

Relatório de Trabalho Prático

David Pires nº37272 e Pedro Maria nº39866

Janeiro 2020



**UNIVERSIDADE
BEIRA INTERIOR**

1 Conteúdo

Contents

1	Conteúdo	1
2	Motivação	2
3	Objetivos	2
4	Implementação	3
4.1	Variáveis auxiliares	3
4.1.1	Lista das salas	3
4.1.2	Lista de objetos	3
4.1.3	Lista de pontos de interesse	3
4.1.4	Garfo dos pontos de interesse	4
4.2	Funções auxiliares	5
4.2.1	Calcular a distância entre dois pontos (x1, y1) e (x2, y2) .	5
4.2.2	Registo de um novo ponto de interesse	5
4.2.3	Algoritmo dijkstra	5
4.3	Resposta às perguntas	7
4.3.1	Pergunta 1: How many rooms are not occupied?	7
4.3.2	Pergunta 2: How many suites did you find until now? . .	7
4.3.3	Pergunta 3: Is it more probable to find people in the corridors or inside the rooms?	7
4.3.4	Pergunta 4: If you want to find a computer, to which type of room do you go to?	8
4.3.5	Pergunta 5: What is the number of the closest single room?	8
4.3.6	Pergunta 6: How can you go from the current room to the elevator?	9
4.3.7	Pergunta 7: How many books do you estimate to find in the next 2 minutes?	9
4.3.8	Pergunta 8: What is the probability of finding a table in a room without books but that has at least one chair? . .	10
5	Conclusão	12
5.1	Partição do trabalho	12
6	Bibliografia	12

2 Motivação

A grande motivação deste projeto foi conseguir de certa forma criar um robô que seja mais inteligente que o robô comum, pois estes não são capazes de reconhecer os objetos presentes no meio, retirar informações e fazer conclusões acerca do mesmo. Sendo então o objetivo deste trabalho desenvolver então essas capacidades.

3 Objetivos

O objetivo deste trabalho é a criação de um programa que consiga comandar um robô móvel num determinado ambiente e que este seja capaz não só de reconhecer o ambiente que o rodeia mas também responder a perguntas específicas sobre o mundo em que está inserido tendo em conta a informação que o robô recolheu até ao dado momento.

4 Implementação

4.1 Variáveis auxiliares

Para a conclusão deste trabalho, recorreremos à criação de várias variáveis auxiliares, para o manuseamento das informações obtidas pelo robô e pedidas pelo utilizador do programa.

4.1.1 Lista das salas

```
room_list = [(1,-15.6,-3.0,-3.6,-0.8), (2,-12.0,-1.4,-8.9,4.8),  
             ↪ (3,-10.6,4.8,3.6,7.9), (4,-4.6,-0.8,-0.8,4.8),  
             ↪ (5,-15.6,-1.4,-12.5,3.0), (6,-15.6,3.0,-12.5,7.9),  
             ↪ (7,-15.6,7.9,-10.6,11.1), (8,-10.6,7.9,-5.6,11.1),  
             ↪ (9,-5.6,7.9,-0.5,11.1), (10,-0.5,7.9,3.6,11.1),  
             ↪ (11,-0.8,1.4,3.6,4.8), (12,-0.8,-0.8,3.6,1.4),  
             ↪ (13,-8.9,-0.8,-6.5,4.8), (14,-6.5,-0.8,-4.6,4.8)]
```

Esta é uma lista pré definida no programa que contem as coordenadas de cada sala. Esta lista é composta por tuplos do tipo (roomNumber, x1, y1, x2, y2) em que roomNumber corresponde ao numero da sala, x1 e y1, correspondem às coordenadas do canto inferior esquerdo da sala e x2, y2, corresponde às coordenadas do canto superior direito da sala.

4.1.2 Lista de objetos

```
object_list = []
```

Esta é uma lista, que é incrementada dinamicamente à medida que o robô descobre um novo objeto. Esta lista será composta por tuplos, com a seguinte estrutura: (roomNumber, objectName, objectID), em que roomNumber corresponde ao numero da sala onde foi encontrado o objeto, objectName corresponde ao nome do objecto e objectID corresponde ao seu ID.

