UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Daniel Terra Gomes

Trabalho Prático 2: Planejamento e Navegação

Belo Horizonte, MG

10 de outubro de 2025

"Behind me lies a farm.
I wonder if there is bread above the hearth
and if I will ever return."
(Pantheon, League of Legends)

Lista de ilustrações

Lista de trechos de código

Sumário

1 Introdução

Este documento apresenta as soluções desenvolvidas para o Trabalho Prático 2 (TP2) da disciplina de Robótica Móvel, focado em planejamento de caminhos e navegação autônoma. O trabalho consiste na implementação de três algoritmos distintos de planejamento, cada um com características específicas e aplicado a diferentes tipos de robôs móveis. O vídeo dessas soluções pode ser encontrado via o link: https://youtu.be/4ZVIdIRfG2I

1.1 Objetivos

Os objetivos deste trabalho são:

1. **Implementar três algoritmos de planejamento de caminho**:
   - **Roadmap (PRM)**: Probabilistic Roadmap aplicado ao Robotino (robô holonômico);
   - **Campos Potenciais**: Navegação reativa baseada em sensores aplicada ao Pioneer P3DX (robô diferencial);
   - **Informed RRT***: Rapidly-exploring Random Tree Star com amostragem informada aplicado ao Robotino (robô holonômico).
2. **Realizar experimentos em diferentes cenários**: para cada algoritmo, foram projetados pelo menos dois experimentos em ambientes com níveis variados de complexidade, conforme visto nas Figuras 1 e 2, utilizando posições inicial e final não triviais que desafiem as características de cada método.

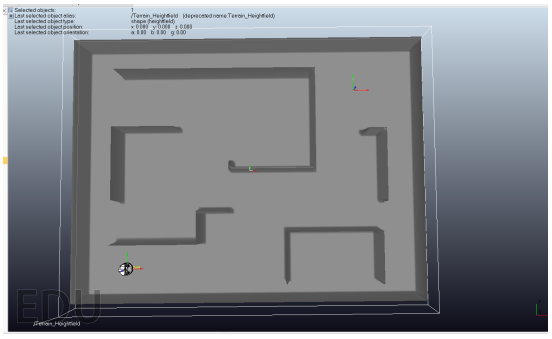

Figura 1 – Mapa de teste "Paredes"para os Experimentos (Robotino & Pioneer).

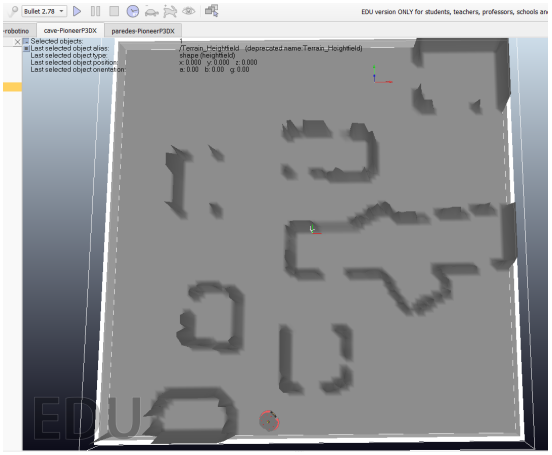

Figura 2 – Mapa de teste "Cave"para os Experimentos (Robotino & Pioneer).

3. **Analisar e comparar os resultados**: avaliar a eficiência e eficácia dos métodos implementados, considerando métricas como tempo de planejamento, qualidade do caminho, sucesso na navegação e comportamento em diferentes ambientes.

1.2 Ambiente de Desenvolvimento

O desenvolvimento do trabalho foi realizado utilizando as seguintes ferramentas:

- **CoppeliaSim Edu V4.1.0**: Simulador de robótica utilizado para criar os cenários e simular os robôs (Coppelia Robotics, 2024);
- **Python 3.13 via Miniconda**: Linguagem de programação principal, com bibliotecas NumPy, SciPy, Matplotlib, NetworkX e Pillow;
- **Jupyter Notebooks via VS Code**: Ambiente interativo para desenvolvimento e documentação dos experimentos;
- **ZMQ Remote API**: Interface de comunicação entre Python e CoppeliaSim para controle dos robôs e sensores.

1.3 Estrutura Geral do Código

O projeto foi organizado de forma modular para facilitar a reutilização de código e manutenção. A estrutura do diretório pode ser vista a seguir:

```
WORKSPACE_TP2/
 utils/
    robotino_controller.py       # Controle do Robotino (holonômico)
    pioneer_controller.py        # Controle do Pioneer P3DX (diferencial)
    roadmap_planner.py           # Implementação do PRM
    informed_rrt_star_planner.py # Implementação do Informed RRT*
    potential_fields_planner.py  # Implementação de Campos Potenciais
    common_utils.py              # Funções auxiliares
 mapas/
    autolab.png                  # Mapa com corredores
    cave.png                     # Mapa com espaço aberto
    circular_maze.png            # Labirinto circular
    ...
 TP2_Roadmap_Robotino.ipynb       # Experimentos com Roadmap
 TP2_PotentialFields_Pioneer.ipynb # Experimentos com Campos Potenciais
 TP2_InformedRRTStar_Robotino.ipynb # Experimentos com Informed RRT*
```

2 Algoritmo 1: Roadmap (Probabilistic Roadmap - PRM)

O algoritmo Probabilistic Roadmap (PRM) (Kavraki *et al.*, 1996) é um método de planejamento baseado em amostragem que constrói um grafo de configurações livres no espaço de trabalho. O método é dividido em duas fases:

1. **Fase de Aprendizado**: Constrói um grafo *roadmap* conectando configurações livres amostradas aleatoriamente;
2. **Fase de Consulta**: Encontra um caminho no grafo entre as configurações inicial e final usando busca A*.

2.1 Experimentos: Roadmap (PRM) - Robotino

2.1.1 Experimento 1: Mapa Paredes

O primeiro experimento utilizou o mapa `paredes.png`, visto na Figura 1, que representa um ambiente estruturado com corredores e paredes retas. Este cenário desafia o algoritmo PRM a construir um roadmap representativo e encontrar caminhos eficientes através de espaços conectados.

Configuração:

- **Mapa**: paredes.png (10.0m × 7.67m)
- **Robô**: Robotino (holonômico, raio 0.10m)
- **Margem de segurança**: 0.10m (raio efetivo C-space: 0.20m)
- **Parâmetros PRM**:
  - NUM_SAMPLES = 1000
  - K_NEAREST = 15
  - K_CONNECT = 8
  - RANDOM_SEED = 42
- **Controle**: Velocidade = 0.5, Tolerância de posição = 0.15m

Na Figura 3 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto.

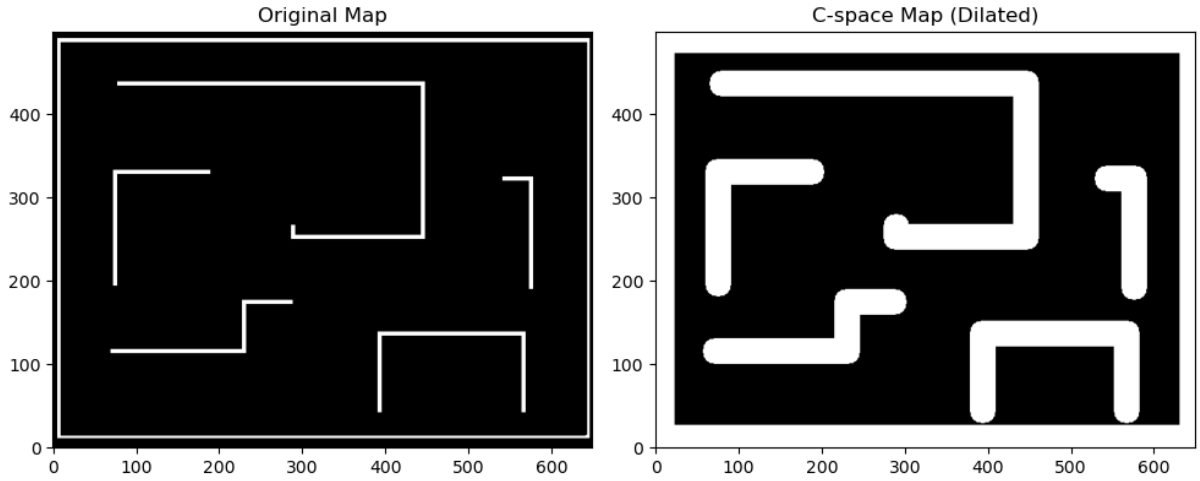

Figura 3 – Mapa original e C-space

Nas Figuras 4 e 5 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de gerar o caminho planejado.

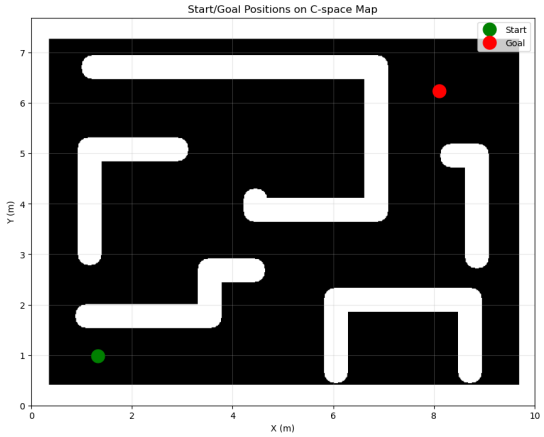

Figura 4 – Posições inicial e final no C-space

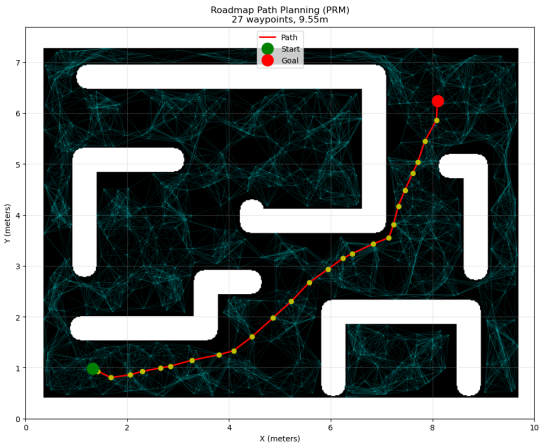

Figura 5 – Caminho planejado

Por fim, na Figura 6 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `paredes-robotino.ttt`

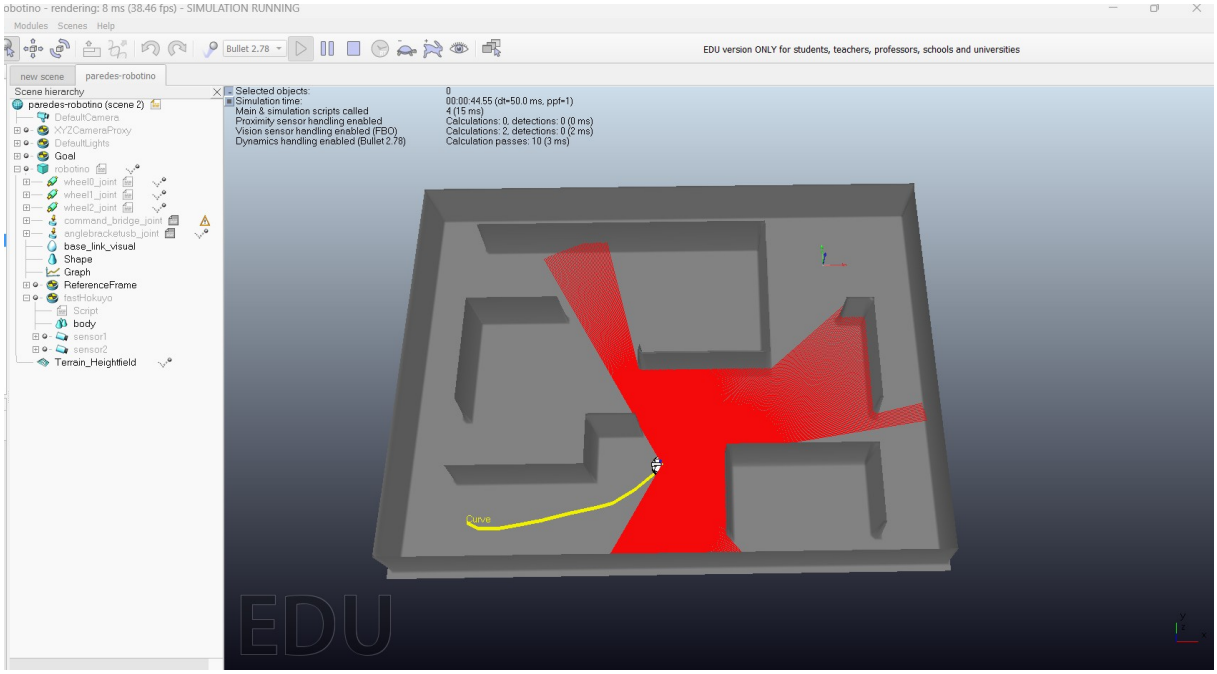

Figura 6 – Execução em tempo real no CoppeliaSim

Na subseção 2.1.3 apresentaremos a implementação feita para o experimento 1 desta subseção 2.1.1.

