Universidade de São Paulo

Ana Clara Amorim Andrade 10691992
Ana Cristina Silva de Oliveira 11965630
Marcelo Duchêne 8596351

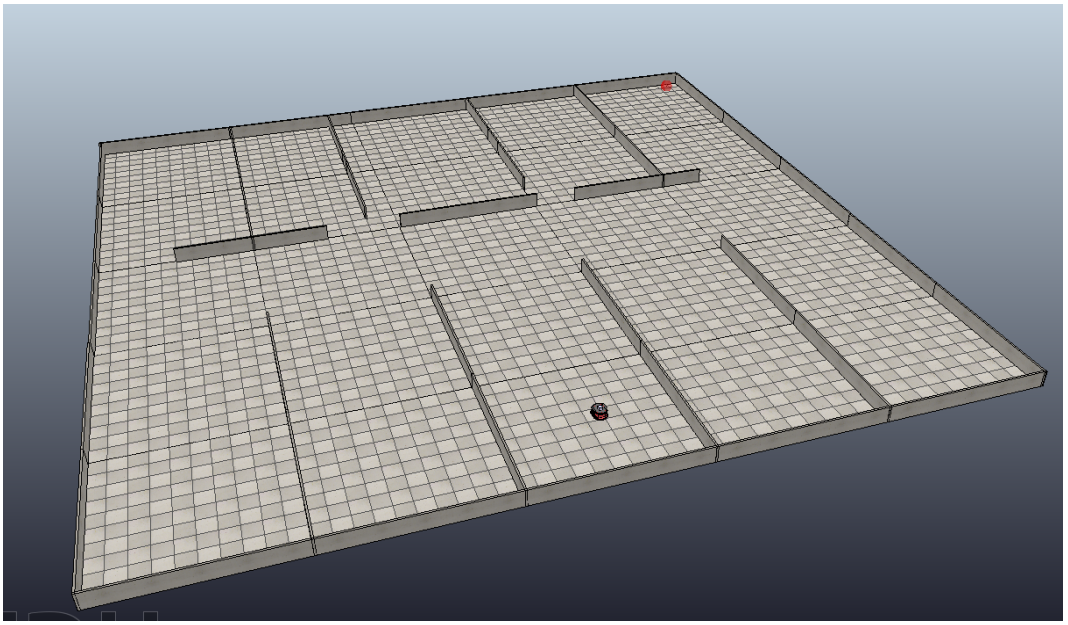
Trabalho final SCC0712 e SCC0714

2021

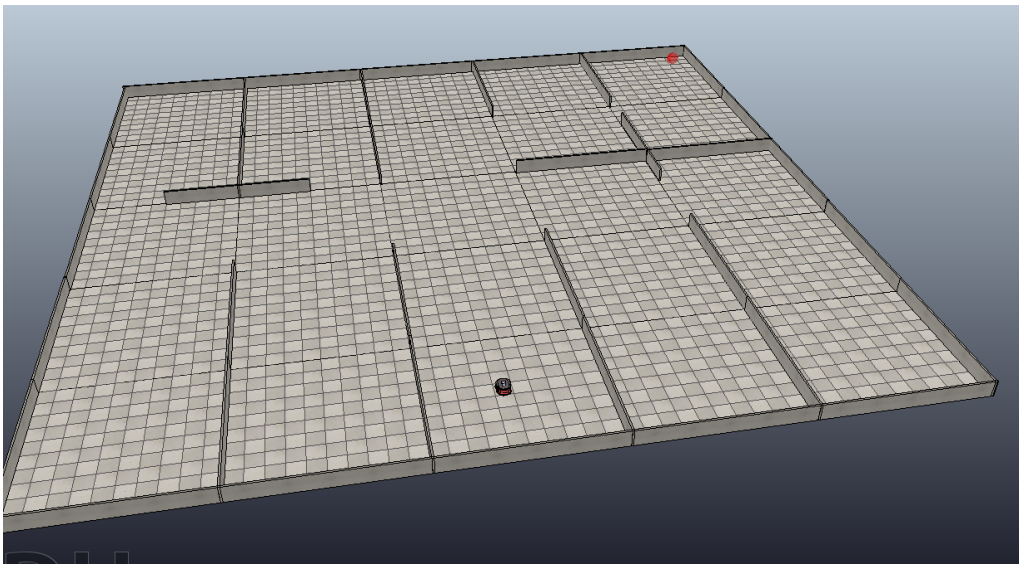
1. Proposta do trabalho

Deverá ser gerada uma representação do mapa do ambiente, para que a partir deste mapa (topológico ou grade) seja então gerada a sequência de pontos de navegação (waypoint). O trabalho será portanto constituído de 2 programas (duas partes), uma que cria o mapa e gera o plano de navegação (waypoint) e outra que faz a navegação seguindo o waypoint. O mapeamento deverá ser obtido de forma automática, seguindo a geração do plano (waypoint) e da navegação.

