```
# Part01:
### 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
import matplotlib.pyplot as plt

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

## Initialise ##

WheelRadius = 3.3  #33mm
RobotRadius = 22.0 # 220 mm
WheelDistance = 28.7 #287mm

while True:
    RPM1 = int(float(input("Input RPM1: ")))
    RPM2 = int(float(input("Input RPM2: ")))
    # RPM1 = 50
    # RPM2 = 100
    if RPM1 <= 0 or RPM2 <= 0:
        print("Enter a positive non-zero value.")
    if RPM1 < RPM2:
        Min = min(RPM1, RPM2, RPM2-RPM1)
        break
    elif RPM1 > RPM2:
        Min = min(RPM1, RPM2, RPM1-RPM2)
        break
    else:
        print("Enter different RPM values for most effective execution.")

actionSet = [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2,RPM2],[RPM1,RPM2],[RPM2,RPM1]]

## Get input for clearance (units) from the obstacle ##
clear = int(float(input("Clearance from obstacles and walls (in cm): ")))
# clear = 1

# w = int(float(input("Heuristic weightage (Enter 1 for default A* execution): ")))
w = 1
# 'threshold' within which the goal node must be reached = 3 units
threshold = 3

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

## Define Map ##

clearance = clear + RobotRadius
rounded = round(clearance)

map = np.ones((200, 600, 3), dtype='uint8')*255
#Wall Barriers
for i in range(0,600):
    for k in range(0,rounded):
        map[k][i] = (0,0,0)
for i in range(0,600):
    for k in range(200-rounded,200):
        map[k][i] = (0,0,0)
for i in range(0,rounded):
    for k in range(0,200):
        map[k][i] = (0,0,0)
for i in range(600-rounded,600):
    for k in range(0,200):
        map[k][i] = (0,0,0)

#Left Most Rectangle Object
# Outer Black Rectangle
for i in range(149-rounded,175+rounded):
    for k in range(0,100+rounded):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(149,175):
    for k in range(0,100):
        map[k][i] = (255,0,0)
#Right Most Rectangle Object
# Outer Black Rectangle
for i in range(249-rounded,275+rounded):
    for k in range(100-rounded,200):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(249,275):
    for k in range(100,200):
        map[k][i] = (255,0,0)

#Circle centered at 420,80, r of 60

#Inner Blue Rectangle
for i in range(359-rounded,480+rounded):
    for k in range(20-rounded,140+rounded):
        if (((i-420)**2 + (k-80)**2)<((60+rounded)**2)):
```

```python
            map[k][i] = (0,0,0)
for i in range(359,480):
    for k in range(20,140):
        if (((i-420)**2 + (k-80)**2)<(60**2)):
            map[k][i] = (255,0,0)

# plt.matshow(map)
# plt.show()

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

## 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, linVel = 0, angVel = 0):
        # 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
        # Node as a consequence of applied 'Linear Velocity'
        self.linVel = linVel
        # Node as a consequence of applied 'Angular Velocity'
        self.angVel = angVel

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

## 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], 360), dtype=Node)
for row in range(nodes.shape[0]):
    for col in range(nodes.shape[1]):
        for angle in range(360):
            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 = []
    linear = []
    angular = []
    while True:
        track.append((y,x,l))
        linear.append(nodes[y][x][l].linVel)
        angular.append(nodes[y][x][l].angVel)
        if nodes[y][x][l].parent == None:
            track.reverse()
            linear.reverse()
            angular.reverse()
            break
        y,x,l = nodes[y][x][l].parent
    print("Path created!")
    return track, linear, angular

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

## Get 'Source' and 'Goal' node and check if it's reachable ##
while True:
    print("Node is a point (X,Y) in cartesian plane for XЄ[0,600] and YЄ[0,200]")
    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: ")))
    # x1, y1, x2, y2 = (50, 100, 60, 100)
    print("Orientation of nodes (in degrees) is the direction of mobile robot for thetaЄ[0,360)")
    a1 = int(float(input("Orientation of Source Node (in degrees): ")))
    # a1 = 0
    # Convert a1 to the corresponding layer number [0,359] in the 3D array of 'nodes'
    if a1 in (0, 360):
        l1 = a1
    else:
        print("The angle entered is invalid!")
        break

    # Check if the given coordinates are in the 'Free Space'
    if nodes[200-y1][x1][l1].free and nodes[200-y2][x2][0].free:
        print("Executing path planning for the given coordinates........!!")
        y1 = 200-y1
        y2 = 200-y2
        break
    else:
        print("The given coordinates are not reachable. Try again with different coordinates")
```

```
#-----------------------------------------------------------------------------------------------------------------------------------------
---#

radius = 5

## 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*_phase2.mp4', cv.VideoWriter_fourcc(*'mp4v'), 60, (600,200))
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()