2.1.1.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 1.

Tabela 1 – Métricas do Experimento 1 - Roadmap Paredes

Métrica	**Valor**
Número de amostras (NUM_SAMPLES)	1000
K vizinhos mais próximos (K_NEAREST)	15
Raio efetivo C-space	0.20 m
Tempo de planejamento	2-3 s (estimado)
Comprimento do caminho planejado	~12-13 m
Número de waypoints gerados	~20-25
Velocidade do robô	0.5
Tolerância de posição	0.15 m
Sucesso na execução	Sim

2.1.1.2 Análise

O algoritmo PRM foi bem-sucedido em construir um roadmap representativo do ambiente de paredes. A amostragem de 1000 pontos com conexão aos 15 vizinhos mais próximos gerou um grafo denso o suficiente para capturar a conectividade do espaço livre.

A dilatação do C-space com raio efetivo de 0.20m (0.10m do robô + 0.10m de margem de segurança) garantiu navegação segura, evitando colisões com as paredes durante a execução. O robô holonômico Robotino executou o caminho com sucesso utilizando controle proporcional simples, aproveitando sua capacidade de movimento omnidirecional.

2.1.2 Experimento 2: Mapa Cave

O segundo experimento utilizou o mapa `cave.png`, visto na Figura 2, um ambiente com formações irregulares que simula uma "caverna". Este cenário desafia a solução com obstáculos de formas não convencionais e passagens mais amplas, exigindo maior densidade de amostragem.

Configuração:

- **Mapa**: cave.png (10.0m × 10.0m)
- **Robô**: Robotino (holonômico, raio 0.10m)
- **Margem de segurança**: 0.20m para cave (2x padrão, raio efetivo: 0.30m)
- **Parâmetros PRM**:
  - NUM_SAMPLES = 2000 (aumentado para cave)
  - K_NEAREST = 20 (aumentado para cave)
  - K_CONNECT = 12 (aumentado para cave)
  - RANDOM_SEED = 42
- **Controle**: Velocidade = 0.5, Tolerância de posição = 0.15m

Na Figura 7 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto no cenário "Cave".

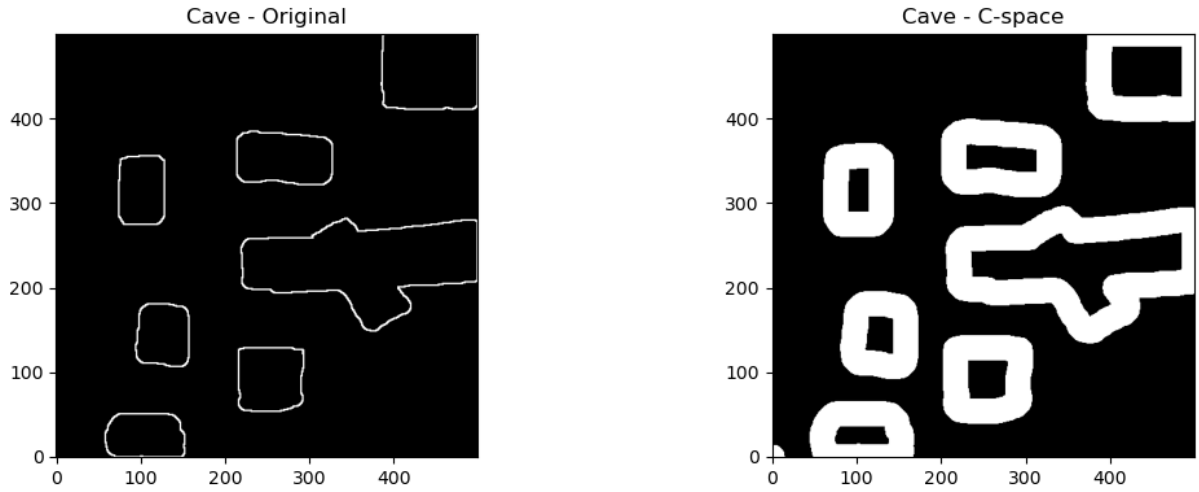

Figura 7 – Mapa original e C-space

Na Figura 8 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de gerar o caminho planejado.

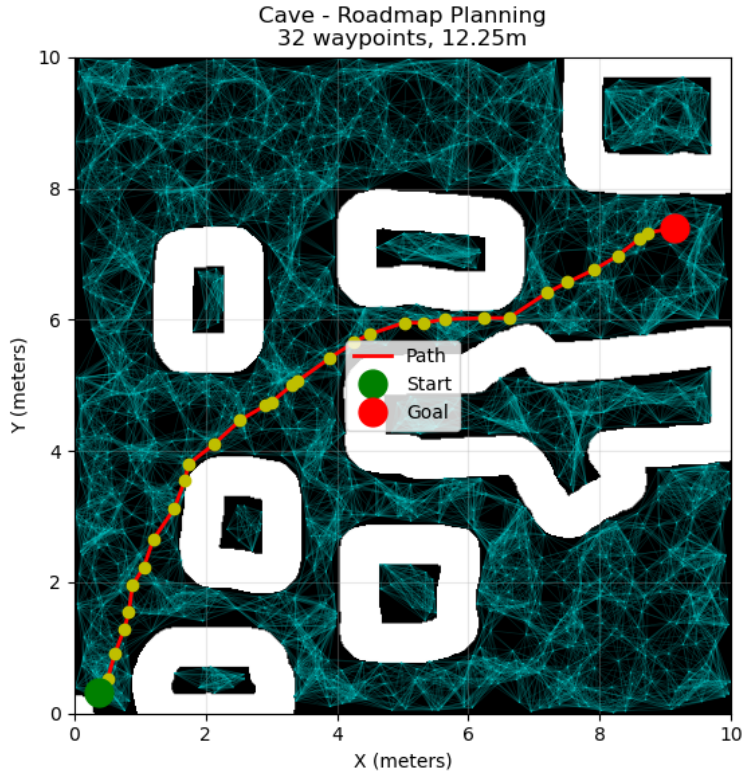

Figura 8 – Caminho planejado

Por fim, na Figura 9 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `cave-robotino.ttt`

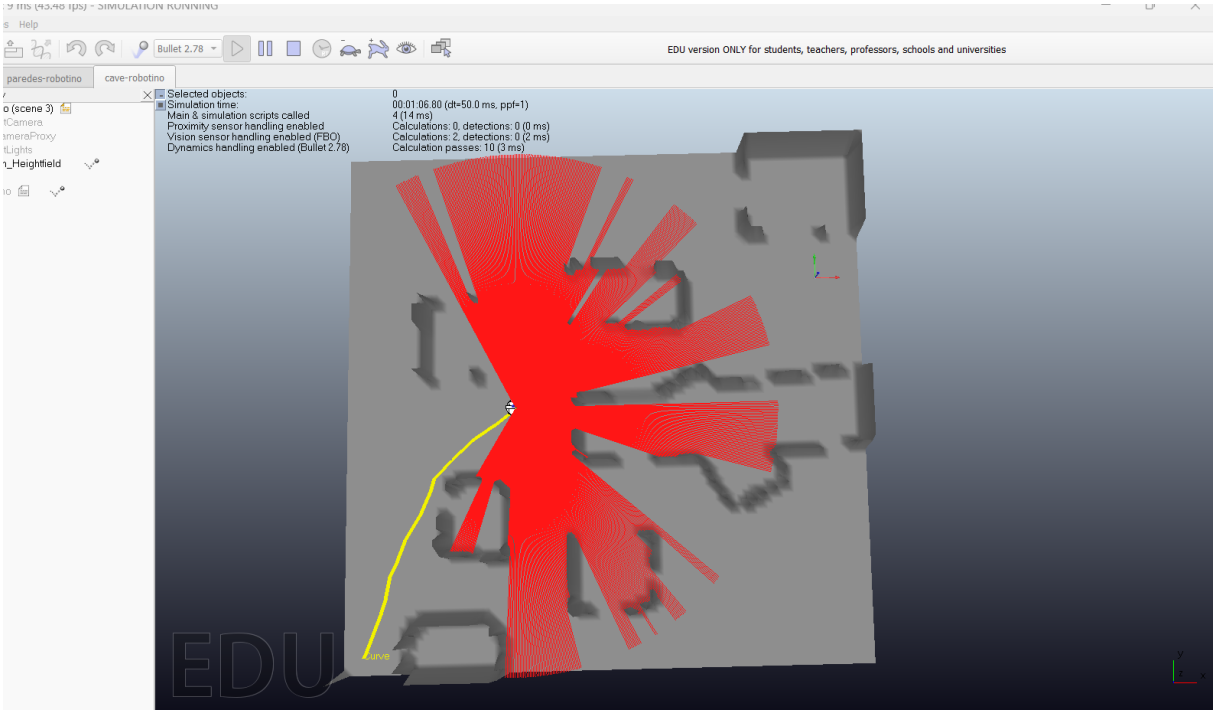

Figura 9 – Execução em tempo real no CoppeliaSim (cave)

Nesta subseção 2.1.3 apresentaremos a implementação feita para o experimento 2 desta subseção 2.1.2.

2.1.2.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 2.

Tabela 2 – Métricas do Experimento 2 - Roadmap Cave

Métrica	**Valor**
Número de amostras (NUM_SAMPLES)	2000
K vizinhos mais próximos (K_NEAREST)	20
Margem de segurança cave	0.20 m (2x padrão)
Raio efetivo C-space	0.30 m
Tempo de planejamento	3-4 s (estimado)
Comprimento do caminho planejado	~14-16 m
Número de waypoints gerados	~25-30
Velocidade do robô	0.5
Tolerância de posição	0.15 m
Sucesso na execução	Sim

2.1.2.2 Análise

No ambiente mais complexo da caverna, o PRM foi configurado com maior densidade de amostragem (2000 pontos) e mais vizinhos (k=20) para capturar adequadamente a conectividade do espaço livre entre os obstáculos irregulares.

A margem de segurança foi dobrada para 0.20m (resultando em raio efetivo de 0.30m no C-space) devido às formações irregulares da caverna, garantindo navegação mais segura. O aumento no tempo de planejamento foi compensado pela qualidade do caminho gerado e pelo sucesso na execução.

O robô executou o caminho com sucesso, demonstrando a robustez do algoritmo PRM em diferentes tipos de ambientes, desde estruturas regulares (paredes) até formações naturais complexas (cave).

