, y-1, y-1, y, y+1] #Mathematical Move the Node Could do in Y
    for move in range(len(moves)): #For eeach move
```

```
        if (CheckInObstacles(move_x[move], move_y[move],Bottom_Rectangle_Points_OBS
,Top_Rectangle_Points_OBS, Triangle_Points_OBS, Hexagon_Points_OBS) #Check if move
goes into an obstacle
            or CheckInWorkspace(move_x[move], move_y[move]) #Check if a move goues
outside of workspace
            or CurrentNode.ReturnParentState() == [move_x[move], move_y[move]]):
#Check if the move takes you back to the parent node state. (AKA in the Closed List)
            possible_moves.remove(moves[move]) #Remove the possible move if any of the
conditions are met.
    return possible_moves

#----------------Defining my Map Coloring Function---------------##
def WSColoring(Workspace, Location, Color):
    x,_,_ = Workspace.shape #Get Shape of Workspace
    translation_y = Location[0] #Where in Y
    translation_x = x - Location[1] - 1 #Where in X - (Shifts origin from top left to
bottom right when plotting!)
    Workspace[translation_x,translation_y,:] = Color #Change the Color to a set Color
    return Workspace

##--------------Defining my GetInitialState Function-------------##
def GetInitialState():
    print("Enter Initial Node X and Y, separated by spaces: ")
    Init_State=[int(x) for x in input().split()]
    return Init_State


##--------------Defining my GetGoalState Function-------------##
def GetGoalState():
    print("Enter Goal Node X and Y, separated by spaces: ")
    Goal_State=[int(x) for x in input().split()]
    return  Goal_State

#----------------Defining my Compare to Goal Function-----------------##

def CheckGoal(CurrentNode, GoalNode):
    if np.array_equal(CurrentNode, GoalNode) or CurrentNode == GoalNode: #Double Check
-- If the mathematical array is equal OR if the two Node Classes are Equal.
        return True
    else:
        return False

##-----------------------"Main Script"---------------------------##

#Area Information
SizeAreaX = 600
SizeAreaY = 250

video_name = ('dijkstra_brendan_neal') #Initialize Video Object
fourcc = cv.VideoWriter_fourcc(*"mp4v") #Initialize Video Writer using fourcc
video = cv.VideoWriter(str(video_name)+".mp4", fourcc, 300, (SizeAreaX, SizeAreaY))
#Initialize the Name, method, frame rate, and size of Video.
Workspace = np.zeros((SizeAreaY, SizeAreaX,3), dtype = np.uint8) #Initialize the
workspace as 0s at first. Integer data type to write to video.
Workspace[:,:] = (0,0,0) #Set all colors to black.

#Drawing Extended Obstacle Space Using cv2.fillPoly
```

```
    #Color is RED
    Rectangle1_Obs_Space = cv.fillPoly(Workspace, [Bottom_Rectangle_Points_OBS],
    [255,0,0])
    Rectangle2_Obs_Space = cv.fillPoly(Workspace, [Top_Rectangle_Points_OBS], [255,0,0])
    Triangle_Obs_Space = cv.fillPoly(Workspace, [Triangle_Points_OBS], [255,0,0])
    Hexagon_Obs_Space = cv.fillPoly(Workspace, [Hexagon_Points_OBS], [255,0,0])

    #Drawing Original Obstacles using cv2.fillPoly -- Drawn After to Layer Properly
    #Color is BLUE
    Rectangle1= cv.fillPoly(Workspace, [Bottom_Rectangle_Points], [0,0,255])
    Rectangle2 = cv.fillPoly(Workspace, [Top_Rectangle_Points], [0,0,255])
    Triangle= cv.fillPoly(Workspace, [Triangle_Points], [0,0,255])
    Hexagon = cv.fillPoly(Workspace, [Hexagon_Points], [0, 0, 255])

    plt.imshow(Workspace, origin='lower') #Show the initial state.
    plt.show()

    InitState = GetInitialState() #Grab Initial State
    GoalState = GetGoalState() #Grab Goal State

    #Check to see if the initial state is in an obstacle
    if CheckInObstacles(InitState[0], InitState[1], Bottom_Rectangle_Points_OBS ,
    Top_Rectangle_Points_OBS, Triangle_Points_OBS, Hexagon_Points_OBS):
        print("Initial State is in an obstacle, try again!")
        exit()

    #Check to see if the goal state is in an obstacle.
    if CheckInObstacles(GoalState[0], GoalState[1], Bottom_Rectangle_Points_OBS ,
    Top_Rectangle_Points_OBS, Triangle_Points_OBS, Hexagon_Points_OBS):
        print("Goal State is in an obstacle, try again!")
        exit()

    #Check to see if the initial state falls within the workspace
    if CheckInWorkspace(InitState[0], InitState[1]):
        print("Initial State is Outside of Workspace, try again!")
        exit()