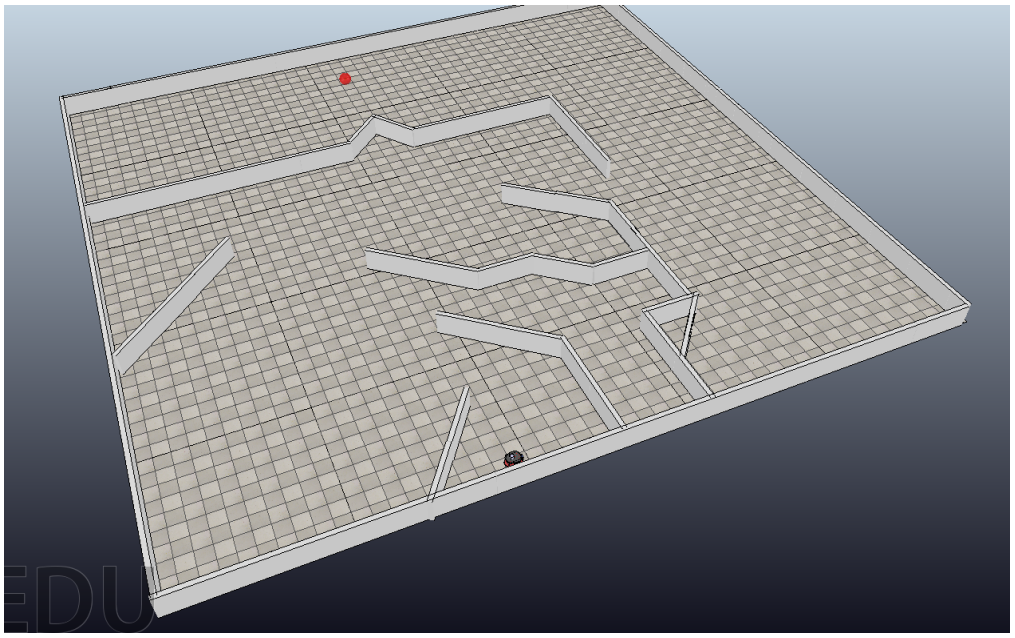
Para isso são disponibilizados 3 mapas, em que devemos através do mapeamento gerar os waypoints de forma a fazer com que o robô chegue ao ponto final (sinal vermelho)