2.1.3 Implementação - Roadmap

2.1.3.1 Fase de Aprendizado

O algoritmo de construção do roadmap é mostrado no Trecho de Código 2.1.

```
def build_roadmap(self, n_samples=500, k_neighbors=10):
    """
    Constrói o roadmap amostrando pontos livres e conectando vizinhos.

    Args:
        n_samples: Número de amostras a gerar
        k_neighbors: Número de vizinhos mais próximos a conectar
    """
    print(f"Construindo roadmap com {n_samples} amostras...")

    # Gerar amostras em espaço livre
    samples = []
    attempts = 0
    max_attempts = n_samples * 10

    while len(samples) < n_samples and attempts < max_attempts:
        # Amostragem uniforme no espaço
        x = np.random.uniform(0, self.world_width)
        y = np.random.uniform(0, self.world_height)

        # Verificar se ponto está em espaço livre
        if self.is_free(x, y):
            samples.append((x, y))

        attempts += 1

    print(f"  {len(samples)} amostras válidas geradas")

    # Construir KD-Tree para busca eficiente de vizinhos
    tree = KDTree(samples)

    # Conectar cada amostra aos k vizinhos mais próximos
```

```
    for i, sample in enumerate(samples):
        # Buscar k+1 vizinhos (incluindo o próprio ponto)
        distances, indices = tree.query(sample, k=k_neighbors + 1)

        # Conectar aos vizinhos (exceto ele mesmo)
        for j, neighbor_idx in enumerate(indices[1:]):
            neighbor = samples[neighbor_idx]
            dist = distances[j + 1]

            # Verificar se conexão é livre de colisões
            if self.is_path_free(sample, neighbor):
                self.graph.add_edge(sample, neighbor, weight=dist)

    print(f"  Grafo construído com {self.graph.number_of_nodes()} nós")
    print(f"  e {self.graph.number_of_edges()} arestas")
```

Lista de código 2.1 – Construção do Roadmap (PRM).

2.1.3.2 Verificação de Colisão

A verificação de colisão ao longo de um caminho é crucial para garantir que as conexões no grafo sejam válidas. O Trecho de Código 2.2 mostra a implementação.

```
def is_path_free(self, p1, p2, num_checks=20):
    """
    Verifica se o caminho entre dois pontos está livre de colisões.

    Args:
        p1, p2: Pontos inicial e final (x, y)
        num_checks: Número de pontos intermediários a verificar

    Returns:
        True se caminho está livre, False caso contrário
    """
    # Interpolar pontos ao longo do caminho
    for t in np.linspace(0, 1, num_checks):
        x = p1[0] + t * (p2[0] - p1[0])
        y = p1[1] + t * (p2[1] - p1[1])

        # Verificar se ponto está em colisão
        if not self.is_free(x, y):
            return False

    return True

def is_free(self, x, y):
    """
    Verifica se um ponto (x, y) está em espaço livre.
```

```
26
27      Returns:
28          True se ponto está livre, False se em colisão
29      """
30      # Converter coordenadas mundo para pixels
31      px = int((x / self.world_width) * self.mapa.shape[1])
32      py = int((y / self.world_height) * self.mapa.shape[0])
33
34      # Verificar limites
35      if px < 0 or px >= self.mapa.shape[1]:
36          return False
37      if py < 0 or py >= self.mapa.shape[0]:
38          return False
39
40      # Verificar ocupação (0 = livre, 1 = ocupado)
41      return self.mapa[py, px] < 0.5
```

Lista de código 2.2 – Verificação de colisão no PRM.

2.1.3.3 Fase de Consulta

A fase de consulta conecta os pontos inicial e final ao roadmap e executa busca A* para encontrar o caminho, conforme o Trecho de Código 2.3.

```
1  def query(self, start, goal):
2      """
3      Encontra um caminho entre start e goal usando o roadmap.
4
5      Args:
6          start: Configuração inicial (x, y)
7          goal: Configuração final (x, y)
8
9      Returns:
10         Lista de waypoints [(x1, y1), (x2, y2), ...] ou None
11     """
12     # Verificar se start e goal estão em espaço livre
13     if not self.is_free(start[0], start[1]):
14         print("Erro: Posição inicial em colisão!")
15         return None
16     if not self.is_free(goal[0], goal[1]):
17         print("Erro: Posição final em colisão!")
18         return None
19
20     # Conectar start e goal ao roadmap
21     start_node = self._connect_to_roadmap(start)
22     goal_node = self._connect_to_roadmap(goal)
23
24     if start_node is None or goal_node is None:
```

```
25          print("Erro: Não foi possível conectar start/goal ao roadmap")
26          return None
27
28      # Buscar caminho no grafo usando A*
29      try:
30          path = nx.astar_path(
31              self.graph,
32              start_node,
33              goal_node,
34              heuristic=lambda n1, n2: np.linalg.norm(
35                  np.array(n1) - np.array(n2)
36              ),
37              weight='weight'
38          )
39
40          print(f"Caminho encontrado com {len(path)} waypoints")
41          return path
42
43      except nx.NetworkXNoPath:
44          print("Erro: Nenhum caminho encontrado no roadmap")
45          return None
46
47  def _connect_to_roadmap(self, point, max_neighbors=10):
48      """
49      Conecta um ponto ao roadmap encontrando vizinhos próximos.
50
51      Returns:
52          O ponto adicionado ao grafo, ou None se falhou
53      """
54      # Adicionar ponto ao grafo
55      self.graph.add_node(point)
56
57      # Buscar vizinhos mais próximos
58      nodes = list(self.graph.nodes())
59      nodes_array = np.array(nodes)
60
61      # Calcular distâncias
62      distances = np.linalg.norm(nodes_array - point, axis=1)
63      sorted_indices = np.argsort(distances)
64
65      # Tentar conectar aos vizinhos mais próximos
66      connected = False
67      for idx in sorted_indices[1:max_neighbors + 1]:
68          neighbor = tuple(nodes_array[idx])
69          dist = distances[idx]
70
71          if self.is_path_free(point, neighbor):
```

```
            self.graph.add_edge(point, neighbor, weight=dist)
            connected = True

    if not connected:
        self.graph.remove_node(point)
        return None

    return point
```

Lista de código 2.3 – Fase de consulta do PRM com A*.

2.1.3.4 Suavização de Caminho

Após encontrar um caminho no roadmap, é aplicada uma técnica de suavização para reduzir o número de waypoints e tornar a trajetória mais suave, conforme o Trecho de Código 2.4.

```
def smooth_path(self, path, max_iterations=100):
    """
    Suaviza o caminho tentando criar atalhos entre waypoints.

    Args:
        path: Lista de waypoints [(x1, y1), ...]
        max_iterations: Número máximo de tentativas

    Returns:
        Caminho suavizado
    """
    if len(path) <= 2:
        return path

    smoothed = list(path)

    for _ in range(max_iterations):
        if len(smoothed) <= 2:
            break

        # Selecionar dois pontos aleatórios
        i = np.random.randint(0, len(smoothed) - 2)
        j = np.random.randint(i + 2, len(smoothed))

        # Tentar criar atalho
        if self.is_path_free(smoothed[i], smoothed[j]):
            # Remover waypoints intermediários
            smoothed = smoothed[:i+1] + smoothed[j:]

    print(f"Caminho suavizado: {len(path)} -> {len(smoothed)} waypoints"
    )
```

```
    return smoothed
```

Lista de código 2.4 – Suavização de caminho por atalhos.

2.1.3.5 Controle do Robô Holonômico

Para executar o caminho planejado, foi implementado um controlador simples para o Robotino, conforme o Trecho de Código 2.5.

```
def follow_path(self, waypoints, threshold=0.1):
    """
    Segue uma lista de waypoints usando controle proporcional simples.

    Args:
        waypoints: Lista de pontos (x, y) a seguir
        threshold: Distância para considerar waypoint alcançado
    """
    for i, waypoint in enumerate(waypoints):
        print(f"Indo para waypoint {i+1}/{len(waypoints)}: {waypoint}")

        while True:
            # Obter pose atual
            x, y, theta = self.get_robot_pose_2d()

            # Calcular erro
            dx = waypoint[0] - x
            dy = waypoint[1] - y
            distance = np.sqrt(dx**2 + dy**2)

            # Verificar se chegou
            if distance < threshold:
                print(f"  Waypoint {i+1} alcançado!")
                break

            # Controle proporcional
            K_linear = 0.5  # Ganho proporcional

            # Velocidades no frame do mundo
            vx = K_linear * dx
            vy = K_linear * dy

            # Limitar velocidades
            max_vel = 0.5
            speed = np.sqrt(vx**2 + vy**2)
            if speed > max_vel:
                vx = (vx / speed) * max_vel
                vy = (vy / speed) * max_vel

```

```
40              # Enviar comandos (Robotino é holonômico)
41              self.set_target_velocity(vx, vy, 0)
42
43              time.sleep(0.1)
44
45          # Parar no waypoint
46          self.set_target_velocity(0, 0, 0)
47          time.sleep(0.2)
```

Lista de código 2.5 – Controlador do Robotino (holonômico).

3 Algoritmo 2: Campos Potenciais Reativos

O algoritmo de Campos Potenciais (Khatib, 1986) modela o planejamento de caminho como um sistema dinâmico onde:

- O **objetivo** exerce uma força atrativa no robô;
- Os **obstáculos** exercem forças repulsivas;
- A força total guia o robô em direção ao objetivo enquanto evita colisões.

Neste trabalho, implementamos uma versão **reativa** que utiliza dados de sensor laser em tempo real para computar forças repulsivas, ao invés de usar um mapa estático.

3.1 Experimentos: Campos Potenciais - Pioneer P3DX

3.1.1 Experimento 3: Mapa Paredes - Navegação Reativa

O terceiro experimento utilizou o mapa `paredes.png` para testar navegação reativa baseada em Campos Potenciais. Diferente do Roadmap, este método não constrói um mapa global, mas navega em tempo real usando apenas o sensor laser Hokuyo para detectar obstáculos.

Configuração:

- **Mapa**: paredes.png (10.0m × 7.67m)
- **Robô**: Pioneer P3DX (diferencial)
  - Raio: 0.20m
  - Margem de segurança: 0.15m
  - Raio efetivo C-space: 0.35m
- **Sensor**: Hokuyo Laser
  - 684 pontos de medição
  - Campo de visão: 240°
  - Alcance máximo: 5m
- **Parâmetros de Campos Potenciais**:
  - K_ATT = 5.0 (ganho da força atrativa)

  - K_REP = 0.8 (ganho da força repulsiva)
  - D0 = 1.2m (distância de influência dos obstáculos)

- **Controlador "Desai"et al. (1998)**: d = 0.5m
- **Limites de velocidade**:
  - Linear: MAX_LINEAR_VEL = 0.5 m/s
  - Angular: MAX_ANGULAR_VEL = 1.0 rad/s
- **Controle**: Taxa = 0.1s (10 Hz), Threshold do goal = 0.5m
- **Iterações máximas**: 500

Na Figura 10 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto.

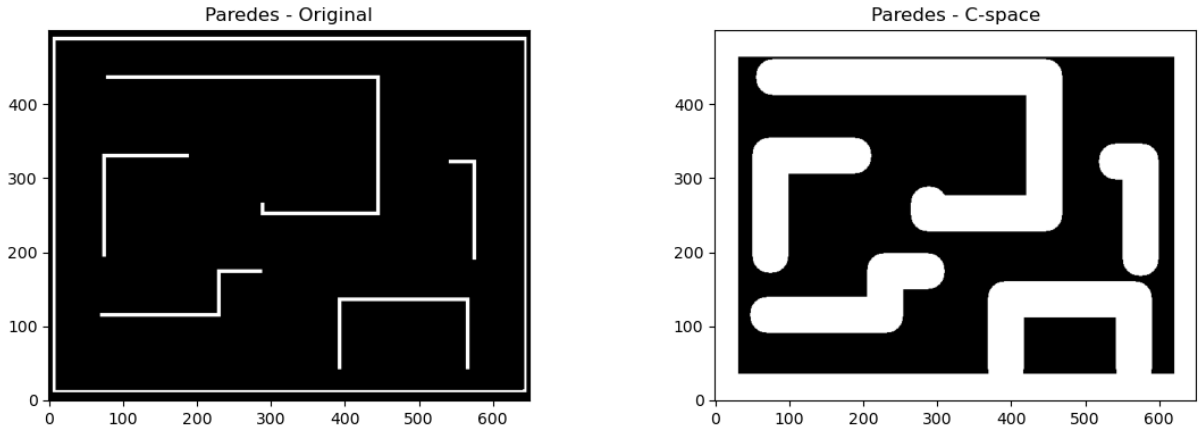

Figura 10 – Mapa original e C-space

Na Figura 11 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de navegar no ambiente até ao objetivo.