# 'deg' to 'rad' conversion
pi2 = 2*np.pi
deg = np.pi/180

# time 't' (in seconds) such that the change in orientation can be atleast 15 degrees
t = (30*deg) / ((WheelRadius / WheelDistance) * (Min*pi2/60))
# t = 0.1

# Minimum value of cost to go 'c2g_min'
c2g_min = None

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 (threshold)
    if ((y2-y1)**2 + (x2-x1)**2) <= ((threshold)**2):
        print("Path Planning Successfull!!!")
        # Stop the robot at the final node (velocities = 0)
        nodes[y1][x1][l1].linVel = 0
        nodes[y1][x1][l1].angVel = 0
        # Call 'Back-Tracking' function
        Path, Linear_Vel, Angular_Vel = backTrack(x1,y1,l1)
        print("PATH")
        print(Path)
        # for i in Path:
        #     print(i)
        print("LINEAR VELOCITY")
        print(Linear_Vel)
        # for i in Linear_Vel:
        #     print(i)
        print("ANGULAR VELOCITY")
        print(Angular_Vel)
        # for i in Angular_Vel:
        #     print(i)
        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 actionSet: [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2,RPM2],[RPM1,RPM2],
[RPM2,RPM1]]
        # Get neighbouring nodes to the current 'open' node and add it to the Heap Queue 'open_nodes'

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

        # Iterate over 'actions' list
        for action in actionSet:
            # Angular velocity of left and right wheels (wl, wr) respectively (in rad/sec)
            wl = (action[0]*pi2) / 60
            wr = (action[1]*pi2) / 60
```

```
            # Change in of robot orientation 'theta' (in radians), corresponding to 'action'
            theta = (WheelRadius / WheelDistance) * (wl - wr) * t
            # 'theta' (in degrees)
            theta_deg = round(theta / deg)
            phi = theta_deg+l1
            if phi < 0:
                phi = 360 + phi
            elif phi >= 360:
                phi = 360 - phi

            # Distance traveled along X and Y axis (in cm), corresponding to 'action'
            dy = ((WheelRadius/2) * (wl + wr) * np.sin(phi*deg) * t)
            dx = ((WheelRadius/2) * (wl + wr) * np.cos(phi*deg) * t)
            y = round(y1 + dy)
            x = round(x1 + dx)
            l = phi

            # Ignore the nodes if there is negligible change in position due to change in orientation
            # if l1 == 0 or l1 == 180:
            #     if y == y1 and theta_deg != 0:
            #         continue
            # elif l1 == 90 and l1 == 270:
            #     if x == x1 and theta_deg != 0:
            #         continue
            # else:
            #     if theta_deg == 0:
            #         if y == y1 or x == x1:
            #             continue

            # print(f"({y}, {x}, {l}), dx: {dx}, dy: {dy}, theta: {theta}")

            # If the new node exceeds from the map
            if x >= 600 or y >= 200:
                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 'action'
                c2c = math.sqrt((y1-y)**2 + (x1-x)**2)
                # Cost to Go 'c2g'
                c2g = (math.sqrt((y2-y)**2 + (x2-x)**2))*w
                if c2g_min == None or c2g < c2g_min:
                    c2g_min = c2g

                # 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].cog = c2g
                    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)
                    # Make a note of Linear Velocities(m/s) and Angular Velocities(rad/s) as a consequence of which the new node has
arrised
                    nodes[y][x][l].linVel = math.sqrt((dx/t)**2 + (dy/t)**2) / 100
                    nodes[y][x][l].angVel = -1 * theta / t
                    # 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)
                    # Make a note of Linear Velocities(m/s) and Angular Velocities(rad/s) as a consequence of which the new node has
arrised
                    nodes[y][x][l].linVel = math.sqrt((dx/t)**2 + (dy/t)**2) / 100
                    nodes[y][x][l].angVel = -1 * theta / t
                    # 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]

            # print("c2g_min: ", c2g_min)
            # print("node.c2g: ", nodes[y][x][l].cog)
            # # Proceed if the node does not have minimum cost to go, pop next node
            # if (round((nodes[y][x][l].cog) * (10**10))) != round(c2g_min * 10**10):
            #     continue
            # print("Entered!!!!!!")
            # 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

        # Assign the Linear Velocity and Angular Velocity of current node to parent node
        nodes[y1][x1][l1].linVel = nodes[y][x][l].linVel
        nodes[y1][x1][l1].angVel = nodes[y][x][l].angVel
        # Update x1 and y1 for next iteration
        y1 = y
        x1 = x
```

```
        l1 = l
        # print(priority, node)

# 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("time(t) = ", t)
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()

#===================================================================================================================================#







