```
### IMPLEMENTATION OF A* ALGORITHM FOR A MOBILE ROBOT ###
#====================================================================================================================================#


## Import Necesary Packages ##
import cv2 as cv
import heapq as hq
import math
import numpy as np
import time

#--------------------------------------------------------------------------------------------------------------------------------
---#

# ## Get input for clearance (units) from the obstacle and Step Size of the mobile robot ##
clear = int(float(input("Clearance from obstacles and walls: ")))
radius = int(float(input("Radius of mobile robot: ")))
s = int(float(input("Step size of the mobile robot in range [1,10]: ")))
if s<2:
    s=2
w = int(float(input("Heuristic weightage (Enter 1 for default A* execution): ")))
clearance = clear + radius
round = round(clearance/2) + clearance%2
border = round//2

## Define Map ##
map = np.ones((500, 1200, 3), dtype='uint8')*255

#Wall Barriers
for i in range(0,1200):
    for k in range(0,round):
        map[k][i] = (0,0,0)
for i in range(0,1200):
    for k in range(500-round,500):
        map[k][i] = (0,0,0)
for i in range(0,round):
    for k in range(0,500):
        map[k][i] = (0,0,0)
for i in range(1200-round,1200):
    for k in range(0,500):
        map[k][i] = (0,0,0)




#Left Most Rectangle Object
# Outer Black Rectangle
for i in range(99-round,175+round):
    for k in range(0,400+round):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(99,175):
    for k in range(0,399):
        map[k][i] = (255,0,0)

#Rectangle 2:
#Outer Black Rectangle
for i in range(274-round,350+round):
    for k in range(99-round,500):
        map[k][i] = (0,0,0)
# Inner Blue Rectangle
for i in range(274,350):
    for k in range(99,500):
        map[k][i] = (255,0,0)

#Right Obstacle (HorseShoe)
#Outer Black Rectangle
for i in range(1019-round,1100+round):
    for k in range(49-round,450+round):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(1019,1100):
    for k in range(49,450):
        map[k][i] = (255,0,0)

#Outer Black Rectangle
for i in range(899-round,1020):
    for k in range(374-round,450+round):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(899,1020):
    for k in range(374,450):
        map[k][i] = (255,0,0)

#Outer Black Rectangle
for i in range(899-round,1020):
    for k in range(49-round,120+round):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(899,1020):
    for k in range(49,120):
        map[k][i] = (255,0,0)


# Hexagonal Polygon (Blue) with 5mm Border (Black)
#Outer Black Boundary
for i in range(519-round,780+round):
```

```
    for k in range(174,325):
        map[k][i] = (0,0,0)
#Interior Blue Rectangle
for i in range(519,780):
    for k in range(174,325):
        map[k][i] = (255,0,0)

#TopLeftTriangle
#Outer Black Boundary
GRat = 75/130
for i in range(519-round,650):
    for k in range(324,400+round):
        XTemp = i-(519-round)
        YTemp = k-324
        if ((XTemp+round)*GRat)>YTemp:
            map[k][i] = (0,0,0)
#Inner Blue Triangle
GRat = 75/130
for i in range(519,650):
    for k in range(324,400):
        XTemp = i-519
        YTemp = k-324
        if (XTemp*GRat)>YTemp:
            map[k][i] = (255,0,0)

#Top Right Triangle
#Outer Black Boundary
GRat = -75/130
for i in range(649,780+round):
    for k in range(324,400+round):
        XTemp = i-649
        YTemp = k-324
        if (XTemp*GRat)+(75+round)>YTemp:
            map[k][i] = (0,0,0)
#Inner Blue Traingle
GRat = -75/130
for i in range(649,780):
    for k in range(324,400):
        XTemp = i-649
        YTemp = k-324
        if (XTemp*GRat)+75>YTemp:
            map[k][i] = (255,0,0)

#Bottom Right Triangle
#Outer Black Boundary
GRat = 75/130
for i in range(649,780+round):
    for k in range(99-round,175):
        XTemp = i-649
        YTemp = k-(99-round)
        if (XTemp*GRat)<YTemp:
            map[k][i] = (0,0,0)
#Inner Blue Traingle
GRat = 75/130
for i in range(649,780):
    for k in range(99,175):
        XTemp = i-649
        YTemp = k-99
        if (XTemp*GRat)<YTemp:
            map[k][i] = (255,0,0)

#Bottom Left Triangle
# Outer Black Boundary
GRat = -75/130
for i in range(519-round,650):
    for k in range(99-round,175):
        XTemp = i-(519-round)
        YTemp = k-(99-round)
        if ((XTemp+round)*GRat)+(75+round)<YTemp:
            map[k][i] = (0,0,0)
# Inner Blue Triangle
GRat = -75/130
for i in range(519,650):
    for k in range(99,175):
        XTemp = i-519
        YTemp = k-99
        if (XTemp*GRat)+75<YTemp:
            map[k][i] = (255,0,0)


#------------------------------------------------------------------------------------------------------------------------------------
---#

## Define a 'Node' class to store all the node informations ##
class Node():
    def __init__(self, coc=None, cost=None, parent=None, free=False, closed=False):
        # Cost of Coming from 'source' node
        self.coc = coc
        # Total Cost = Cost of Coming from 'source' node + Cost of Going to 'goal' node
        self.cost = cost
        # Index of Parent node
        self.parent = parent
        # Boolean variable that denotes (True) if the node is in 'Free Space'
        self.free = free
        # Boolean variable that denotes (True) if the node is 'closed'
        self.closed = closed
```

```python
#-----------------------------------------------------------------------------------------------------------------------------------
---#

## Initiate an array of all possible nodes from the 'map' ##
print("Building Workspace for Mobile Robot........!")