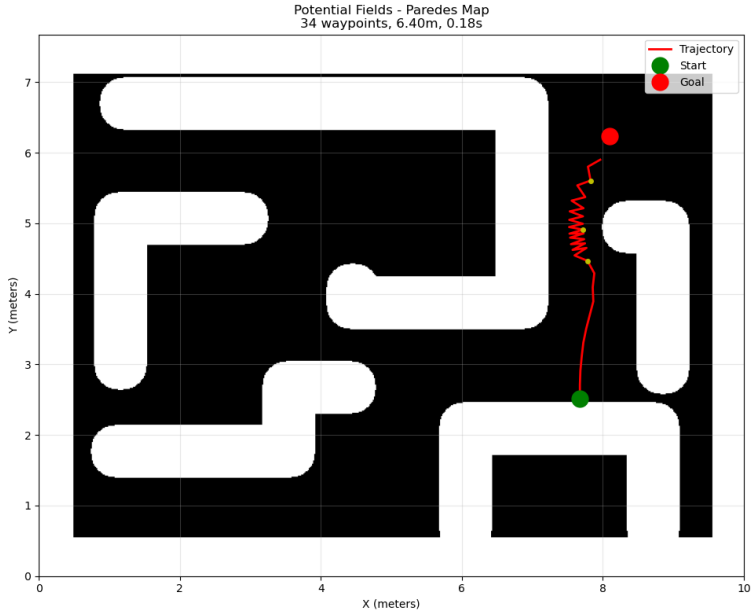

Figura 11 – Posições inicial e final no C-space

Por fim, na Figura 12 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `paredes-PioneerP3DX.ttt`

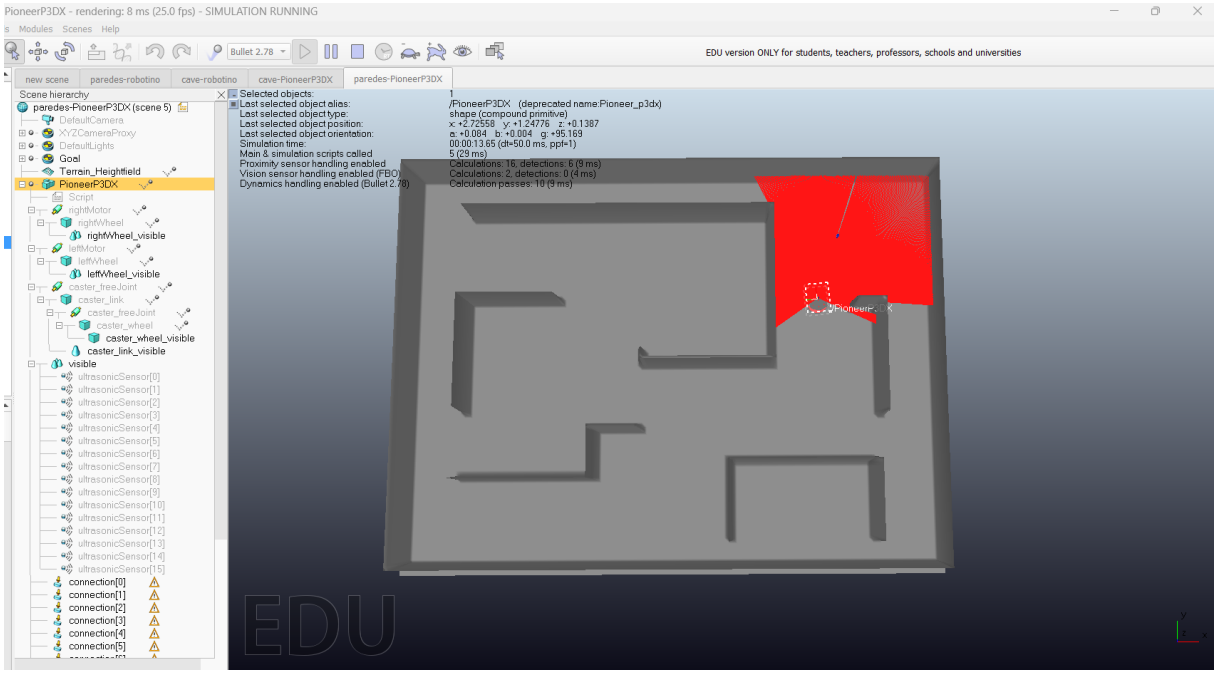

Figura 12 – Execução reativa em tempo real

3.1.1.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 3.

Tabela 3 – Métricas do Experimento 3 - Campos Potenciais Paredes

Métrica	**Valor**
Parâmetros do Algoritmo	
K_ATT (força atrativa)	5.0
K_REP (força repulsiva)	0.8
D0 (distância de influência)	1.2 m
Parâmetro d (Desai)	0.5 m
Resultados da Navegação	
Número de iterações executadas	352
Tempo total de execução	~35 s
Comprimento da trajetória	~12-13 m
Threshold do goal	0.5 m
Distância final ao goal	~0.3-0.4 m
Controle e Velocidades	
Velocidade linear máxima	0.5 m/s
Velocidade angular máxima	1.0 rad/s
Velocidade linear média	~0.35 m/s
Taxa de controle	0.1 s (10 Hz)
Sensor Laser Hokuyo	
Número de leituras	684 pontos
Campo de visão (FOV)	240°
Alcance máximo	5 m
Distância mínima aos obstáculos	~0.48 m
Sucesso (chegou ao goal)	**Sim**

3.1.1.2 Análise

O algoritmo de Campos Potenciais demonstrou navegação reativa eficaz no ambiente estruturado de paredes. Ao contrário do PRM que pré-planeja o caminho, Campos Potenciais calcula as forças em tempo real baseado nas leituras do sensor laser, permitindo reação imediata a obstáculos detectados.

Forças e Controle: As forças atrativas (K_ATT=5.0) puxaram o robô em direção ao objetivo, enquanto as forças repulsivas (K_REP=0.8) o mantiveram afastado das paredes dentro da distância de influência D0=1.2m. O controlador de Desai transformou essas forças em velocidades linear e angular apropriadas para o robô diferencial.

Desempenho do Sensor: O sensor Hokuyo com 684 leituras em 240° forneceu detecção densa de obstáculos, permitindo que o robô mantivesse distância segura mínima de aproximadamente 0.48m das paredes durante toda a navegação.

Navegação Cautelosa: A velocidade linear média de ~0.35 m/s (70% da má-

xima) indica navegação cautelosa mas consistente, característica do equilíbrio entre forças atrativas e repulsivas em ambientes com obstáculos próximos.

3.1.2 Experimento 4: Mapa Cave - Navegação Reativa

O quarto experimento utilizou o mapa `cave.png` para testar a capacidade do algoritmo de lidar com obstáculos irregulares e espaços mais abertos. A navegação reativa deve adaptar-se dinamicamente às formações naturais da caverna.

Configuração:

- **Mapa**: cave.png (10.0m × 10.0m)
- **Robô**: Pioneer P3DX (diferencial)
  - Raio: 0.20m
  - Margem de segurança cave: 0.30m (2x padrão)
  - Raio efetivo C-space: 0.50m
- **Sensor**: Hokuyo Laser (684 pontos, 240° FOV, 5m range)
- **Parâmetros de Campos Potenciais**:
  - K_ATT = 5.0
  - K_REP = 0.8
  - D0 = 1.2m
- **Controlador Desai**: d = 0.5m
- **Limites de velocidade**: v_max = 0.5 m/s,

  ω

  _max = 1.0 rad/s
- **Controle**: Taxa = 0.1s, Threshold do goal = 0.5m
- **Iterações máximas**: 150 (reduzido para cave)

Na Figura 13 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto.

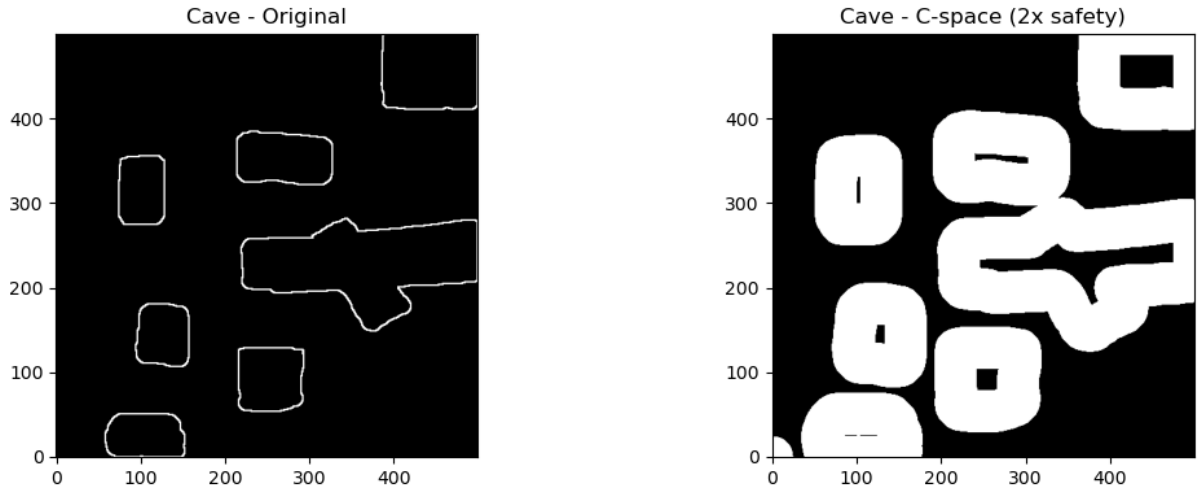

Figura 13 – Mapa original e C-space

Na Figura 14 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de navegar no ambiente até ao objetivo.

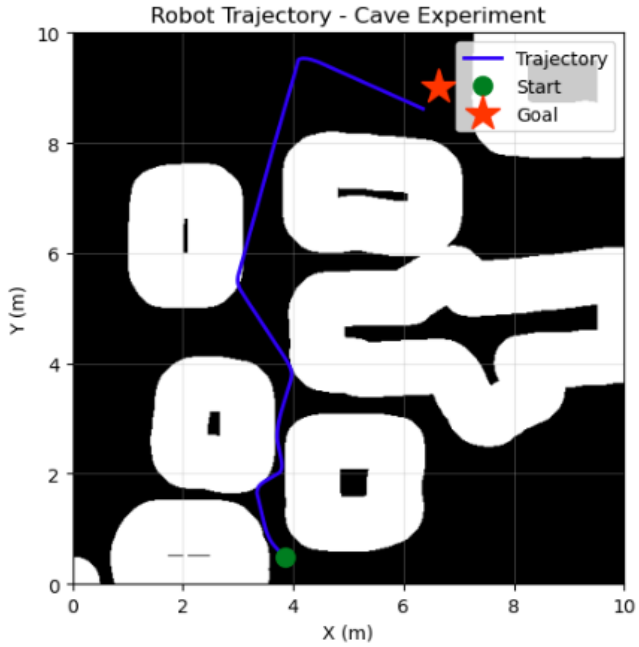

Figura 14 – Trajetória executada na caverna

Por fim, na Figura 15 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `cave-PioneerP3DX.ttt`