#Part02
#!/usr/bin/env python3


import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import sys
import termios
import time


### 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 matplotlib.pyplot as plt

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

## Initialise ##

WheelRadius = 3.3  #33mm
RobotRadius = 22.0 # 220 mm
WheelDistance = 28.7 #287mm

while True:
    RPM1 = int(float(input("Input RPM1: ")))
    RPM2 = int(float(input("Input RPM2: ")))
    # RPM1 = 50
    # RPM2 = 100
    if RPM1 <= 0 or RPM2 <= 0:
        print("Enter a positive non-zero value.")
    if RPM1 < RPM2:
        Min = min(RPM1, RPM2, RPM2-RPM1)
        break
    elif RPM1 > RPM2:
        Min = min(RPM1, RPM2, RPM1-RPM2)
        break
    else:
```

```
        print("Enter different RPM values for most effective execution.")

actionSet = [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2,RPM2],[RPM1,RPM2],[RPM2,RPM1]]

## Get input for clearance (units) from the obstacle ##
clear = int(float(input("Clearance from obstacles and walls (in cm): ")))
# clear = 1

# w = int(float(input("Heuristic weightage (Enter 1 for default A* execution): ")))
w = 1
# 'threshold' within which the goal node must be reached = 3 units
threshold = 3

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

## Define Map ##

clearance = clear + RobotRadius
rounded = round(clearance)

map = np.ones((200, 600, 3), dtype='uint8')*255
#Wall Barriers
for i in range(0,600):
    for k in range(0,rounded):
        map[k][i] = (0,0,0)
for i in range(0,600):
    for k in range(200-rounded,200):
        map[k][i] = (0,0,0)
for i in range(0,rounded):
    for k in range(0,200):
        map[k][i] = (0,0,0)
for i in range(600-rounded,600):
    for k in range(0,200):
        map[k][i] = (0,0,0)

#Left Most Rectangle Object
# Outer Black Rectangle
for i in range(149-rounded,175+rounded):
    for k in range(0,100+rounded):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(149,175):
    for k in range(0,100):
        map[k][i] = (255,0,0)
#Right Most Rectangle Object
# Outer Black Rectangle
for i in range(249-rounded,275+rounded):
    for k in range(100-rounded,200):
        map[k][i] = (0,0,0)
#Inner Blue Rectangle
for i in range(249,275):
    for k in range(100,200):
        map[k][i] = (255,0,0)

#Circle centered at 420,80, r of 60

#Inner Blue Rectangle
for i in range(359-rounded,480+rounded):
    for k in range(20-rounded,140+rounded):
        if (((i-420)**2 + (k-80)**2)<((60+rounded)**2)):
            map[k][i] = (0,0,0)
for i in range(359,480):
    for k in range(20,140):
        if (((i-420)**2 + (k-80)**2)<(60**2)):
            map[k][i] = (255,0,0)

# plt.matshow(map)
# plt.show()

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

## 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, linVel = 0, angVel = 0):
        # 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
        # Node as a consequence of applied 'Linear Velocity'
        self.linVel = linVel
        # Node as a consequence of applied 'Angular Velocity'
        self.angVel = angVel

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

## 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], 360), dtype=Node)
```

```python
for row in range(nodes.shape[0]):
    for col in range(nodes.shape[1]):
        for angle in range(360):
            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 = []
    linear = []
    angular = []
    while True:
        track.append((y,x,l))
        linear.append(nodes[y][x][l].linVel)
        angular.append(nodes[y][x][l].angVel)
        if nodes[y][x][l].parent == None:
            track.reverse()
            linear.reverse()
            angular.reverse()
            break
        y,x,l = nodes[y][x][l].parent
    print("Path created!")
    return track, linear, angular

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

## Get 'Source' and 'Goal' node and check if it's reachable ##
while True:
    print("Node is a point (X,Y) in cartesian plane for X∈[0,600] and Y∈[0,200]")
    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: ")))
    # x1, y1, x2, y2 = (50, 100, 60, 100)
    print("Orientation of nodes (in degrees) is the direction of mobile robot for theta∈[0,360)")
    a1 = int(float(input("Orientation of Source Node (in degrees): ")))
    # a1 = 0
    # Convert a1 to the corresponding layer number [0,359] in the 3D array of 'nodes'
    if a1 in (0, 360):
        l1 = a1
    else:
        print("The angle entered is invalid!")
        break

    # Check if the given coordinates are in the 'Free Space'
    if nodes[200-y1][x1][l1].free and nodes[200-y2][x2][0].free:
        print("Executing path planning for the given coordinates........!!")
        y1 = 200-y1
        y2 = 200-y2
        break
    else:
        print("The given coordinates are not reachable. Try again with different coordinates")

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

radius = 5

