RMI - Assignment 1

RMI - TP1 04/11/2022

Pedro Lima - 97860
Nuno Cunha - 98124

TABLE OF CONTENTS

1st Challenge

The first challenge was the Control challenge, in which the goal was to make the robot follow a path using only the line sensor

1st Challenge

The agent for the 1st challenge is a pure reactive one.
It uses the data received by the line sensor and uses it to choose a condition in the code.

```
if self.measures.lineSensor[6]=='1' and self.measures.lineSensor[5]=='1':
    self.driveMotors(0.15,-0.15)
    #print("right")

elif self.measures.lineSensor[0]=='1' and self.measures.lineSensor[1]=='1':
    self.driveMotors(-0.15,0.15)
    #print("left")

elif self.measures.lineSensor[1]=='1' and self.measures.lineSensor[2]=='1':
    self.driveMotors(0.0,0.1)
    #print("adjust left")

elif self.measures.lineSensor[4]=='1' and self.measures.lineSensor[5]=='1':
    self.driveMotors(0.1,0.0)
    #print("adjust right")

elif self.measures.lineSensor[6]=='1':
    self.driveMotors(0.15,-0.15)
    #print("right, resort")

elif self.measures.lineSensor[0]=='1':
    self.driveMotors(-0.15,0.15)
    #print("left, resort ")
else:
    self.driveMotors(0.15,0.15)
```

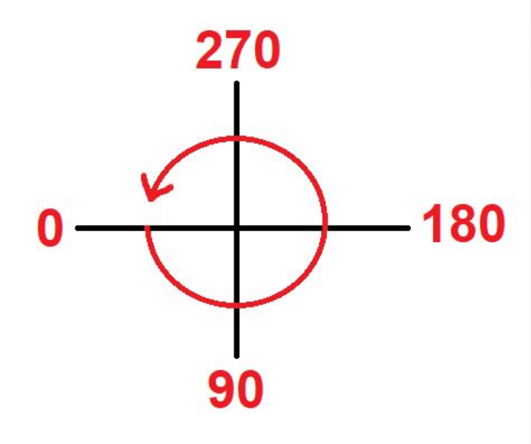
2nd Challenge

The second challenge is the Mapping challenge, which the objective of this challenge is to explore an unknown maze in order to extract its map using the line sensor, GPS and compass

2nd Challenge

```
compass=self.measures.compass+180
```

To facilitate our use of the compass, we always add 180 to its value. This removes the negative values for the compass.
For this reason our compass changed to the circle in the figure.



2nd Challenge

To define each intersection we created a class named vertice. It contains the x and y coordinates of the vertice as well as the dictionary called visitados. The adjacentes dictionary was not used.

These vertices were saved in an array in the Myrob class

```
class Vertice():
    def __init__(self, x, y):
        self.x = x
        self.y = y
        #exemplo {0: v1, 90: v2, 180: v3, 270: v4} se fosse um cruzamento
        self.adjacentes = {}
        #exemplo {0: false, 90: false, 180: true, 270: false} se fosse um cruzamento
        self.visitados = {0: None, 90: None, 180: None, 270: None}
```

```
# lista de vertices detectados
self.vertices=[]
```

2nd Challenge

In each call of the wander function we define 3 variables (right,left and front) as None. These variables are used by the agent to decide which direction to go.

```
# None = nao pode ir, 0 = pode ir mas já foi , 1 = pode ir mas ainda nao foi
right = None
left = None
front = None

if self.can_turn('left'):

    if self.decide('left'):
        left = 1
    else:
        left = 0

if self.can_turn('right'):

    if self.decide('right'):
        right = 1
    else:
        right = 0

if self.can_turn('front') and (left == 0 or right == 0):

    if self.decide('front'):
        front = 1
    else:
        front = 0
```

2nd Challenge

After having the values for the 3 variables we use them to decide if we turn to that given direction or not.

After defining the variables needed for the turning code, we check_intersections function is called.

```
if (left == 1 ) or (left==0 and right==None and front==None) or (ran!=None and ran <= 0.6) : ##cruzamento
    self.right=0
    self.counter+=1
    #guardar a ultima direçao do robô
    self.direction=compass
    self.driveMotors(-0.15,0.15)
    #print("cruzamento")

    self.check_intersections('left')
```

2nd Challenge

After the counter is set to 1, we make a turn to the right or left depending in the self.right value.

