The objective of this section to make the code more robust to more robot and less manual config.

Suggestion

First, we remove the default configuration.

First, we going to introduce about numpy random,

# First task:

We also need to import numpy as np

And to make things consistent for all, we set the

np.random.seed(0)

next, we can start

Replace the code

def draw(self,canvas):  
  
 points = [ (self.x + 30\*math.sin(self.theta)) - 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y - 30\*math.cos(self.theta)) - 30\*math.cos((math.pi/2.0)-self.theta), \  
 (self.x - 30\*math.sin(self.theta)) - 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y + 30\*math.cos(self.theta)) - 30\*math.cos((math.pi/2.0)-self.theta), \  
 (self.x - 30\*math.sin(self.theta)) + 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y + 30\*math.cos(self.theta)) + 30\*math.cos((math.pi/2.0)-self.theta), \  
 (self.x + 30\*math.sin(self.theta)) + 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y - 30\*math.cos(self.theta)) + 30\*math.cos((math.pi/2.0)-self.theta) \  
 ]  
 canvas.create\_polygon(points, fill="blue", tags=self.name)  
  
 self.sensorPositions = [ (self.x + 20\*math.sin(self.theta)) + 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y - 20\*math.cos(self.theta)) + 30\*math.cos((math.pi/2.0)-self.theta), \  
 (self.x - 20\*math.sin(self.theta)) + 30\*math.sin((math.pi/2.0)-self.theta), \  
 (self.y + 20\*math.cos(self.theta)) + 30\*math.cos((math.pi/2.0)-self.theta) \  
 ]  
  
 centre1PosX = self.x  
 centre1PosY = self.y  
 canvas.create\_oval(centre1PosX-15,centre1PosY-15, \  
 centre1PosX+15,centre1PosY+15, \  
 fill="gold",tags=self.name)  
 batteryText = canvas.create\_text(self.x,self.y,text=str(self.battery),tags=self.name)  
  
 wheel1PosX = self.x - 30\*math.sin(self.theta)  
 wheel1PosY = self.y + 30\*math.cos(self.theta)  
 canvas.create\_oval(wheel1PosX-3,wheel1PosY-3, \  
 wheel1PosX+3,wheel1PosY+3, \  
 fill="red",tags=self.name)  
  
 wheel2PosX = self.x + 30\*math.sin(self.theta)  
 wheel2PosY = self.y - 30\*math.cos(self.theta)  
 canvas.create\_oval(wheel2PosX-3,wheel2PosY-3, \  
 wheel2PosX+3,wheel2PosY+3, \  
 fill="green",tags=self.name)  
  
 sensor1PosX = self.sensorPositions[0]  
 sensor1PosY = self.sensorPositions[1]  
 sensor2PosX = self.sensorPositions[2]  
 sensor2PosY = self.sensorPositions[3]  
 canvas.create\_oval(sensor1PosX-3,sensor1PosY-3, \  
 sensor1PosX+3,sensor1PosY+3, \  
 fill="yellow",tags=self.name)  
 canvas.create\_oval(sensor2PosX-3,sensor2PosY-3, \  
 sensor2PosX+3,sensor2PosY+3, \  
 fill="yellow",tags=self.name)

# Task 2

Modify the register, by removing the needs fors

In the positional of register,

We remove

def register(canvas,robot\_configurations,colors):

to

def register(canvas):

Remember, at this stage, we aim to make thing can be expanded for number of robot, and we want to avoid manually assign the robot configuration.

Similarly, the changes we make to the method draw, so, we should modify the function caller.

From

for i, (config,clrx) in enumerate(zip(robot\_configurations,colors)):  
 robot\_name='c'  
 bot=Bot(robot\_name)  
 bot.draw(i, canvas, config, condition, colors, my\_th)

to

for i in range(0,noOfBots):  
 bot = Bot("Bot"+str(i))  
 bot.draw(canvas)

if you notice, no we can simply define the noOfBots. In fact, this is something similar to how we use to generate the number of dirts.

If you manage to made the changes, u should get the following graph:  
  
A screenshot of a computer

Description automatically generated

# Task 3

Here, we need to create the transfer function, for simplicity, lets paste all the code to dynamic\_component

import random  
import math  
import tkinter as tk  
import numpy as np  
from passive\_component import Dirt,Counter  
from robot\_helper import initialise, buttonClicked, WiFiHub, Charger  
class try\_move:  
 def \_\_init\_\_(self,robot\_obj):  
 hhh=22  
 self.currentlyTurning=robot\_obj.currentlyTurning  
 self.ll=robot\_obj.ll  
 self.moving=robot\_obj.moving  
 self.name=robot\_obj.name  
 self.sensorPositions=robot\_obj.sensorPositions  
 self.theta=robot\_obj.theta  
 self.turning=robot\_obj.turning  
 self.vl=robot\_obj.vl  
 self.vr=robot\_obj.vr  
 self.x=robot\_obj.x  
 self.y=robot\_obj.y  
 self.battery=robot\_obj.battery  
 self.draw=robot\_obj.draw  
 self.canvas = robot\_obj.canvas  
 c=1  
 *# cf. Dudek and Jenkin, Computational Principles of Mobile Robotics  
 # cf. Dudek and Jenkin, Computational Principles of Mobile Robotics* def move(self,canvas,registryPassives,dt):  
 if self.battery>0:  
 self.battery -= 1  
 if self.battery==0:  
 self.vl = 0  
 self.vr = 0  
 for rr in registryPassives:  
 if isinstance(rr,Charger) and self.distanceTo(rr)<80:  
 self.battery += 10  
  