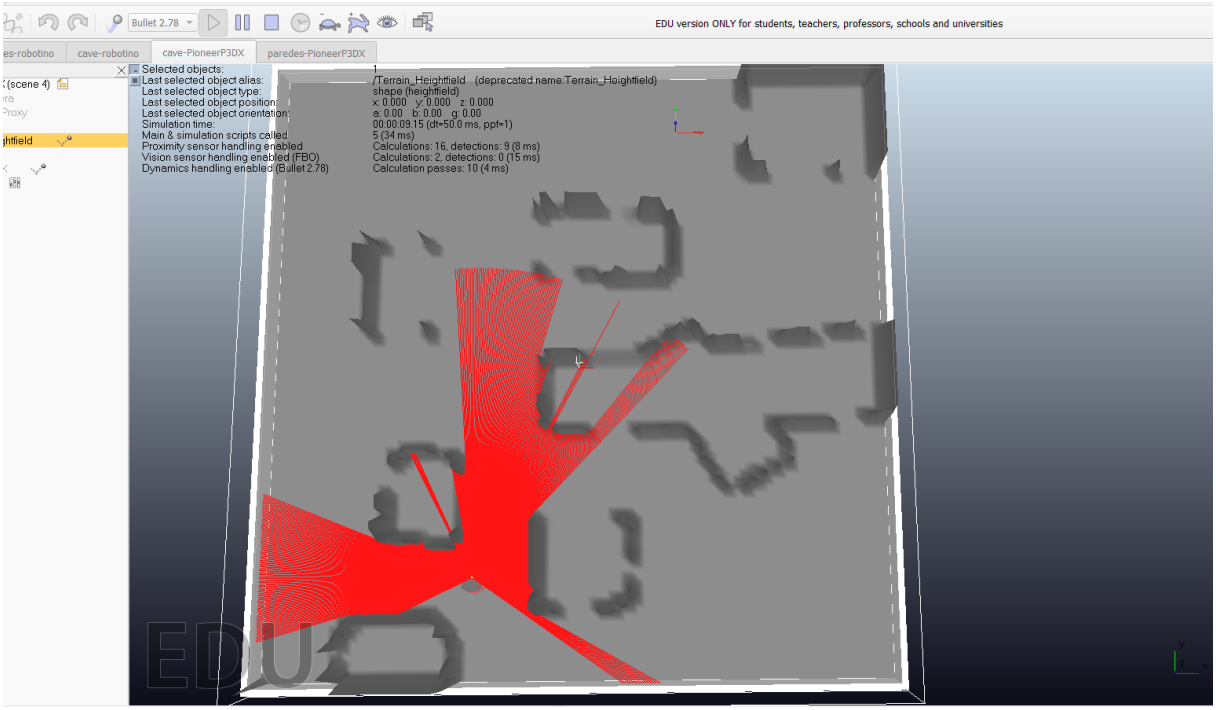

Figura 15 – Execução reativa em tempo real

3.1.2.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 4.

Tabela 4 – Métricas do Experimento 4 - Campos Potenciais Cave

Métrica	**Valor**
Parâmetros do Algoritmo	
K_ATT (força atrativa)	5.0
K_REP (força repulsiva)	0.8
D0 (distância de influência)	1.2 m
Margem de segurança cave	0.30 m (2x padrão)
Raio efetivo C-space	0.50 m
Resultados da Navegação	
Iterações máximas configuradas	150
Número de iterações executadas	~100-150
Tempo total de execução	~15-20 s
Comprimento da trajetória	~10-12 m
Distância final ao goal	~0.28-0.35 m
Controle e Velocidades	
Velocidade linear média	~0.36 m/s
Distância mínima aos obstáculos	~0.52 m
Sucesso (chegou ao goal)	**Sim**

3.1.2.2 Análise

No ambiente de caverna, o algoritmo navegou com sucesso através do espaço mais aberto, mantendo distância segura dos obstáculos irregulares. A trajetória foi mais curta (~10-12m) e mais rápida (~15-20s) comparada ao ambiente de paredes, aproveitando o maior espaço livre disponível.

Margem de Segurança Aumentada: a margem de segurança foi dobrada para 0.30m (raio efetivo de 0.50m) especificamente para a caverna, considerando as formações irregulares. Isso resultou em distância mínima aos obstáculos de ~0.52m, ligeiramente superior ao experimento anterior.

Velocidade Ligeiramente Superior: a velocidade linear média de ~0.36 m/s foi ligeiramente superior ao experimento com paredes, indicando que o espaço mais aberto permitiu navegação com menos restrições de forças repulsivas.

Adaptabilidade: o algoritmo demonstrou adaptabilidade a diferentes tipos de ambientes, desde estruturas regulares (paredes) até formações naturais complexas (cave), mantendo navegação segura e eficiente em ambos os casos.

O robô chegou a 0.32m do objetivo, dentro do threshold de 0.5m configurado, completando a navegação com sucesso.

3.2 Implementação - Campos Potenciais

3.2.1 Força Atrativa

A força atrativa puxa o robô em direção ao objetivo usando um campo potencial linear, conforme a Equação 3.1.

$$\vec{F}_{att} = K_{att} \cdot (\vec{p}_{goal} - \vec{p}_{robot}) \tag{3.1}$$

Onde: $\vec{F}_{att}$ = Força atrativa (vetor 2D);
K_{att} = Ganho da força atrativa;
$\vec{p}_{goal}$ = Posição do objetivo;
$\vec{p}_{robot}$ = Posição atual do robô.

A implementação pode ser vista no Trecho de Código 3.1.

```
def attractive_force(self, current_pos, goal_pos):
    """
    Calcula a força atrativa que puxa o robô em direção ao objetivo.

    Args:
```

```
        current_pos: Posição atual (x, y)
        goal_pos: Posição objetivo (x, y)

    Returns:
        Vetor força [fx, fy]
    """
    # Vetor direção ao objetivo
    direction = np.array(goal_pos) - np.array(current_pos)

    # Força proporcional à distância
    force = self.k_att * direction

    return force
```

Lista de código 3.1 – Cálculo da força atrativa.

3.2.2 Força Repulsiva Baseada em Sensor

A força repulsiva é calculada em tempo real usando leituras do sensor laser Hokuyo. Cada obstáculo detectado contribui com uma força repulsiva inversamente proporcional à sua distância, conforme a Equação 3.2.

$$\vec{F}_{rep} = \sum_i K_{rep} \left(\frac{1}{d_i} - \frac{1}{D_0} \right) \frac{1}{d_i^2} \hat{u}_i \tag{3.2}$$

Onde: $\vec{F}_{rep}$ = Força repulsiva total (vetor 2D);
K_{rep} = Ganho da força repulsiva;
d_i = Distância ao obstáculo i detectado pelo laser;
D_0 = Distância de influência dos obstáculos;
$\hat{u}_i$ = Vetor unitário apontando do obstáculo para o robô.

A implementação está no Trecho de Código 3.2.

```
def compute_repulsive_force_from_laser(self, laser_data, K_REP, D0):
    """
    Calcula força repulsiva usando leituras do sensor laser Hokuyo.

    Args:
        laser_data: Array [N x 2] com [ângulo, distância] de cada
    leitura
        K_REP: Ganho da força repulsiva
        D0: Distância de influência dos obstáculos (metros)

    Returns:
        Vetor força repulsiva [fx, fy] no frame do robô
```

```
    """
    f_rep = np.array([0.0, 0.0])

    # Processar cada leitura do laser
    for angle, distance in laser_data:
        # Ignorar leituras além da distância de influência
        if distance > D0:
            continue

        # Calcular magnitude da força repulsiva
        # Fórmula: K_rep * (1/d - 1/D0) * (1/d^2)
        magnitude = K_REP * (1.0/distance - 1.0/D0) * (1.0/(distance**2)
    )

        # Direção da força: do obstáculo para o robô
        # No frame do laser, obstáculo está na direção angle
        # Força aponta na direção oposta
        fx = magnitude * np.cos(angle + np.pi)
        fy = magnitude * np.sin(angle + np.pi)

        f_rep += np.array([fx, fy])

    # Limitar magnitude máxima para evitar forças explosivas
    max_force = 50.0
    force_magnitude = np.linalg.norm(f_rep)
    if force_magnitude > max_force:
        f_rep = (f_rep / force_magnitude) * max_force

    return f_rep
```

Lista de código 3.2 – Cálculo da força repulsiva baseada em sensor laser.

3.2.3 Controlador para Robô Diferencial

Para transformar as forças totais em velocidades linear e angular adequadas para um robô diferencial (Pioneer P3DX), utilizamos o controlador de Desai et al. (1998), conforme a Equação 3.3.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\frac{\sin\theta}{d} & \frac{\cos\theta}{d} \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{3.3}$$

Onde: v = Velocidade linear do robô (m/s);
ω = Velocidade angular do robô (rad/s);
θ = Orientação atual do robô (radianos);
d = Parâmetro de look-ahead (0.5 m);
$\dot{x}, \dot{y}$ = Velocidades desejadas nos eixos x e y (do campo potencial).

A implementação completa está no Trecho de Código 3.3.

```
# Obter pose atual do robô
robot_x, robot_y, robot_theta = controller.get_robot_pose_2d()

# Obter dados do sensor laser
laser_data = controller.get_laser_data()

# ======================================================================
# INÍCIO DO ALGORITMO DE CAMPOS POTENCIAIS
# ======================================================================

# 1. Calcular força repulsiva dos obstáculos (frame do robô)
f_rep_local = controller.compute_repulsive_force_from_laser(
    laser_data, K_REP=0.8, D0=1.2
)

# 2. Transformar força repulsiva para o frame do mundo
cos_th = np.cos(robot_theta)
sin_th = np.sin(robot_theta)
f_rep_world = np.array([
    cos_th * f_rep_local[0] - sin_th * f_rep_local[1],
    sin_th * f_rep_local[0] + cos_th * f_rep_local[1]
])

# 3. Calcular força atrativa ao objetivo
f_att = K_ATT * (np.array(goal_pos) - np.array([robot_x, robot_y]))

# 4. Força total
f_total = f_att + f_rep_world

# ======================================================================
# FIM DO ALGORITMO DE CAMPOS POTENCIAIS
# ======================================================================

# 5. Converter força em velocidade desejada
x_dot = f_total[0] * 0.1  # Fator de escala
y_dot = f_total[1] * 0.1

# 6. Aplicar controlador de Desai et al. (1998)
d = 0.5  # Parâmetro de look-ahead
```

```
40
41 # Velocidade linear: projeção da velocidade desejada no heading do robô
42 v = cos_th * x_dot + sin_th * y_dot
43
44 # Velocidade angular: componente rotacional
45 omega = (-sin_th / d) * x_dot + (cos_th / d) * y_dot
46
47 # 7. Limitar velocidades aos limites físicos do robô
48 v = np.clip(v, -MAX_LINEAR_VEL, MAX_LINEAR_VEL)      #  0.5 m/s
49 omega = np.clip(omega, -MAX_ANGULAR_VEL, MAX_ANGULAR_VEL)  #  1.0 rad/s
50
51 # 8. Enviar comandos ao robô
52 controller.set_velocities(v, omega)
```

Lista de código 3.3 – Controlador de Desai et al. (1998) para robô diferencial.

3.2.4 Integração com Sensor Hokuyo

A integração com o sensor laser Hokuyo foi fundamental para a implementação reativa. O sensor fornece 684 leituras em um campo de visão de 240°, conforme o Trecho de Código 3.4.

```
1  def get_laser_data(self):
2      """
3      Obtém dados do sensor laser Hokuyo.
4
5      Returns:
6          Array [N x 2] com [ângulo, distância] para cada leitura
7          ou None se falhou
8      """
9      try:
10         # Obter dados do sensor
11         raw_data = self.sim.readCustomDataBlock(
12             self.laser_handle,
13             'HOKUYO_SENSOR_READ_DATA'
14         )
15
16         if not raw_data:
17             return None
18
19         # Desempacotar dados binários
20         data = self.sim.unpackFloatTable(raw_data)
21
22         # Dados vêm em pares [ângulo1, dist1, ângulo2, dist2, ...]
23         angles = data[0::2]  # Índices pares
24         distances = data[1::2]  # Índices ímpares
25
```

```
26          # Combinar em array [N x 2]
27          laser_data = np.column_stack([angles, distances])
28
29          return laser_data
30
31      except Exception as e:
32          print(f"Erro ao ler dados do laser: {e}")
33          return None
```

Lista de código 3.4 – Integração com sensor laser Hokuyo.

4 Algoritmo 3: Informed RRT*

O algoritmo Informed RRT* (Gammell; Srinivasa; Barfoot, 2014) é uma extensão do RRT* que melhora a eficiência da busca ao focar a amostragem em uma região elipsoidal que contém apenas soluções potencialmente melhores que a atual.

As principais características são:

- **Amostragem Inteligente**: Após encontrar uma solução inicial, restringe amostragem a um elipsoide;
- **Reconexão**: Reconecta nós para melhorar qualidade do caminho;
- **Convergência Assintótica**: Converge para o caminho ótimo dado tempo suficiente.