The last if is used to detect if we could go to the front using the line sensor.

```
if self.counter>0:


    if self.right==1:
        if self.direction>=80 and self.direction<=100 and compass<350:
            self.driveMotors(0.1,0.0)
            self.Turn_to_0=0
            #print("turn right to 0")
        elif (self.direction>=350 or self.direction<=10) and compass>270:
            self.driveMotors(0.1,0.0)
            self.Turn_to_0=0

        elif compass> self.direction-90 and self.Turn_to_0:
            self.driveMotors(0.1,0.0)
            #print("TURN LEFT")
        else:
            self.counter=0
            self.Turn_to_0=1
            self.number_sides_detected=0
            self.right=None

        if self.measures.lineSensor[0]=='1' and self.measures.lineSensor[1]=='1'
            #print('frente e direita')
            if self.check_false_front("fd"):
                self.check_intersections('front')
```

2nd Challenge

For the case when the robot doesn’t detect the line, we make it turn 180 degrees.
For this we update the turn_180 variable to 1 and counter to activate the turning code.

The check intersections function is also called to create a vertex in that spot.

```
elif self.measures.lineSensor==['0','0','0','0','0','0','0']:
    self.turn_180=1
    self.counter+= 1
    self.direction=compass

    value = 90 * round(self.direction / 90)
    if value == 360:
        value = 0
    if value == 0 or value == 90:
        self.driveMotors(-0.15,0.15)
    else:
        self.driveMotors(0.15,-0.15)
    self.check_intersections('back')
```

2nd Challenge

After the counter is set to 1, and we need to turn 180 degrees this code is triggered.

```
if self.turn_180 == 1:
    value = 90 * round(self.direction / 90)
    if value == 360:
        value = 0

    # print(value)
    # print(compass)

    if value==180:
        if not (compass>=350 or compass<=10 ):
            self.driveMotors(0.15,-0.15)
        else:
            self.turn_180=0
            self.counter=0
```

2nd Challenge

```
self.adjacent_dict={
    # (846,398): set{(856,400,270, 4)},
    # }
}
```

```
def check_intersections(self,side):

    v_check = [v for v in self.vertices if v.x == self.round_positions(self.measures.x) and v.y == self.round_positions(self.measures.y)]
    #verificar se o vertice já existe no nosso array ou se é um vertice novo

    value = 90 * round(self.direction / 90)
    if value == 360:
        value = 0

    if self.last_vertice != None:

        cost = 0
        if self.last_vertice[0] == self.round_positions(self.measures.x):
            cost = abs(self.last_vertice[1] - self.round_positions(self.measures.y))

        if self.last_vertice[1] == self.round_positions(self.measures.y):
            cost = abs(self.last_vertice[0] - self.round_positions(self.measures.x))
```

The check_intersections function serves to update our adjacency dictionary

First we calculate the cost of going from the last vertice to the current one.

2nd Challenge

If the cost is not 0 then we update the adjacency dictionary with the appropriate values.

```
if cost != 0:

    if self.last_vertice not in self.adjacent_dict.keys():
        self.adjacent_dict[self.last_vertice] = set()

    if self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])] == set():
        self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])].add((self.round_positions(self.measures.x),self.round_positions(self.measures.y),value,cost))


    temp = 0
    for s in self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])].copy():
        if s[2] == value and s[3] > cost:
            self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])].remove(s)
            self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])].add((self.round_positions(self.measures.x),self.round_positions(self.measures.y),value,cost))
            temp = 1
            break
        elif s[2] == value and s[3] < cost:
            temp = 1
            break

    if temp == 0:
        self.adjacent_dict[(self.last_vertice[0],self.last_vertice[1])].add((self.round_positions(self.measures.x),self.round_positions(self.measures.y),value,cost))
```

2nd Challenge

At the end of the function, we create a new vertex or update the visited list of the current vertex if it already existed in the list.

To update the visited list we call the check_adacentes function.

```
if  v_check == [] and side !=None:

    v = Vertice(self.round_positions(self.measures.x),self.round_positions(self.measures.y))

    v = self.check_adjacentes(v,side)


    self.vertices.append(v)
    #print("new Vertice")
    #print(v.get_visitados())


#no caso do vertice já existir, verificar se o caminho já existe ou se é um caminho novo
elif side !=None:
    v = v_check[0]
    v = self.check_adjacentes(v,side)

    index = next((i for i, item in enumerate(self.vertices) if item.x == v.x and item.y == v.y), None)

    if index!=None:
        self.vertices[index] = v
```