4.1.3 Lista de pontos de interesse

```
point_list = []
```

Esta é uma lista incrementada dinamicamente com pontos de interesse, que neste caso se podem traduzir em "portas" (menos nos casos se passagem de um corredor para outro). Esta lista é composta por tuplos, com a seguinte estrutura: (uuid, x, y, roomA, roomB), onde:

- uuid é um ID único aleatoriamente gerado, representante deste ponto.
- x e y são as coordenadas do ponto.
- roomA e roomB são as salas entre o ponto (visto que este é um ponto correspondente à passagem de uma sala para outra).

4.1.4 Garfo dos pontos de interesse

```
from collections import defaultdict

class Graph():
    def __init__(self):
        self.edges = defaultdict(list)
        self.weights = {}

    def add_edge(self, from_node, to_node, weight):
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight

    def getWeight(self, from_node, to_node):
        return self.weights[(from_node, to_node)]
```

```
graph = Graph()
```

Para isto trabalhamos numa classe feita originalmente por Ben Keen (Link na Bibliografia). Este é um grafo bi-dirigido com pesos, em que cada peso corresponde à distância de um ponto ao outro e as ligações corresponderão a pontos possíveis de alcançar sem paredes como obstáculos. Cada nodo é um uuid, representante de um ponto na lista de pontos.



4.2 Funções auxiliares

Neste trabalho recorreremos à criação de várias funções auxiliares, de forma a facilitar o nosso trabalho. Aqui vamos explicar algumas dessas funções.

4.2.1 Calcular a distância entre dois pontos (x1, y1) e (x2, y2)

```
import math

def calculateDistance(x1, y1, x2, y2):
    dx = x2-x1
    dy = y2-y1
    distance = math.sqrt((dx**2) + (dy**2))
    return distance
```

Esta função localiza-se no ficheiro CoordHelper.py (Decidimos criar este ficheiro separado, para caso, venha a ser necessário criar mais funções auxiliares relacionadas com Coordenadas) e retorna um valor que representa a distância entre dois pontos no plano x, y.

4.2.2 Registo de um novo ponto de interesse

```
def new_point(coordX, coordY, roomA, roomB):
    uid = uuid.uuid4()
    point_list.append((uid, coordX, coordY, roomA, roomB))
    for point in point_list:
        if point[3] == roomA or point[3] == roomB or point
            ↪ [4] == roomA or point[4] == roomB:
                distance = CoordHelper.calculateDistance(
                    ↪ point[1], point[2], coordX, coordY)
                graph.add_edge(uid, point[0], distance)
```

Esta função é chamada sempre que o robô passa de uma sala para outra. Esta usa a biblioteca uuid para gerar um novo uuid, e guarda os dados na lista de pontos e para cada ponto de interesse alcançável sem obstáculos, calcula a distância e adiciona uma ponte no grafo.

4.2.3 Algoritmo dijkstra

```
def dijkstra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous node, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()
```

```

while current_node != end:
    visited.add(current_node)
    destinations = graph.edges[current_node]
    weight_to_current_node = shortest_paths[current_node][1]

    for next_node in destinations:
        weight = graph.weights[(current_node, next_node)] +
            ↪ weight_to_current_node
        if next_node not in shortest_paths:
            shortest_paths[next_node] = (current_node, weight)
        else:
            current_shortest_weight = shortest_paths[next_node]
            ↪ ][1]
            if current_shortest_weight > weight:
                shortest_paths[next_node] = (current_node, weight)
                ↪ )

    next_destinations = {node: shortest_paths[node] for node in
        ↪ shortest_paths if node not in visited}
    if not next_destinations:
        return "Route Not Possible"
    # next node is the destination with the lowest weight
    current_node = min(next_destinations, key=lambda k:
        ↪ next_destinations[k][1])

# Work back through destinations in shortest path
path = []
while current_node is not None:
    path.append(current_node)
    next_node = shortest_paths[current_node][0]
    current_node = next_node
# Reverse path
path = path[::-1]
return path

```

Para isto usamos o código feito por Ben Keen (Link na Bibliografia). Esta função retornará uma lista de pontos, que representará o menor caminho a percorrer para ir de um ponto até outro.