Mapa 1



Mapa 2



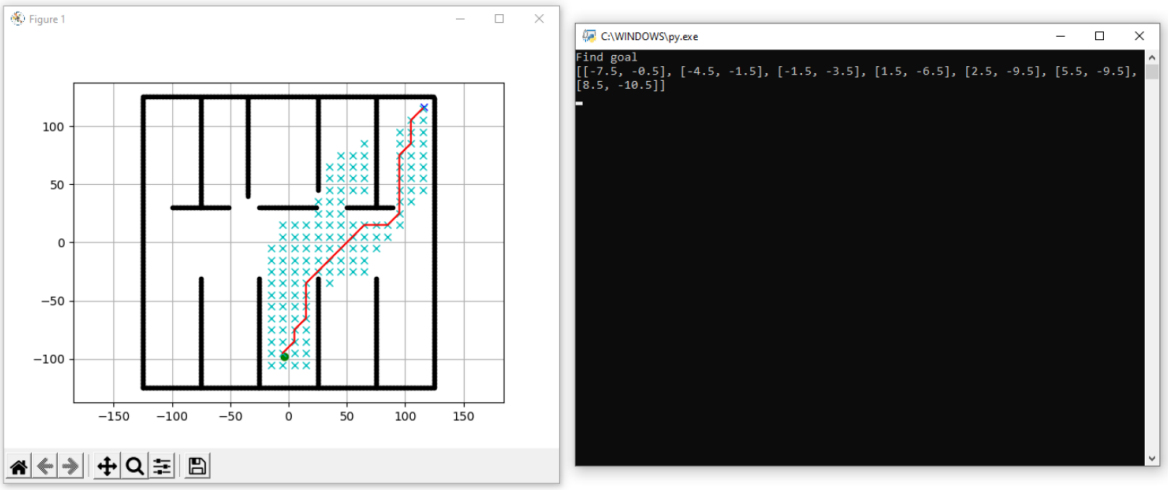
Mapa 3

2. Solução

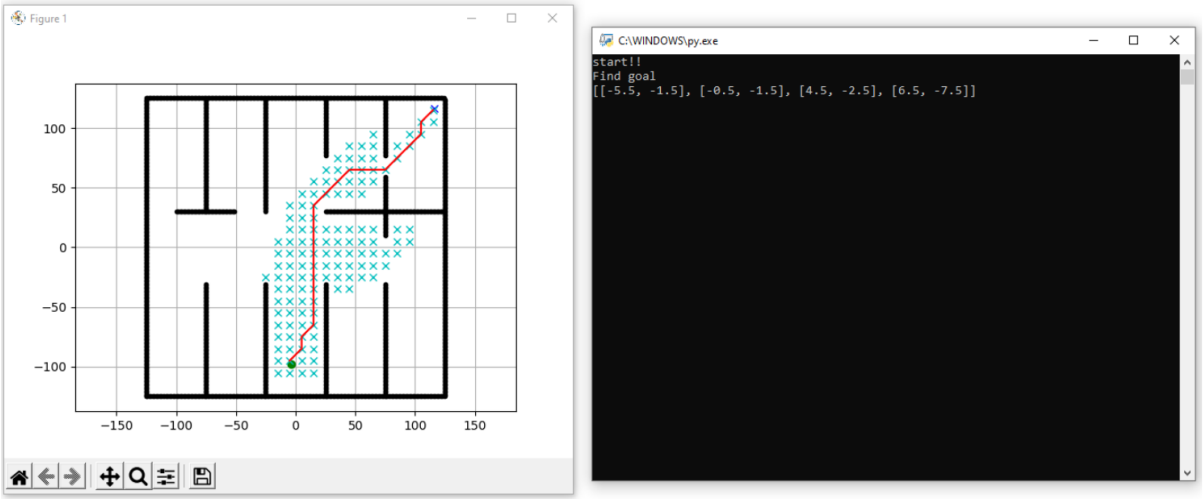
Como solução para este trabalho criamos um programa em python que faz navegação por um algoritmo de A*, que é um algoritmo para **Busca de Caminho**. Ele busca o caminho em um grafo de um vértice inicial até um vértice final. Ele é a combinação de aproximações heurísticas como do algoritmo Breadth First Search (Busca em Largura) e da formalidade do Algoritmo de Dijkstra.

Nesse algoritmo de Python criamos manualmente cada um dos cenários acima, o robô determinado pelo ponto verde encontra um caminho até o destino final representado pelo X azul, esse caminho está representado pela linha em vermelho.

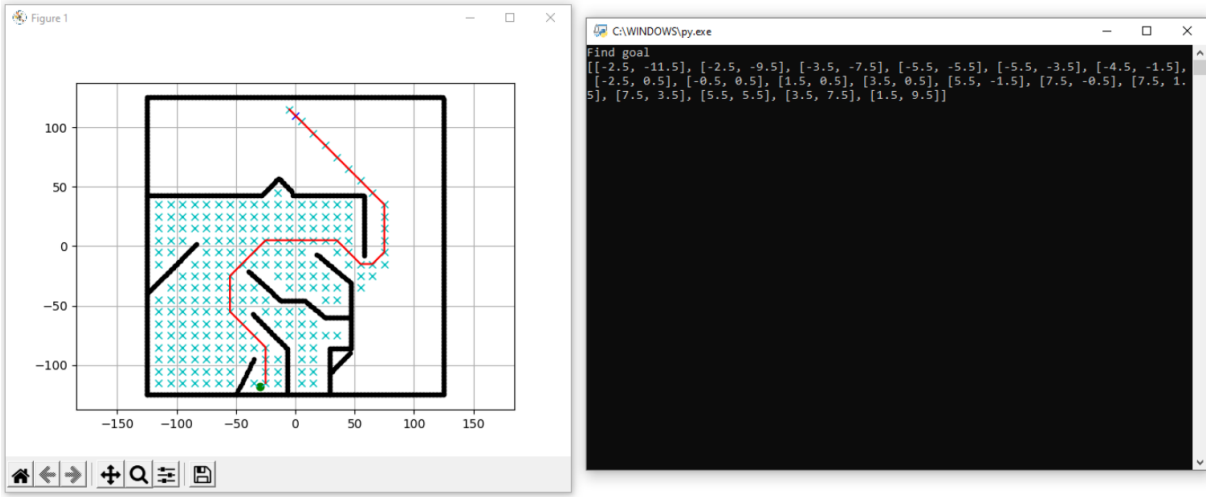
Para gerar os waypoints, imprimimos no terminal uma quantidade de pontos N para cada cenário em cima do caminho que o robô traçou, ou seja, em cima da linha vermelha, após termos gerados os waypoints para cada cenário inserimos eles manualmente no código em LUA do V-rep.