 if self.vl==self.vr:  
 R = 0  
 else:  
 R = (self.ll/2.0)\*((self.vr+self.vl)/(self.vl-self.vr))  
  
 omega = (self.vl-self.vr)/self.ll  
 ICCx = self.x-R\*math.sin(self.theta) *#instantaneous centre of curvature* ICCy = self.y+R\*math.cos(self.theta)  
 m = np.matrix( [ [math.cos(omega\*dt), -math.sin(omega\*dt), 0], \  
 [math.sin(omega\*dt), math.cos(omega\*dt), 0], \  
 [0,0,1] ] )  
 v1 = np.matrix([[self.x-ICCx],[self.y-ICCy],[self.theta]])  
 v2 = np.matrix([[ICCx],[ICCy],[omega\*dt]])  
 newv = np.add(np.dot(m,v1),v2)  
 newX = newv.item(0)  
 newY = newv.item(1)  
 newTheta = newv.item(2)  
 newTheta = newTheta%(2.0\*math.pi) *#make sure angle doesn't go outside [0.0,2\*pi)* self.x = newX  
 self.y = newY  
 self.theta = newTheta  
 if self.vl==self.vr: *# straight line movement* self.x += self.vr\*math.cos(self.theta) *#vr wlog* self.y += self.vr\*math.sin(self.theta)  
 if self.x<0.0:  
 self.x=999.0  
 if self.x>1000.0:  
 self.x = 0.0  
 if self.y<0.0:  
 self.y=999.0  
 if self.y>1000.0:  
 self.y = 0.0  
 *#self.updateMap()* canvas.delete(self.name)  
 self.draw(canvas)  
  
 def pickUpAndPutDown(self,xp,yp):  
 self.x = xp  
 self.y = yp  
 self.canvas.delete(self.name)  
 self.draw(self.canvas)  
  
  
  
  
 def senseCharger(self, registryPassives):  
 lightL = 0.0  
 lightR = 0.0  
 for pp in registryPassives:  
 if isinstance(pp,Charger):  
 lx,ly = pp.getLocation()  
 distanceL = math.sqrt( (lx-self.sensorPositions[0])\*(lx-self.sensorPositions[0]) + \  
 (ly-self.sensorPositions[1])\*(ly-self.sensorPositions[1]) )  
 distanceR = math.sqrt( (lx-self.sensorPositions[2])\*(lx-self.sensorPositions[2]) + \  
 (ly-self.sensorPositions[3])\*(ly-self.sensorPositions[3]) )  
 lightL += 200000/(distanceL\*distanceL)  
 lightR += 200000/(distanceR\*distanceR)  
 return lightL, lightR  
  
 def senseHubs(self, registryPassives):  
 signal = []  
 for pp in registryPassives:  
 if isinstance(pp,WiFiHub):  
 lx,ly = pp.getLocation()  
 distanceL = math.sqrt( (lx-self.sensorPositions[0])\*(lx-self.sensorPositions[0]) + \  
 (ly-self.sensorPositions[1])\*(ly-self.sensorPositions[1]) )  
 distanceR = math.sqrt( (lx-self.sensorPositions[2])\*(lx-self.sensorPositions[2]) + \  
 (ly-self.sensorPositions[3])\*(ly-self.sensorPositions[3]) )  
 signal.append(200000/(distanceL\*distanceL))  
 signal.append(200000/(distanceR\*distanceR))  
 return signal  
  
 def distanceTo(self,obj):  
 xx,yy = obj.getLocation()  
 return math.sqrt( math.pow(self.x-xx,2) + math.pow(self.y-yy,2) )  
  
 def collectDirt(self, canvas, registryPassives, count):  
 toDelete = []  
 for idx,rr in enumerate(registryPassives):  
 if isinstance(rr,Dirt):  
 if self.distanceTo(rr)<30:  
 canvas.delete(rr.name)  
 toDelete.append(idx)  
 count.itemCollected(canvas)  
 for ii in sorted(toDelete,reverse=True):  
 del registryPassives[ii]  
 return registryPassives  
  
 def transferFunction(self,chargerL,chargerR):  
 *# wandering behaviour* if self.currentlyTurning==True:  
 self.vl = -2.0  
 self.vr = 2.0  
 self.turning -= 1  
 else:  
 self.vl = 5.0  
 self.vr = 5.0  
 self.moving -= 1  
 if self.moving==0 and not self.currentlyTurning:  
 self.turning = random.randrange(20,40)  
 self.currentlyTurning = True  
 if self.turning==0 and self.currentlyTurning:  
 self.moving = random.randrange(50,100)  
 self.currentlyTurning = False  
 *#battery - these are later so they have priority* if self.battery<600:  
 *# if chargerR>chargerL:  
 # self.vl = 2.0  
 # self.vr = -2.0  
 # elif chargerR<chargerL:  
 # self.vl = -2.0  
 # self.vr = 2.0  
 # if abs(chargerR-chargerL)<chargerL\*0.1: #approximately the same  
 # self.vl = 5.0  
 # self.vr = 5.0* self.vl = 5\*math.sqrt(chargerR)  
 self.vr = 5\*math.sqrt(chargerL)  
 if chargerL+chargerR>200 and self.battery<1000:  
 self.vl = 0.0  
 self.vr = 0.0

and to call the function, you can import it simply by calling

from dynamic\_component import try\_move