4.3 Resposta às perguntas

4.3.1 Pergunta 1: How many rooms are not occupied?

```
def question1():
    counterOccupied = 0
    counterNotKnown = 0
    for roomNumber in range(5, len(room_list) + 1):
        counterObj = 0
        for obj in object_list:
            if obj[0] == roomNumber:
                counterObj += 1
                if obj[1] == "person":
                    counterOccupied += 1
        if counterObj == 0:
            counterNotKnown += 1
    print( "There are %d rooms not occupied by people in %d
    ↪ known rooms." % ((10 - counterNotKnown) -
    ↪ counterOccupied), (10 - counterNotKnown) )
```

Para responder a esta pergunta é feita uma análise a todas as salas que o robô conhece e para cada uma das salas vai ser verificado se há pelo menos uma pessoa em cada sala ou não, havendo assim uma contagem das salas que estão ocupadas por pessoas.

4.3.2 Pergunta 2: How many suites did you find until now?

```
def question2():
    counter = 0
    for roomNumber in range(1, len(room_list) + 1):
        if (getRoomType(roomNumber) == "Suite room"):
            counter += 1
    print( "I've found %d Suite rooms so far." % counter )
```

Para responder a esta pergunta é feita uma análise a todas as salas que o robô conhece e todas aquelas que forem do tipo "Suite room", e então retornamos a contagem das mesmas.

4.3.3 Pergunta 3: Is it more probable to find people in the corridors or inside the rooms?

```
def question3():
    counterHall = 0
    counterRooms = 0
    for obj in object_list:
        if obj[1] == "person":
```



```

        if obj[0] <= 4:
            counterHall += 1
        else:
            counterRooms += 1
    if counterHall > counterRooms:
        print( "Is more likely to meet people in the halls"
        ↪ ".")
    elif counterHall < counterRooms:
        print( "Is more likely to meet people in the rooms"
        ↪ ".")
    elif counterHall == 0 and counterRooms == 0:
        print ( "I don't know any person yet." )
    else:
        print( "The probability of find people in rooms or
        ↪ in the halls is equal." )

```

Para responder a esta pergunta contamos o número de pessoas nos corredores e o número de pessoas nas salas, comparamos os dois valores e retornamos resultado dessa comparação.

4.3.4 Pergunta 4: If you want to find a computer, to which type of room do you go to?

```

def question4():
    roomNumber = -1
    for obj in object_list:
        if obj[1] == "computer":
            if getRoomType(obj[0]) == "Meeting_room" or
            ↪ getRoomType(obj[0]) == "Generic_room"
            ↪ : # Only for privacy :)
                roomNumber = obj[0]
                break
            roomNumber = obj[0]
    if roomNumber == -1:
        print( "I don't know any room with a computer." )
    else:
        print( "Go to room number %d to find a Computer."
        ↪ % roomNumber )

```

Para responder a esta pergunta apenas se vai verificar em todas as salas que o robô conhece quais contêm computadores e retornamos uma sala que contenha então um computador.

4.3.5 Pergunta 5: What is the number of the closest single room?

```

def closestSingleRoom(atualX, atualY):
    min_room = -1
    min_distance = 9999999
    for room in room_list:
        if (getRoomType(room[0]) == "Single_room"):
            tempDistance = calculateDistance(atualX,
                ↪ atualY, dijkstraRooms(match_room(
                ↪ atualX, atualY), room[0]))
            if (tempDistance < min_distance):
                min_distance = tempDistance
                min_room = room[0]

    return min_room

def question5():
    csr = closestSingleRoom(x_ant, y_ant)
    if csr != -1:
        print("The closest Single room is %d." % csr)
    else:
        print("I don't know any Single room yet.")