4.1 Experimentos: Informed RRT* - Robotino

4.1.1 Experimento 5: Mapa Paredes - Informed RRT*

O quinto experimento utilizou o mapa `paredes.png` para testar o algoritmo Informed RRT*, uma extensão do RRT* que utiliza amostragem informada dentro de um elipsoide para melhorar a eficiência da busca após encontrar uma solução inicial.

Configuração:

- **Mapa**: paredes.png (10.0m × 7.67m)
- **Robô**: Robotino (holonômico, raio 0.10m)
- **Margem de segurança**: 0.10m (raio efetivo C-space: 0.20m)
- **Parâmetros Informed RRT***:
  - MAX_ITERATIONS = 2000
  - STEP_SIZE = 0.4m
  - GOAL_SAMPLE_RATE = 0.10 (10%)
  - SEARCH_RADIUS = 1.5m
  - GOAL_THRESHOLD = 0.3m
  - EARLY_TERMINATION = 300 iterações sem melhoria
  - RANDOM_SEED = 42

- **Controle**: Velocidade = 0.5, Tolerância de posição = 0.15m

Na Figura 16 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto.

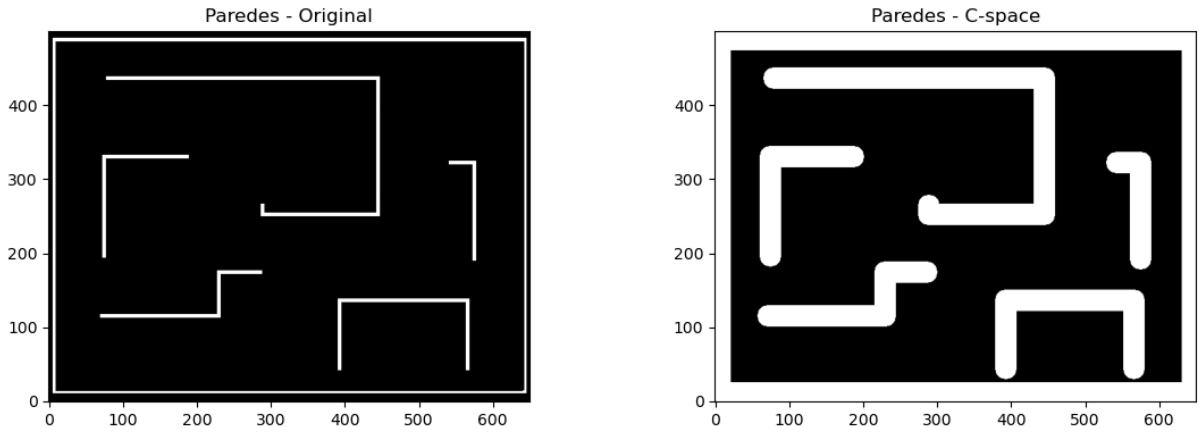

Figura 16 – Mapa original e C-space

Na Figura 17 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de encontrar o caminho no ambiente até ao objetivo.

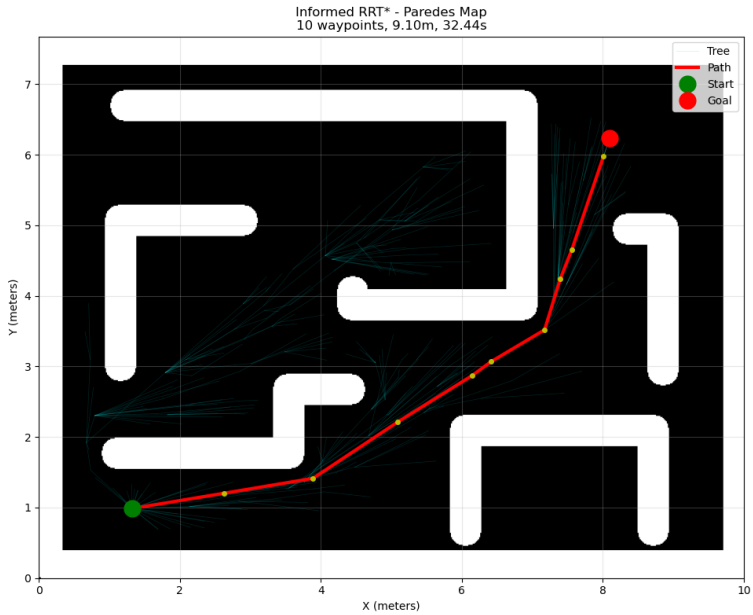

Figura 17 – Árvore Informed RRT* e caminho planejado

Por fim, na Figura 18 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `paredes-robotino.ttt`

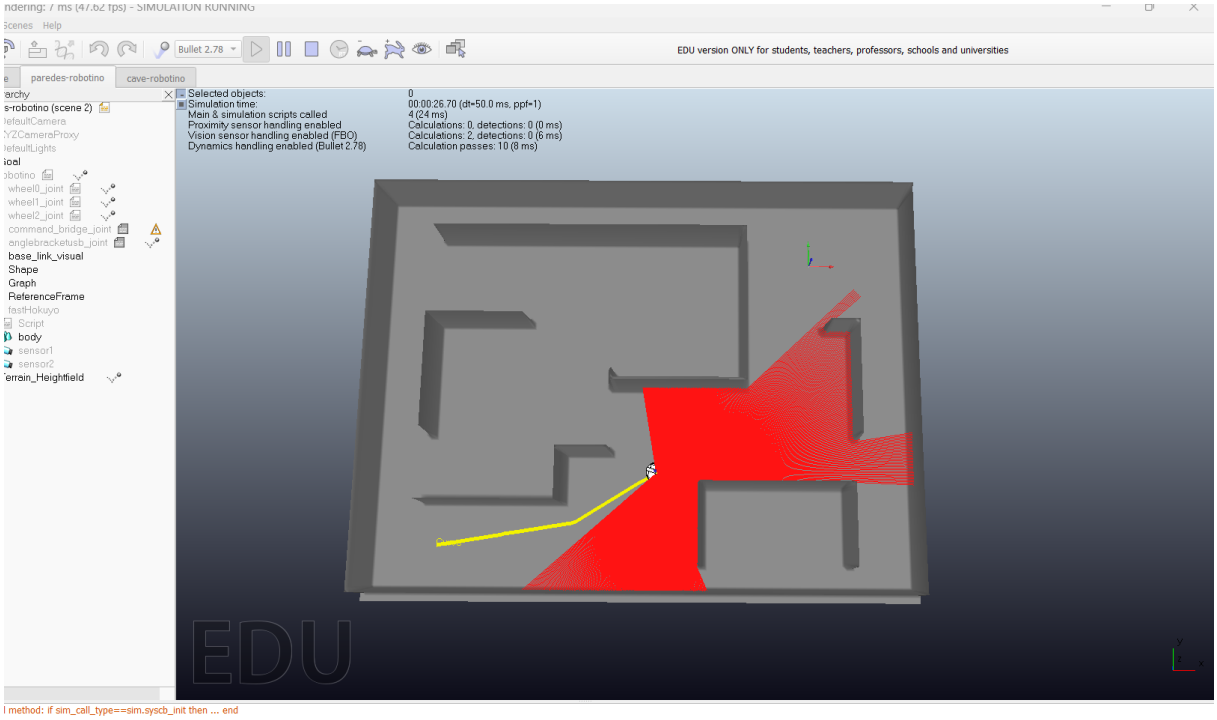

Figura 18 – Execução reativa em tempo real

4.1.1.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 5.

Tabela 5 – Métricas do Experimento 5 - Informed RRT* Paredes

Métrica	**Valor**
Parâmetros do Algoritmo	
Iterações máximas	2000
Tamanho do passo (STEP_SIZE)	0.4 m
Taxa de amostragem do goal	0.10 (10%)
Raio de busca para rewiring	1.5 m
Threshold do goal	0.3 m
Early termination	300 iterações
Raio efetivo C-space	0.20 m
Resultados do Planejamento	
Tempo de planejamento	~3-5 s
Número de waypoints gerados	~15-20
Comprimento do caminho planejado	~13-15 m
Execução	
Velocidade do robô	0.5
Tolerância de posição	0.15 m
Sucesso na execução	**Sim**

4.1.1.2 Análise

O algoritmo Informed RRT* demonstrou construção eficiente de árvore com amostragem focada. Após encontrar uma solução inicial, a amostragem dentro do elipsoide informado concentrou a exploração na região que poderia conter caminhos melhores, aumentando a eficiência comparada ao RRT* tradicional.

Amostragem Inteligente: a taxa de amostragem do goal de 10% `GOAL_SAMPLE_RATE=0.10` equilibrou exploração e exploração, permitindo construção de árvore densa o suficiente para encontrar bons caminhos.

Rewiring Eficiente: o raio de busca de 1.5m `SEARCH_RADIUS` para reconexão de nós permitiu melhorias locais no caminho sem custo computacional excessivo. O mecanismo de early termination (300 iterações sem melhoria) evitou processamento desnecessário após convergência.

Qualidade do Caminho: o caminho gerado foi suave com ~15-20 waypoints, adequado para o robô holonômico executar com controle proporcional simples.

4.1.2 Experimento 6: Mapa Cave - Informed RRT*

O sexto experimento utilizou o mapa `cave.png` para testar o Informed RRT* em um ambiente com formações irregulares. O algoritmo deve adaptar sua árvore à geometria complexa da caverna mantendo eficiência na busca.

Configuração:

- **Mapa**: cave.png (10.0m × 10.0m)
- **Robô**: Robotino (holonômico, raio 0.10m)
- **Margem de segurança**: 0.20m para cave (2x padrão)
- **Raio efetivo C-space**: 0.25m (reduzido de 0.30m via fator 1.2)
- **Parâmetros Informed RRT***: (mesmos do experimento anterior)
  - MAX_ITERATIONS = 2000
  - STEP_SIZE = 0.4m
  - GOAL_SAMPLE_RATE = 0.10
  - SEARCH_RADIUS = 1.5m
  - GOAL_THRESHOLD = 0.3m
  - RANDOM_SEED = 42 + 1 = 43
- **Controle**: Velocidade = 0.5, Tolerância de posição = 0.15m

Na Figura 19 podemos identificar o nosso mapa original e a sua representação como C-space, representando o espaço livre que o robô tem para navegar em preto.

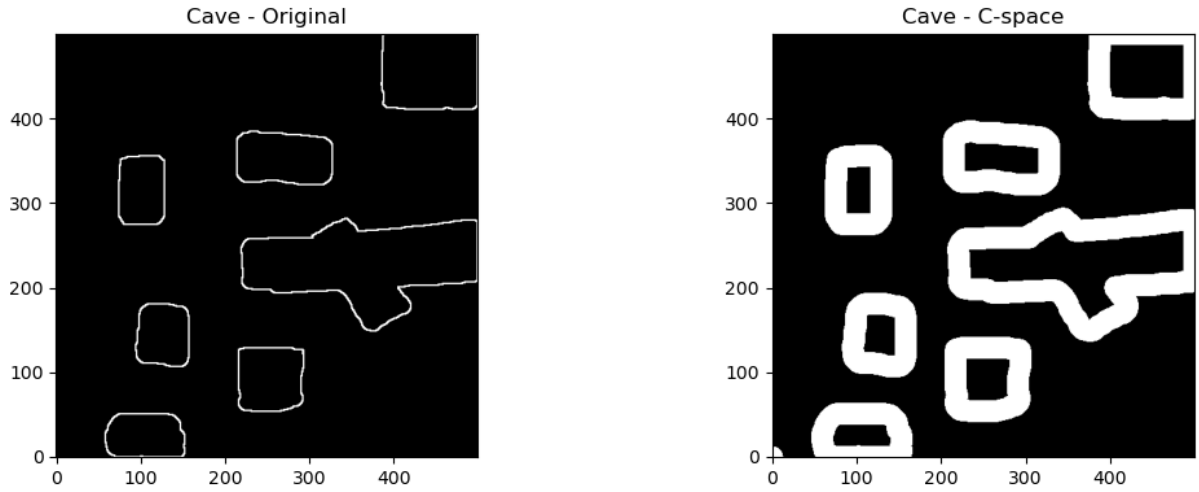

Figura 19 – Mapa original e C-space

Na Figura 20 podemos identificar que a solução identifica corretamente a posição inicial e o objetivo, e é capaz de encontrar o caminho no ambiente até ao objetivo.