nodes = np.zeros((map.shape[0], map.shape[1], 12), dtype=Node)
for row in range(nodes.shape[0]):
    for col in range(nodes.shape[1]):
        for angle in range(12):
            nodes[row][col][angle] = Node()
            # If the node index is in the 'Free Space' of 'map', assign (True)
            if map[row][col][2] == 255:
                nodes[row][col][angle].free = True
                continue

#-----------------------------------------------------------------------------------------------------------------------------------
---#

## Define a 'Back-Tracking' function to derive path from 'source' to 'goal' node ##
def backTrack(x,y,l):
    print("Backtracking!!")
    track = []
    while True:
        track.append((y,x,l))
        if nodes[y][x][l].parent == None:
            track.reverse()
            break
        y,x,l = nodes[y][x][l].parent
    print("Path created!")
    return track

#-----------------------------------------------------------------------------------------------------------------------------------
---#

## Get 'Source' and 'Goal' node and check if it's reachable ##
while True:
    print("Node is a point (X,Y) in cartesian plane for Xâ[0,1200] and Yâ[0,500]")
    x1 = int(float(input("X - Coordinate of Source Node: ")))
    y1 = int(float(input("Y - Coordinate of Source Node: ")))
    x2 = int(float(input("X - Coordinate of Goal Node: ")))
    y2 = int(float(input("Y - Coordinate of Goal Node: ")))
    print("Orientation of nodes (in degrees) is the direction of mobile robot from [180, 150, 120, .., 30, 0, -30, -60, .., -150]")
    a1 = int(float(input("Orientation of Source Node (in degrees): ")))
    a2 = int(float(input("Orientation of Goal Node (in degrees): ")))
    # Convert a1, a2 to the corresponding layer number in the 3D array of 'nodes'
    l = []
    for a in [a1,a2]:
        if a >= 0 and a <= 180:
            l.append(a//30)
        elif a < 0 and a > -180:
            l.append((360+a)//30)
        else:
            print("Enter valid orientation from the list above")
            break
    l1, l2 = l

    # Check if the given coordinates are in the 'Free Space'
    if nodes[500-y1][x1][l1].free and nodes[500-y2][x2][l2].free:
        print("Executing path planning for the given coordinates........!!")
        y1 = 500-y1
        y2 = 500-y2
        break
    else:
        print("The given coordinates are not reachable. Try again with different coordinates")

#-----------------------------------------------------------------------------------------------------------------------------------
---#

## Create a copy of map to store the search state for every 500 iterations ##
img = map.copy()
# Mark 'source' and 'goal' nodes on the 'img'
xs, ys = x1, y1
xg, yg = x2, y2
cv.circle(img,(xs,ys),radius,(0,255,255),-1) # Source --> 'Yellow'
cv.circle(img,(xg,yg),radius,(255,0,255),-1) # Goal --> 'Purple'
# Write out to 'dijkstra_output.avi' video file
out = cv.VideoWriter('A*_output.mp4', cv.VideoWriter_fourcc(*'mp4v'), 60, (1200,500))
out.write(img)

#-----------------------------------------------------------------------------------------------------------------------------------
---#

## Define a function to search all the nodes from 'source' to 'goal' node using Dijkstra's Search ##

# Initiate a Priority Queue / Heap Queue with updatable priorities to store all the currently 'open nodes' for each iteration
open_nodes = []

iterations = 0
start = time.time()
while True:

    iterations += 1
    if nodes[y1][x1][l1].parent != None:
        # Change the color of all pixels explored to 'green', except 'source' and 'goal' colors
        parent_y,parent_x,parent_l = nodes[y1][x1][l1].parent
        cv.line(img,(parent_x,parent_y),(x1,y1),(0,255,0),1)
        # Write search state 'img' for every 500 iterations
```

```
        if iterations/500 == iterations//500:
            # Mark 'source' and 'goal' nodes on the 'img'
            cv.circle(img,(xs,ys),radius,(0,255,255),-1)
            cv.circle(img,(xg,yg),radius,(255,0,255),-1)
            out.write(img)

    # 'nodes[y1][x1][l1]' --> current 'open' node
    if nodes[y1][x1][l1].parent == None:
        # Cost to come for the source node is '0' itself
        nodes[y1][x1][l1].coc = 0
        # Update Total Cost with Cost to Come and Cost to go to the goal is the 'euclidean distance' times the 'Heuristic Weightage'
        nodes[y1][x1][l1].cost = (nodes[y1][x1][l1].coc + (math.sqrt((y2-y1)**2 + (x2-x1)**2))*w)

    # Verify if the current 'open' node is in threshold of 'goal' node (1.5 units radius)
    if l1 == l2 and ((y2-y1)**2 + (x2-x1)**2) <= ((1.5*radius)**2):
        print("Path Planning Successfull!!!")