## 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*_phase2.mp4', cv.VideoWriter_fourcc(*'mp4v'), 60, (600,200))
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()

# 'deg' to 'rad' conversion
pi2 = 2*np.pi
deg = np.pi/180

# time 't' (in seconds) such that the change in orientation can be atleast 2 degrees
t = (1*deg) / ((WheelRadius / WheelDistance) * (Min*pi2/60))
t = 

# Minimum value of cost to go 'c2g_min'
```

```python
c2g_min = None

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 (threshold)
    if ((y2-y1)**2 + (x2-x1)**2) <= ((threshold)**2):
        print("Path Planning Successfull!!!")
        # Stop the robot at the final node (velocities = 0)
        nodes[y1][x1][l1].linVel = 0
        nodes[y1][x1][l1].angVel = 0
        # Call 'Back-Tracking' function
        Path, Linear_Vel, Angular_Vel = backTrack(x1,y1,l1)
        print("PATH")
        print(Path)
        # for i in Path:
        #     print(i)
        print("LINEAR VELOCITY")
        print(Linear_Vel)
        # for i in Linear_Vel:
        #     print(i)
        print("ANGULAR VELOCITY")
        print(Angular_Vel)
        # for i in Angular_Vel:
        #     print(i)
        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 actionSet: [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2,RPM2],[RPM1,RPM2],
[RPM2,RPM1]]
        # Get neighbouring nodes to the current 'open' node and add it to the Heap Queue 'open_nodes'

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

        # Iterate over 'actions' list
        for action in actionSet:
            # Angular velocity of left and right wheels (wl, wr) respectively (in rad/sec)
            wl = (action[0]*pi2) / 60
            wr = (action[1]*pi2) / 60

            # Change in of robot orientation 'theta' (in radians), corresponding to 'action'
            theta = (WheelRadius / WheelDistance) * (wl - wr) * t
            # 'theta' (in degrees)
            theta_deg = round(theta / deg)
            phi = theta_deg+l1
            if phi < 0:
                phi = 360 + phi
            elif phi >= 360:
                phi = 360 - phi

            # Distance traveled along X and Y axis (in cm), corresponding to 'action'
            dy = ((WheelRadius/2) * (wl + wr) * np.sin(phi*deg) * t)
            dx = ((WheelRadius/2) * (wl + wr) * np.cos(phi*deg) * t)
            y = round(y1 + dy)
            x = round(x1 + dx)
            l = phi

            # Ignore the nodes if there is negligible change in position due to change in orientation
            # if l1 == 0 or l1 == 180:
            #     if y == y1 and theta_deg != 0:
            #         continue
            # elif l1 == 90 and l1 == 270:
            #     if x == x1 and theta_deg != 0:
            #         continue
            # else:
            #     if theta_deg == 0:
            #         if y == y1 or x == x1:
            #             continue

            print(f"({y}, {x}, {l}), dx: {dx}, dy: {dy}, theta: {theta}")

            # If the new node exceeds from the map
            if x >= 600 or y >= 200:
                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 'action'
                c2c = math.sqrt((y1-y)**2 + (x1-x)**2)
                # Cost to Go 'c2g'
                c2g = (math.sqrt((y2-y)**2 + (x2-x)**2))*w
                if c2g_min == None or c2g < c2g_min:
                    c2g_min = c2g

                # 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].cog = c2g
                    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)
                    # Make a note of Linear Velocities(m/s) and Angular Velocities(rad/s) as a consequence of which the new node has
arrised
                    nodes[y][x][l].linVel = math.sqrt((dx/t)**2 + (dy/t)**2) / 100
                    nodes[y][x][l].angVel = -1 * theta / t
                    # 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)
                    # Make a note of Linear Velocities(m/s) and Angular Velocities(rad/s) as a consequence of which the new node has
arrised
                    nodes[y][x][l].linVel = math.sqrt((dx/t)**2 + (dy/t)**2) / 100
                    nodes[y][x][l].angVel = -1 * theta / t
                    # 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]

            # print("c2g_min: ", c2g_min)
            # print("node.c2g: ", nodes[y][x][l].cog)
            # # Proceed if the node does not have minimum cost to go, pop next node
            # if (round((nodes[y][x][l].cog) * (10**10))) != round(c2g_min * 10**10):
            #     continue
            # print("Entered!!!!!!")
            # 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

        # Assign the Linear Velocity and Angular Velocity of current node to parent node
        nodes[y1][x1][l1].linVel = nodes[y][x][l].linVel
        nodes[y1][x1][l1].angVel = nodes[y][x][l].angVel
        # Update x1 and y1 for next iteration
        y1 = y
        x1 = x
        l1 = l
        print(priority, node)

# 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("time(t) = ", t)
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)
```