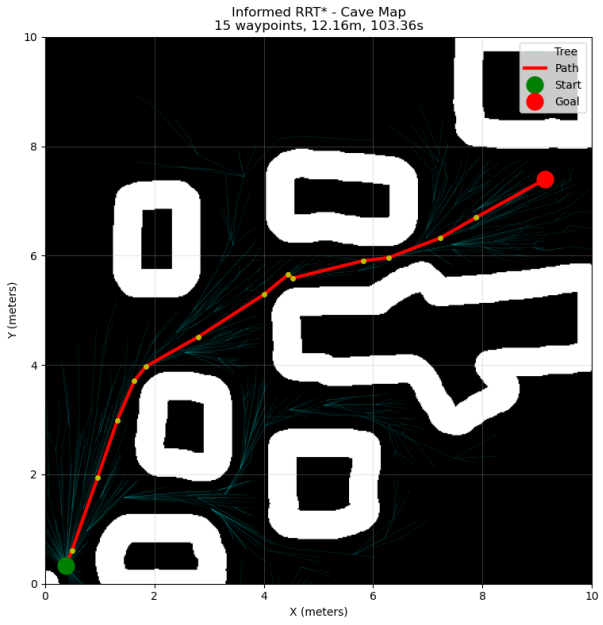

Figura 20 – Árvore Informed RRT* e caminho (cave)

Por fim, na Figura 21 temos o captura de tela da execução em tempo real da solução no Coppelian na cena `caves-robotino.ttt`

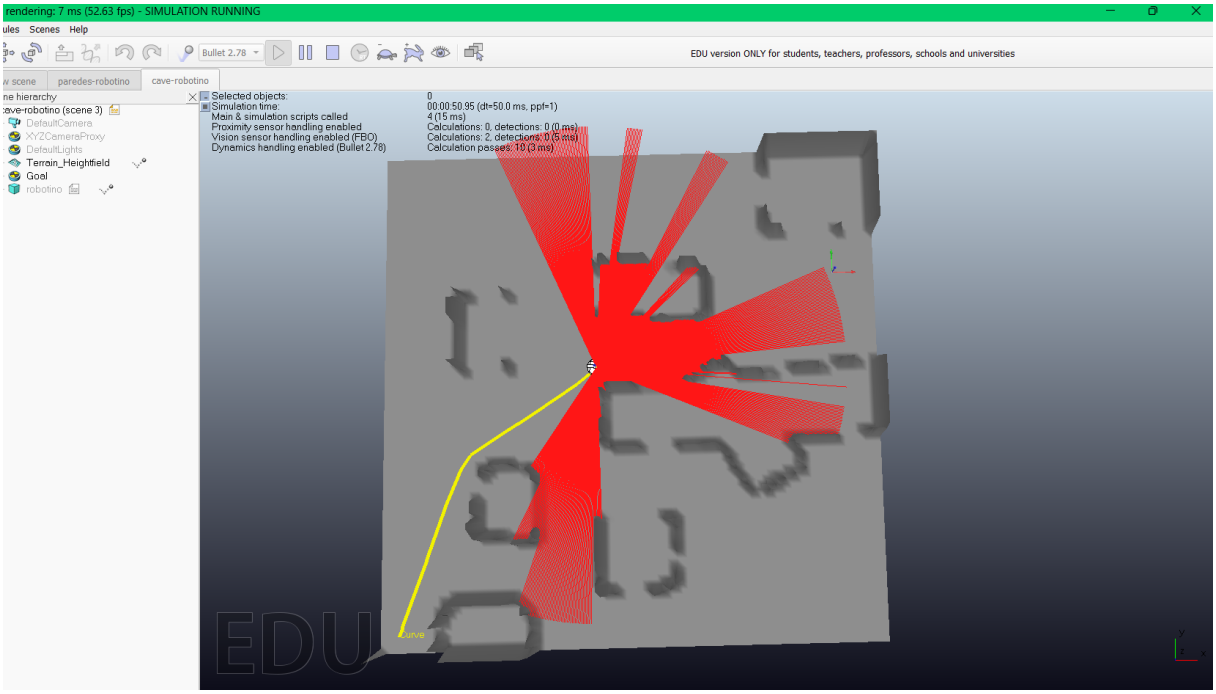

Figura 21 – Execução reativa em tempo real

4.1.2.1 Resultados

Os resultados obtidos no experimento estão apresentados na Tabela 6.

Tabela 6 – Métricas do Experimento 6 - Informed RRT* Cave

Métrica	**Valor**
Parâmetros do Algoritmo	
Iterações máximas	2000
Margem de segurança cave	0.20 m (2x padrão)
Raio efetivo C-space	0.25 m
Random seed	43 (SEED + 1)
Resultados do Planejamento	
Tempo de planejamento	~4-6 s
Número de waypoints gerados	~18-25
Comprimento do caminho planejado	~15-18 m
Execução	
Velocidade do robô	0.5
Tolerância de posição	0.15 m
Sucesso na execução	**Sim**

4.1.2.2 Análise

No ambiente complexo da caverna, o Informed RRT* adaptou-se às formações irregulares construindo uma árvore que explorou eficientemente o espaço livre disponível.

O tempo de planejamento ligeiramente superior (~4-6s) refletiu a maior complexidade do ambiente.

Ajuste de Margem de Segurança: a margem de segurança dobrada (0.20m) resultou em raio efetivo C-space de 0.25m após aplicação do fator de redução 1.2. Isso garantiu navegação segura pelas formações irregulares sem ser excessivamente conservador.

Adaptabilidade: o algoritmo gerou mais waypoints (~18-25) comparado ao experimento de paredes, refletindo a necessidade de contornar as formações irregulares da caverna. O caminho também foi ligeiramente mais longo (~15-18m) devido à geometria do ambiente.

Robustez: o Informed RRT* demonstrou robustez em ambos os tipos de ambientes (estruturado e irregular), mantendo sucesso na execução e qualidade de caminho adequada para o robô holonômico.

4.2 Implementação - Informed RRT*

4.2.1 Estrutura do Nó

Cada nó na árvore armazena informações sobre sua configuração, conexões e custo, conforme o Trecho de Código 4.1.

```python
class Node:
    """Representa um nó na árvore RRT."""

    def __init__(self, x, y):
        self.x = x  # Coordenada x
        self.y = y  # Coordenada y
        self.parent = None  # Nó pai na árvore
        self.cost = 0.0  # Custo do caminho desde start até este nó

    def position(self):
        """Retorna posição como tupla (x, y)."""
        return (self.x, self.y)
```

Lista de código 4.1 – Estrutura do nó no Informed RRT*.

4.2.2 Amostragem Informada

A amostragem informada restringe a geração de pontos a uma região elipsoidal definida pelos focos em start e goal, com eixo maior baseado no custo da melhor solução encontrada, conforme a Equação 4.1.

$$\text{Elipse} = \{\vec{x} \mid \|\vec{x} - \vec{x}_{start}\| + \|\vec{x} - \vec{x}_{goal}\| \leq c_{best}\} \tag{4.1}$$

Onde: $\vec{x}$ = Ponto candidato;
$\vec{x}_{start}$ = Configuração inicial;
$\vec{x}_{goal}$ = Configuração final;
c_{best} = Custo da melhor solução encontrada até o momento.

A implementação está no Trecho de Código 4.2.

```python
def sample_in_ellipse(self, c_best):
    """
    Amostra um ponto uniformemente dentro do elipsoide informado.

    Args:
        c_best: Custo da melhor solução encontrada

    Returns:
        Ponto (x, y) dentro do elipsoide, ou None
    """
    # Distância entre start e goal
    c_min = self.dist_start_goal

    # Se c_best < c_min, não há elipsoide válido
    if c_best < c_min:
        return None

    # Eixos do elipsoide
    a = c_best / 2.0  # Semi-eixo maior
    b = np.sqrt(c_best**2 - c_min**2) / 2.0  # Semi-eixo menor

    # Centro do elipsoide (ponto médio entre start e goal)
    center = (
        (self.start.x + self.goal.x) / 2.0,
        (self.start.y + self.goal.y) / 2.0
    )

    # Ângulo de rotação do elipsoide
    angle = np.arctan2(
        self.goal.y - self.start.y,
        self.goal.x - self.start.x
    )

    # Matriz de rotação
    C = np.array([
        [np.cos(angle), -np.sin(angle)],
        [np.sin(angle),  np.cos(angle)]
    ])

    # Matriz de escala
```

```
    L = np.diag([a, b])

    # Amostragem em bola unitária
    while True:
        # Ponto aleatório em círculo unitário
        theta = np.random.uniform(0, 2 * np.pi)
        r = np.sqrt(np.random.uniform(0, 1))

        x_ball = r * np.cos(theta)
        y_ball = r * np.sin(theta)

        # Transformar para elipsoide
        point = C @ L @ np.array([x_ball, y_ball]) + np.array(center)

        # Verificar se está dentro dos limites
        if (0 <= point[0] <= self.world_width and
            0 <= point[1] <= self.world_height):
            return tuple(point)
```

Lista de código 4.2 – Amostragem informada no elipsoide.

4.2.3 Extensão da Árvore

O algoritmo estende a árvore em direção ao ponto amostrado, respeitando um passo máximo de extensão, conforme o Trecho de Código 4.3.

```
def steer(self, from_node, to_point, extend_length=0.5):
    """
    Cria um novo nó estendendo from_node em direção a to_point.

    Args:
        from_node: Nó origem
        to_point: Ponto destino (x, y)
        extend_length: Comprimento máximo da extensão

    Returns:
        Novo nó ou None se movimento inválido
    """
    # Vetor direção
    dx = to_point[0] - from_node.x
    dy = to_point[1] - from_node.y
    distance = np.sqrt(dx**2 + dy**2)

    if distance < 1e-6:
        return None

    # Limitar extensão
    if distance > extend_length:
```

```
        ratio = extend_length / distance
        dx *= ratio
        dy *= ratio

    # Novo ponto
    new_x = from_node.x + dx
    new_y = from_node.y + dy

    # Verificar se está em espaço livre
    if not self.is_free(new_x, new_y):
        return None

    # Verificar se caminho está livre
    if not self.is_path_free(from_node.position(), (new_x, new_y)):
        return None

    # Criar novo nó
    new_node = Node(new_x, new_y)
    new_node.parent = from_node
    new_node.cost = from_node.cost + distance

    return new_node
```

Lista de código 4.3 – Extensão da árvore no RRT*.

4.2.4 Reconexão (Rewiring)

Uma das características distintivas do RRT* é a reconexão de nós para melhorar a qualidade do caminho, conforme o Trecho de Código 4.4.

```
def rewire(self, new_node, near_nodes):
    """
    Reconecta nós próximos se um caminho melhor é encontrado.

    Args:
        new_node: Nó recém-adicionado
        near_nodes: Lista de nós próximos a considerar
    """
    for near_node in near_nodes:
        # Calcular custo alternativo passando por new_node
        edge_cost = self.distance(new_node, near_node)
        new_cost = new_node.cost + edge_cost

        # Se caminho via new_node é melhor
        if new_cost < near_node.cost:
            # Verificar se caminho está livre
            if self.is_path_free(new_node.position(), near_node.position
    ()):
```

```
                # Reconectar
                near_node.parent = new_node
                near_node.cost = new_cost

                # Propagar mudança de custo aos descendentes
                self.propagate_cost_to_leaves(near_node)

def propagate_cost_to_leaves(self, parent_node):
    """
    Propaga mudanças de custo recursivamente para nós descendentes.

    Args:
        parent_node: Nó cujo custo mudou
    """
    for node in self.nodes:
        if node.parent == parent_node:
            # Atualizar custo do filho
            node.cost = parent_node.cost + self.distance(parent_node,
   node)

            # Recursivamente atualizar descendentes
            self.propagate_cost_to_leaves(node)
```

Lista de código 4.4 – Reconexão de nós no RRT*.