        # Call 'Back-Tracking' function
        path = backTrack(x1,y1,l1)
        break

    # If the current 'node' is not in the threshold region of 'goal' node, 'close' the node and explore neighbouring nodes
    else:
        # Close the node and explore corresponding neighbours
        nodes[y1][x1][l1].closed = True
        # Perform All Possible Action Sets from: {(-60, -30, 0, 30, 60)}
        # Get neighbouring nodes to the current 'open' node and add it to the Heap Queue 'open_nodes'

        # Initiate a list to iterate over 'actions' sets with Cost of Come = Step Size (s)
        actions = [0, 30, 60, -30, -60]
        # angle of orientation of robot 'theta' in degrees
        theta = 30*l1
        deg = np.pi/180

        # Cost to come of the current open node (y1,x1)
        dist = nodes[y1][x1][l1].coc

        # Iterate over 'actions' list
        for action in actions:
            phi = theta+action
            y = int(y1 + s*np.sin(phi*deg))
            x = int(x1 + s*np.cos(phi*deg))
            l = int(phi/30)

            if l >= 12:
                l = l-12
            elif l < 0:
                l = 12+l

            # If the new node exceeds from the map
            if x >= 1200 or y >= 500:
                continue
            # If the neighbour node is already 'closed', iterate over next action
            if nodes[y][x][l].closed:
                continue
            # Check if new node is in 'Free Space'
            if nodes[y][x][l].free:
                # Cost to Come 'c2c' corresponding to each angle
                if action < 0:
                    action = -1*action
                c2c = s / np.cos(action*deg)
                # Cost to Go 'c2g'
                c2g = (math.sqrt((y2-y)**2 + (x2-x)**2))*w
                # If the new node is visited for the first time, update '.coc', '.cost' and '.parent'
                if nodes[y][x][l].coc == None:
                    nodes[y][x][l].coc = dist + c2c
                    nodes[y][x][l].cost = (nodes[y][x][l].coc + c2g)
                    cost = nodes[y][x][l].cost
                    nodes[y][x][l].parent = (y1,x1,l1)
                    # Add new node to 'open_nodes'
                    hq.heappush(open_nodes, (cost, (y, x, l)))
                # If the new node was already visited, update '.coc' and '.parent' only if the new_node.coc is less than the existing 
value
                elif (dist + c2c) < nodes[y][x][l].coc:
                    nodes[y][x][l].coc = dist + c2c
                    cost = (nodes[y][x][l].coc + c2g)
                    nodes[y][x][l].parent = (y1,x1,l1)
                    # Update 'priority' of new node in 'open_nodes'
                    hq.heappush(open_nodes, (cost, (y, x, l)))

        while True:
            # Pop next element from 'open_nodes'
            (priority, node) = hq.heappop(open_nodes)
            y = node[0]
            x = node[1]
            l = node[2]
            # If priority is greater than node.cost, pop next node
            if priority == (nodes[y][x][l].cost) and nodes[y][x][l].closed == False:
                break

        # Update x1 and y1 for next iteration
        y1 = y
        x1 = x
        l1 = l

# Write last frame to video file
# Mark 'source' and 'goal' nodes on the 'img'
cv.circle(img,(xs,ys),radius,(0,255,255),-1)
```

```
cv.circle(img,(xg,yg),radius,(255,0,255),-1)
out.write(img)
print("Number of iterations: ",iterations)
#-----------------------------------------------------------------------------------------------------------------------------------
---#

end = time.time()
runntime = end-start
print("Path Planning Time: ",runntime)

#-----------------------------------------------------------------------------------------------------------------------------------
---#

# Iterate over 'optimalPath' and change each pixel in path to 'Red'
count = 0
for i in range(0,len(path)-1):
    count+=1
    pt1 = (path[i][1], path[i][0])
    pt2 = (path[i+1][1], path[i+1][0])
    cv.line(img,pt1,pt2,(0,0,255),1)
    # Write to video file for every 2 iterations
    if count/2 == count//2:
       out.write(img)

# Last frame in path travelling
for i in range(120):
    out.write(img)

# Display 'Optimal Path' for 5 seconds
cv.imshow("Optimal Path", img)
cv.waitKey(5*1000)

out.release()

#====================================================================================================================================#
```