    #Check to see if the goal state falls within the workspace.
    if CheckInWorkspace(GoalState[0], GoalState[1]):
        print("Goal State is Outside of Workspace, try again!")
        exit()

    Open_List = PriorityQueue() #Initialize list using priority queue.
    starting_node = Node(InitState, None, None, 0) #Generate starting node based on the
    initial state given above.
    Open_List.put((starting_node.ReturnCost(), starting_node)) #Add to Open List
    GoalReach = False #Initialze Goal Check Variable



    Closed_List= []#Initialize Closed List of nodes, size of workspace, and setting their
    cost to infinity to allow for Dijkstra searching.
    Map_Weights = np.array([[Node([i,j],None, None, math.inf)for j in range(SizeAreaY)]for
    i in range(SizeAreaX)]) #Map Weights for Comparison

    Working_Space = WSColoring(Workspace, InitState, [0,255,0]) #Plot initial state in
    GREEN on Workspace.
```

```python
Working_Space = WSColoring(Workspace, GoalState, [0,255,0]) #Plot goal state in GREEN on Workspace.


##--------------------------CONDUCT DIJKSTRA---------------------##

starttime = timeit.default_timer() #Start the Timer when serch starts
print("Dijkstra Search Starting!!!!")

while not (Open_List.empty() and GoalReach): #While the open list is not empty, and the goal has not been reached.
    current_node = Open_List.get()[1] #Grab first (lowest cost) item from Priority Queue.
    i, j = current_node.ReturnState() #Get the State of node.
    Closed_List.append([i,j]) #Add popped node location to Closed List
    Working_Space = WSColoring(Working_Space, current_node.ReturnState(), [255, 255, 255]) #Draw Explpred Node White
    video.write(cv.cvtColor(Working_Space, cv.COLOR_RGB2BGR)) #Write exploration to video.

    XMoves = {'Up':i ,'UpRight':i+1, 'Right':i+1, 'DownRight':i+1, 'Down':i, 'DownLeft':i-1,'Left':i-1, 'UpLeft':i-1} #Create Dictionary for X Moves
    YMoves = {'Up':j+1 ,'UpRight':j+1, 'Right':j, 'DownRight':j-1, 'Down':j-1, 'DownLeft':j-1,'Left':j, 'UpLeft':j+1} #Create Dictionary for Y Moves
    Moves_C2C = {'Up':1 ,'UpRight':1.4, 'Right':1, 'DownRight':1.4, 'Down':1, 'DownLeft':1.4,'Left':1, 'UpLeft':1.4} #Create a dictionary for the costs based on move.

    goalreachcheck = CheckGoal(current_node.ReturnState(), GoalState) #Check if we have reached goal.

    if goalreachcheck: #If we have reached goal node.
        print("Goal Reached!")
        print("Total Cost:", current_node.ReturnCost()) #Print Total Cost
        MovesPath, Path = current_node.ReturnPath() #BackTrack to find path.

        for nodes in Path:
            Position = nodes.ReturnState() #Get the state of the nodes in the backtracked path.
            Working_Space = WSColoring(Working_Space, Position, [255,0,255]) #Color them magenta
            video.write(cv.cvtColor(Working_Space, cv.COLOR_RGB2BGR)) #Write to video.

        ##----------Extend Video a Little Longer to see Everything--------##
        for i in range(200):
            video.write(cv.cvtColor(Working_Space, cv.COLOR_RGB2BGR))

        break

    else: #If you have NOT reached the goal node
        NewNodes = GeneratePossibleMoves(current_node) #Generate New Nodes from the possible moves current node can take.
        Parent_Cost = current_node.ReturnCost() #Get the Cost of the Parent.
        if NewNodes not in Closed_List: #Check to see if the new node position is currently in the closed list
            for move in NewNodes: #For each new node generated by the possible moves.
                Child_Position = [XMoves.get(move), YMoves.get(move)] #Get the Child Position
```

```
                print("Possible Moves Are:", Child_Position) #Print Possible moves to
show algorithm is searching.
                C2C = Parent_Cost + Moves_C2C.get(move) #Get New Cost from the parent
and add it to the associated cost from dictionary.
                if(C2C < Map_Weights[Child_Position[0],
Child_Position[1]].ReturnCost()): #Compare the Costs, and if the current cost is less
than the other cost to get there
                    New_Child = Node(Child_Position, current_node, move, C2C)
#Regenerate the Node with the Lower Cost
                    Map_Weights[Child_Position[0], Child_Position[1]] = New_Child
#Replace in Weighted Map
                    Open_List.put((New_Child.ReturnCost(), New_Child)) #Queue up the
Child in the open list.
    if goalreachcheck: #If you reach goal
        break #Break the Loop

stoptime = timeit.default_timer() #Stop the Timer, as Searching is complete.

print("The Algorithm took", stoptime-starttime, "seconds to solve.")

video.release()

plt.imshow(Working_Space) #Show Final Solved Path
plt.show()
```