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

out.release()

#==================================================================================================================#










lin_vel = Linear_Vel
ang_vel = Angular_Vel
t = t


# lin_vel = [0.03455751918948772, 0.01727875959474386, 0.01727875959474386, 0.01727875959474386, 0.01727875959474386,
0.01727875959474386, 0.01727875959474386, 0.01727875959474386, 0.01727875959474386, 0.01727875959474386, 0.05183627878423159,
0.01727875959474386, 0.01727875959474386, 0.06911503837897544, 0.06911503837897544, 0.01727875959474386, 0.03455751918948772,
0.03455751918948772, 0.03455751918948772, 0.03455751918948772, 0.05183627878423159, 0.01727875959474386, 0.01727875959474386,
0.01727875959474386, 0.03455751918948772, 0.01727875959474386, 0.03455751918948772, 0.01727875959474386, 0.03455751918948772,
0.03455751918948772, 0.01727875959474386, 0.01727875959474386, 0.01727875959474386, 0.03455751918948772, 0.01727875959474386,
0.06911503837897544, 0.06911503837897544, 0.01727875959474386, 0.03455751918948772, 0.01727875959474386, 0.01727875959474386, 0]
# ang_vel = [-0.0, -0.12040947452783178, 0.12040947452783178, -0.12040947452783178, 0.12040947452783178, -0.12040947452783178,
0.12040947452783178, -0.12040947452783178, 0.12040947452783178, 0.12040947452783178, 0.12040947452783178, 0.12040947452783178, -
0.12040947452783178, -0.0, -0.0, -0.12040947452783178, -0.24081894905566356, 0.24081894905566356, -0.0, -0.24081894905566356,
0.12040947452783178, 0.12040947452783178, 0.12040947452783178, 0.12040947452783178, 0.24081894905566356, 0.12040947452783178, -
0.24081894905566356, 0.12040947452783178, 0.24081894905566356, -0.24081894905566356, 0.12040947452783178, 0.12040947452783178, -
0.12040947452783178, -0.24081894905566356, -0.12040947452783178, -0.0, -0.0, -0.12040947452783178, 0.24081894905566356,
0.12040947452783178, -0.12040947452783178, 0]

# t =  8.917623043060303

input('Enter to Start Gazebo Simulation...')


class move(Node):

    def __init__(self,lin_vel,ang_vel,t):
        #define init method
        super().__init__('keyboard_control_node')
        #define linear velocity, angular velocity and t
        self.lin_vel = lin_vel
        self.ang_vel = ang_vel
        self.t = t
        #create publisher node
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.settings = termios.tcgetattr(sys.stdin)


    def publish_moves(self):
        print('Executing Path Planner...')
        #establish node
        velocity_message = Twist()
        linear_vel=0.0
        angular_vel=0.0
        #publish the desired speeds
        velocity_message.linear.x = linear_vel
        velocity_message.angular.z = angular_vel
        self.cmd_vel_pub.publish(velocity_message)

        count = 0
        for i in range(len(self.lin_vel)-1):
            # Publish the twist message
            lin_vel = (float(self.lin_vel[i]))
            ang_vel = (float(self.ang_vel[i]))
            # print('lin vel: ', lin_vel)
            velocity_message.linear.x = lin_vel
            velocity_message.angular.z = ang_vel
            self.cmd_vel_pub.publish(velocity_message)
            time.sleep(self.t)
            print(count)
            count += 1


        velocity_message.linear.x = linear_vel
        velocity_message.angular.z = angular_vel
        self.cmd_vel_pub.publish(velocity_message)

        print('Path Complete!')

def main(args=None):
    rclpy.init(args=args)
    turtlebot = move(lin_vel,ang_vel,t)
    turtlebot.publish_moves()
    turtlebot.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```