```

Para responder a esta pergunta recorreremos ao cálculo da distância de onde o robô está até cada Single room conhecida, calculando o menor caminho para cada Single room com o algoritmo dijkstra e comparando as distâncias. Assim retornará a Single room mais perto (com menor distância, a andar).

4.3.6 Pergunta 6: How can you go from the current room to the elevator?

```

def question6():
    roomPath = getRoomPath(dijkstraRooms(match_room(x_ant,
        ↪ y_ant), -1), match_room(x_ant, y_ant))
    roomPath = roomPath[1:-1]
    result = "Visit the follow rooms to go to the Elevator:"
    for room in roomPath:
        result += str(room) + " "
    print(result)

```

Para responder a esta pergunta usamos o algoritmo dijkstra para calcular o menor caminho desde a sala atual do robô até ao elevador (representado como a sala -1). Retornamos uma lista ordenada das salas a visitar até chegar ao elevador.

4.3.7 Pergunta 7: How many books do you estimate to find in the next 2 minutes?

```
startTime = time.time()
```

```
def question7():
    actualTime = time.time()
    counterBooks = 0
    for obj in object_list:
        if obj[1] == "book":
            counterBooks += 1
    result = (120 * counterBooks) / (actualTime - startTime)
    print( "I think I will find %d books in the next 2
    ↪ minutes." % int(result))
```

Para responder a esta pergunta recorreremos à biblioteca time, guardando no início do programa o tempo o tempo de início, e cada vez que for feita esta questão um tempo. Para responder à pergunta será dada uma resposta com base no seguinte cálculo:

$(120 * \text{numberOfBooks}) / \text{timeInterval}$

Em que, numberOfBooks corresponde ao número de livros encontrados pelo robô até agora e timeInterval corresponde ao tempo decorrido em segundos desde o início do programa.

4.3.8 Pergunta 8: What is the probability of finding a table in a room without books but that has at least one chair?

```
def question8():
    counterRoomWithChairAndNotBook = 0
    counterRoomWithTableAndChairAndNotBook = 0
    for room in range(5, 14):
        counterBook = 0
        counterChair = 0
        counterTable = 0
        for obj in object_list:
            if obj[0] == room:
                if obj[1] == "chair":
                    counterChair += 1
                if obj[1] == "book":
                    counterBook += 1
                if obj[1] == "table":
                    counterTable += 1
        if counterChair > 0 and counterBook == 0:
            counterRoomWithChairAndNotBook += 1
        if counterChair > 0 and counterTable > 0 and
        ↪ counterBook == 0:
            counterRoomWithTableAndChairAndNotBook += 1
    if counterRoomWithChairAndNotBook == 0:
```

```

        print( "I don't know any room without books but
        ↪ that has at least one chair yet." )
    else:
        result = counterRoomWithTableAndChairAndNotBook /
        ↪ counterRoomWithChairAndNotBook
        print( "The probability of finding a table in a
        ↪ room without books but that has at least one
        ↪ chair is %d." % result )

```

Para responder a esta pergunta usamos a Regra de Bayes, calculando

$$P(T|C, B)$$

em que:

- T corresponde à probabilidade de encontrar uma ou mais Mesas em uma sala.
- C corresponde à probabilidade de encontrar uma ou mais Cadeiras em uma sala.
- B corresponde à probabilidade de encontrar uma ou mais Livros em uma sala.

$$P(T|C, B)$$

Corresponde à probabilidade de encontrar uma ou mais mesas em uma sala sabendo que contem uma ou mais cadeiras e não contem livros.

5 Conclusão

Com este trabalho concluímos que mesmo para um ambiente controlado podem surgir algumas dificuldades em questões que aparentam ser simples, e que assim sendo serão ainda mais complicadas de implementar no mundo real.

5.1 Partição do trabalho

Perguntas 1 a 4: Pedro Maria

Perguntas 5 a 8: David Pires

Relatório:

Motivação, Objetivos e Conclusão: Pedro Maria

Perguntas 1 a 4: Pedro Maria

Perguntas 5 a 8: David Pires

Apresentação: Pedro Maria

6 Bibliografia

- <http://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/>