Cenário 1



Cenário 2



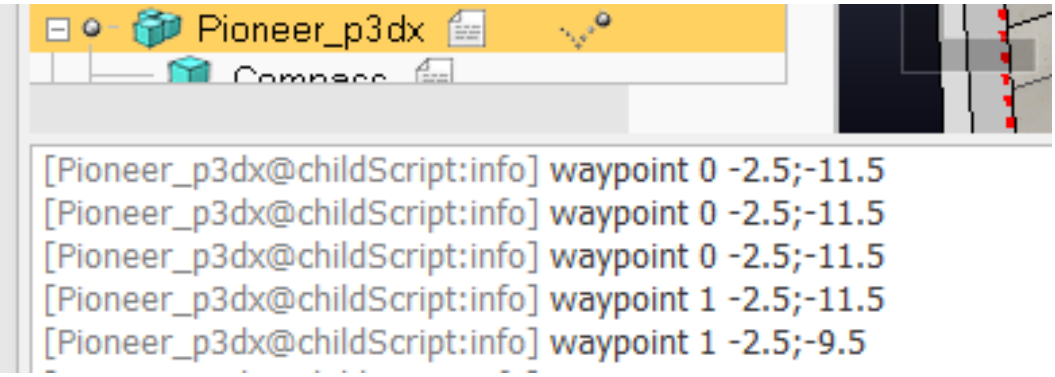
Cenário 3

2.1 Breve explicação código V-rep

O código do V-rep consiste em uma máquina de estados: Girar para a esquerda, girar para a direita, seguir parede e ir até o objetivo, também possui uma condição em que o robô está suficientemente próximo ao waypoint e deve atualizar para o próximo. Todos esses com o objetivo de fazer com que o robô chegue até os waypoints. Nos 3 cenários utilizamos do mesmo código, que diferem entre si somente nas posições dos waypoints e na quantidade dos mesmos.

Sensores utilizados:

- Bússola
- GPS
- 16 sensores ultrassônicos ao redor do robô



Print referente a qual Waypoint o robô está seguindo

2.1 Breve explicação código Python

Realizamos a conversão do mapa do Coppelia Sim utilizando a escala 1:10, para aumentar a precisão, pois o mapa do Python é discreto.

A continuidade das paredes se deve ao fato de definirmos um raio para o robô (robot_radius) que é maior que a distância entre os pontos da parede, garantindo assim que o robô não as atravesse.

```
# coordenadas inicias do robo
sy = -9.8 * 10  # [m]
sx = -0.4 * 10 # [m]
# coordenadas do objetivo
gx = 11.6 * 10 # [m]
gy = 11.6 * 10# [m]
#grid resolução
grid_size = 10.0  # [m]
#tamanho do robo
robot_radius = 5.0  # [m]
```

Inicialização de variáveis Python

```python
#gera obstaculos do mapa
ox, oy = [], []
for i in range(-125, 125):
    ox.append(i)
    oy.append(-125.0)
for i in range(-125, 125):
    ox.append(125.0)
    oy.append(i)
for i in range(-125, 125):
    ox.append(-125.0)
    oy.append(i)
for i in range(-125, 125):
    ox.append(i)
    oy.append(125.0)
```

Geração de mapas

É gerado um caminho final ponto a ponto através da resolução escolhida (gride_size), e imprimimos em cada mapa uma quantidade suficiente de pontos, fazendo a conversão para a escala do Coppelia, para que o robô não fique preso em mínimos locais.

```python
def calc_final_path(self, goal_node, closed_set):
    # gera caminho final
    rx, ry = [self.calc_grid_position(goal_node.x, self.min_x)], [
        self.calc_grid_position(goal_node.y, self.min_y)]
    parent_index = goal_node.parent_index

    cont = 0
    xy = []
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(self.calc_grid_position(n.x, self.min_x))
        ry.append(self.calc_grid_position(n.y, self.min_y))
        parent_index = n.parent_index

        cont = cont + 1
        if cont == 3:
            #Armazena no vetor os pontos em escala do Vrep
            xy.insert(0,[self.calc_grid_position(n.y, self.min_y)/10,(-1*self.calc_grid_position(n.x, self.min_x))/10])
            cont = 0
    #Printa os Waypoints no formato do Vrep
    print(xy)
    return rx, ry
```

Geração caminho final e print dos pontos

3. Como simular o programa

Para rodar o programa Python basta digitar: python *'nome_do_arquivo.py'* e para rodar o programa no Coppelia basta clicar no Start simulation, lembrando que os programas rodam independentemente, pois os pontos gerados no Python já foram inseridos no código do Coppelia.