2nd Challenge

The first thing the function does is update the direction from with the robot came.

```
def check_adjacentes(self,v,side):

    value = 90 * round(self.direction / 90)
    if value == 360:
        value = 0

    if value>=180:
        v.add_visitado(value-180, True)
    else:
        v.add_visitado(value+180, True)
```

2nd Challenge

Then we update the visited list depending in the side we are turning and the line sensor.

```
if self.measures.lineSensor[6]=='1' and self.measures.lineSensor[5]=='1':

    if self.direction>=80 and self.direction<=100 and v.visitados[0] != True:

        #v.add_adjacente(0, v1)
        v.add_visitado(0, True) if side == 'right' else v.add_visitado(0, False)
```

2nd Challenge

At 50 ticks of the simulator clock we create a matrix using the adjacency dict to facilitate the writing to the file.

```python
if ( (int(self.simTime) - self.measures.time) <= 50) :

    matrix = self.createMatrix()
    #print(matrix)

    with open('mapa.txt', 'w') as f:
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == 0:
                    f.write(' ')
                elif matrix[i][j] == 1:
                    f.write('-')
                elif matrix[i][j] == 2:
                    f.write('|')
                elif matrix[i][j] == 3:
                    f.write('I')

            f.write('\n')
```

3th Challenge

The third challenge, the Planning challenge, is the follow up of the 2nd challenge were we need to explore an unknown maze in order to locate a number of target spots and compute the shortest closed path that allows to visit those target spots starting and ending in the starting spot

3th Challenge

Most of the code is from the 2nd challenge, so we will only present the differences.

We made a function that calculate the cost of a given path, it compares the distance between 2 consecutive points until completes the path.

```python
def calculate_cost(self,caminho):


    #print(caminho)

    cost=0
    for i in range(len(caminho)-1):

        cost+=dist(caminho[i], caminho[i+1])

    #print(cost)

    return cost
```

3th Challenge

The function print_path_file() is used to output the file with the shortest path.

To print out the file we need to write the position of the robot every 2 units, because we only have the intersection points we calculated the difference and with the rest of division write the position of the robot every 2 units

```
with open('plan.txt', 'w') as f:
        for i in caminho:

            temp1 = i[0] - initial_pos[0]
            temp2 = i[1] - initial_pos[1]


            if last_temp1 != None and (temp1 - last_temp1 != 2 and temp1 - last_temp1 != -2 and temp1 - last_temp1 != 0):

                dif = temp1 - last_temp1

                for i in range(1,abs(dif//2)):
                    if dif < 0:
                        f.write(str(last_temp1-2*i))
                        f.write(" ")
                        f.write(str(last_temp2))
                        f.write('\n')

                    if dif > 0:
                        f.write(str(last_temp1+2*i))
                        f.write(" ")
                        f.write(str(last_temp2))
                        f.write('\n')
```

3th Challenge

In this section of the code we save the positions of the targets and define it as an especial intersection to be in the adjacency matrix. We also define the initial position of the robot based on the time of the simulator.

```
if self.measures.ground != -1 and self.measures.ground != self.last_ground:

    #print(self.measures.ground)

    self.direction = compass
    self.check_intersections(None)

    self.checkpoints.add( (self.round_positions(self.measures.x), self.round_positions(self.measures.y))  )

    self.last_ground = self.measures.ground



# Definir posição do robo
if self.measures.time<=2:
    self.inicio = ( self.round_positions(self.measures.x), self.round_positions(self.measures.y) )
```

3th Challenge

We trigger this code when we have 50 ticks in the simulator clock.
To calculate the best path we make all the possible combinations of the order of the targets and calculate the associate cost for that path using the dijkstra_algorithm() and the calculate_cost() functions.
Then save the best path and write it to a file and finish the run.

```
d = self.checkpoints
d.remove(self.inicio)
#print(d)

best_caminho = []

for i in permutations(d,len(d)):

    caminho = self.dijkstra_algorithm(self.inicio, i[0])

    if caminho == None:
        continue

    for j in range(len(i)-1):
        temp = self.dijkstra_algorithm(i[j], i[j+1])

        if temp == None:
            continue

        temp.pop(0)
        caminho = caminho + temp
    if temp == None:
        continue

    temp = self.dijkstra_algorithm(i[-1],self.inicio)
    temp.pop(0)
    caminho = caminho + temp

    # calcular custo do caminho
    #print('-----------------------')
    #self.calculate_cost(caminho)
    if best_caminho == [] or ( self.calculate_cost(caminho) < self.calculate_cost(best_caminho) ):
        best_caminho = caminho
```

THANKS

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.

Please keep this slide for attribution.