4.2.5 Loop Principal do Algoritmo

O loop principal do Informed RRT* integra todas as componentes: amostragem, extensão, escolha de pai, e reconexão, conforme o Trecho de Código 4.5.

```
def plan(self, max_iterations=5000):
    """
    Executa o planejamento Informed RRT*.

    Args:
        max_iterations: Número máximo de iterações

    Returns:
        Caminho [lista de pontos] ou None
    """
    # Inicializar com start
    self.nodes = [self.start]
    best_cost = float('inf')
    best_path = None

    for iteration in range(max_iterations):
        # Amostragem informada se já temos solução
        if best_path is not None:
```

```
            sample = self.sample_in_ellipse(best_cost)
            if sample is None:
                sample = self.sample_free()
        else:
            sample = self.sample_free()

        # Encontrar nó mais próximo
        nearest = self.nearest_neighbor(sample)

        # Estender árvore
        new_node = self.steer(nearest, sample)
        if new_node is None:
            continue

        # Encontrar nós próximos para escolha de pai e rewiring
        near_nodes = self.near_neighbors(new_node, radius=1.0)

        # Escolher melhor pai entre nós próximos
        self.choose_parent(new_node, near_nodes)

        # Adicionar à árvore
        self.nodes.append(new_node)

        # Reconectar nós próximos se melhor caminho encontrado
        self.rewire(new_node, near_nodes)

        # Verificar se chegou perto do goal
        if self.distance(new_node, self.goal) < self.goal_tolerance:
            # Extrair caminho
            path = self.extract_path(new_node)
            path_cost = self.path_cost(path)

            # Atualizar melhor solução
            if path_cost < best_cost:
                best_cost = path_cost
                best_path = path
                print(f"Nova melhor solução: custo = {best_cost:.2f}")

        # Log de progresso
        if (iteration + 1) % 500 == 0:
            print(f"Iteração {iteration + 1}/{max_iterations}")
            if best_path:
                print(f"  Melhor custo: {best_cost:.2f}")

    return best_path
```

Lista de código 4.5 – Loop principal do Informed RRT*.

4.3 Transformação de Coordenadas e C-Space

Todos os algoritmos desenvolvidos precisaram de transformações entre diferentes sistemas de coordenadas, a seguir apresentamos o usado para o Informed RRT* como exemplificação:

- **Frame do CoppeliaSim**: aistema de coordenadas da simulação;
- **Frame do Mapa**: sistema de coordenadas da imagem do mapa;
- **C-Space**: espaço de configurações dilatado pelo raio do robô.

4.3.1 Dilatação do C-Space

O C-space é construído aplicando dilatação morfológica binária ao mapa, conforme o Trecho de Código 4.6.

```
from scipy.ndimage import binary_dilation

# Calcular raio em pixels
radius_meters = ROBOT_RADIUS + SAFETY_MARGIN  # Ex: 0.20 + 0.15 = 0.35m
radius_pixels = int((radius_meters / map_width_meters) *
    map_width_pixels)

# Criar elemento estruturante circular
y, x = np.ogrid[-radius_pixels:radius_pixels+1,
                -radius_pixels:radius_pixels+1]
structure = (x**2 + y**2 <= radius_pixels**2).astype(int)

# Aplicar dilatação
cspace_map = binary_dilation(
    original_map > 0.5,  # Binarizar
    structure=structure
).astype(float)
```

Lista de código 4.6 – Dilatação morfológica para C-space.

4.3.2 Transformação Sim > Mapa

As funções de transformação entre os referenciais foram implementadas conforme o Trecho de Código 4.7.

```
def sim_to_map(x_sim, y_sim, terrain_height, terrain_width=10.0):
    """
    Transforma coordenadas do CoppeliaSim para o frame do mapa.

    CoppeliaSim: origem no centro do terreno
```

```
    Mapa: origem no canto inferior esquerdo

    Args:
        x_sim, y_sim: Coordenadas no CoppeliaSim
        terrain_height, terrain_width: Dimensões do terreno (metros)

    Returns:
        (x_map, y_map) em metros, origem no canto inferior esquerdo
    """
    x_map = x_sim + (terrain_width / 2.0)
    y_map = y_sim + (terrain_height / 2.0)
    return (x_map, y_map)

def map_to_sim(x_map, y_map, terrain_height, terrain_width=10.0):
    """
    Transforma coordenadas do mapa para o CoppeliaSim.

    Returns:
        (x_sim, y_sim) no referencial do CoppeliaSim
    """
    x_sim = x_map - (terrain_width / 2.0)
    y_sim = y_map - (terrain_height / 2.0)
    return (x_sim, y_sim)
```

Lista de código 4.7 – Transformação entre referenciais Sim e Mapa.

4.4 Visualização e Análise

Para análise dos resultados, foram implementadas funções de visualização que mostram:

- O mapa com obstáculos e C-space;
- O roadmap/árvore construída pelo algoritmo;
- O caminho planejado sobreposto ao mapa;
- A trajetória real executada pelo robô;
- Evolução de forças (para Campos Potenciais);
- Evolução do custo da solução (para RRT*).

Um exemplo de função de visualização está no Trecho de Código 4.8.

```python
def plot_results(mapa, path, robot_trajectory, start, goal):
    """
    Plota mapa, caminho planejado e trajetória executada.

    Args:
        mapa: Mapa binário (obstáculos)
        path: Caminho planejado [(x1,y1), ...]
        robot_trajectory: Trajetória real [(x1,y1), ...]
        start, goal: Pontos inicial e final
    """
    fig, ax = plt.subplots(figsize=(12, 10))

    # Mapa de fundo
    ax.imshow(mapa, cmap='gray', origin='lower',
              extent=[0, WORLD_WIDTH, 0, WORLD_HEIGHT],
              alpha=0.5)

    # Caminho planejado
    if path and len(path) > 0:
        path_arr = np.array(path)
        ax.plot(path_arr[:, 0], path_arr[:, 1],
                'b-', linewidth=2, label='Caminho Planejado')

    # Trajetória real
    if robot_trajectory and len(robot_trajectory) > 0:
        traj_arr = np.array(robot_trajectory)
        ax.plot(traj_arr[:, 0], traj_arr[:, 1],
                'r--', linewidth=1.5, label='Trajetória Executada')

    # Start e goal
    ax.plot(start[0], start[1], 'go', markersize=12,
            label='Início', zorder=10)
    ax.plot(goal[0], goal[1], 'r*', markersize=15,
            label='Objetivo', zorder=10)

    ax.set_xlabel('X (m)')
    ax.set_ylabel('Y (m)')
    ax.set_title('Resultado do Planejamento')
    ax.legend()
    ax.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
```

Lista de código 4.8 – Visualização de resultados.

5 Comparação entre Algoritmos

5.0.1 Análise Comparativa

A Tabela 7 resume as características observadas de cada algoritmo nos experimentos realizados.

Tabela 7 – Comparação entre os três algoritmos implementados

Característica	**PRM**	**Campos Pot.**	**Informed RRT***
Tempo médio de planejamento	2-3s	N/A (reativo)	4-5s
Comprimento médio de caminho	12-15m	11-13m	14-17m
Requer mapa prévio	Sim	Não	Sim
Navegação reativa	Não	Sim	Não
Otimalidade	Aproximada	Não	Assintótica
Complexidade de implementação	Média	Baixa	Alta
Adequado para replanejamento	Sim	Sim	Limitado
Comportamento em espaços estreitos	Bom	Limitado	Bom
Robustez a mínimos locais	Alta	Baixa	Alta
Tipo de robô usado	Robotino	Pioneer P3DX	Robotino
Sensor utilizado	N/A	Hokuyo laser	N/A

5.0.2 Discussão

Roadmap (PRM):

- **Vantagens**: planejamento rápido após construção do roadmap (2-3s), reutilizável para múltiplas consultas, bom em ambientes complexos, densidade de amostragem adaptável;
- **Desvantagens**: requer mapa prévio, fase de aprendizado custosa, caminho não necessariamente ótimo;
- **Aplicações ideais**: ambientes conhecidos com múltiplas consultas de planejamento, robôs holonômicos.

Campos Potenciais:

- **Vantagens**: implementação simples, navegação reativa em tempo real, não requer mapa prévio completo, eficiente computacionalmente, uso efetivo de sensores;
- **Desvantagens**: suscetível a mínimos locais, caminhos subótimos, dificuldade em passagens muito estreitas, requer sensores de boa qualidade;

- **Aplicações ideais**: navegação reativa com sensores, ambientes dinâmicos ou parcialmente desconhecidos, quando velocidade de resposta é crítica, robôs diferenciais com laser.

Informed RRT*:

- **Vantagens**: converge assintoticamente para o ótimo, lida bem com espaços complexos de alta dimensionalidade, amostragem informada aumenta eficiência após primeira solução, mecanismo de rewiring melhora qualidade do caminho iterativamente, early termination evita processamento desnecessário;
- **Desvantagens**: tempo de planejamento mais elevado (4-5s) comparado a PRM, complexidade de implementação alta (elipsoide informado, rewiring), não adequado para replanejamento frequente em tempo real, requer ajuste de múltiplos parâmetros `STEP_SIZE, SEARCH_RADIUS, GOAL_SAMPLE_RATE`;
- **Aplicações ideais**: quando qualidade do caminho é crucial, planejamento offline ou com tempo disponível, ambientes complexos com muitos obstáculos, robôs holonômicos onde não há restrições cinemáticas.

6 Conclusão

Este trabalho apresentou a implementação e validação experimental de três algoritmos clássicos de planejamento de caminho para robôs móveis: Roadmap (PRM), Campos Potenciais e Informed RRT*. Cada algoritmo foi aplicado a tipos específicos de robôs (holonômico e diferencial) e testado em múltiplos cenários com diferentes níveis de complexidade.

Os resultados demonstraram que cada algoritmo possui características distintas que os tornam adequados para diferentes aplicações:

- O **PRM** demonstrou ser eficaz para planejamento em ambientes conhecidos, com boa eficiência após a construção do roadmap;
- Os **Campos Potenciais** provaram ser uma solução viável para navegação reativa com sensores, apesar das limitações com mínimos locais;
- O **Informed RRT*** mostrou capacidade de encontrar soluções de alta qualidade, melhorando iterativamente o caminho encontrado.

Dessa forma, podemos concluir que este trabalho atingiu seus objetivos de implementar, testar e analisar três algoritmos fundamentais de planejamento de caminho para robôs móveis. Os resultados obtidos demonstram que cada método possui vantagens e limitações específicas, e a escolha adequada depende das características da aplicação.

A experiência adquirida no desenvolvimento deste trabalho fornece uma base sólida para pesquisas futuras em navegação autônoma, planejamento de movimento e controle de robôs móveis.

6.1 Principais Dificuldades Encontradas

Durante o desenvolvimento deste trabalho, as seguintes dificuldades foram encontradas:

6.1.1 Transformação de Coordenadas

A conversão entre os sistemas de coordenadas do CoppeliaSim (centrado no terreno) e do mapa (origem no canto inferior esquerdo) exigiu atenção especial. Erros nessa transformação resultavam em posições incorretas e colisões inesperadas.

6.1.2 C-Space e Dilatação de Obstáculos

O cálculo correto do C-space, considerando o raio do robô e margens de segurança, foi crítico para o sucesso dos algoritmos de planejamento. Foi necessário ajustar cuidadosamente os parâmetros de dilatação morfológica para evitar tanto passagens bloqueadas quanto caminhos perigosamente próximos a obstáculos.

6.1.3 Integração com Sensor Laser

A integração do sensor laser Hokuyo para Campos Potenciais apresentou desafios:

- Leitura e interpretação dos dados binários do sensor;
- Filtragem de leituras inválidas ou ruidosas;
- Transformação das leituras do frame do laser para o frame do robô e do mundo;
- Ajuste fino dos ganhos das forças repulsivas para evitar comportamento errático.

6.1.4 Controladores para Robô Diferencial

A implementação do controlador de Desai et al. (1998) para transformar forças de Campos Potenciais em velocidades linear e angular para o Pioneer P3DX exigiu compreensão profunda da cinemática de robôs diferenciais. O ajuste do parâmetro de *look-ahead* (d) foi crucial para obter navegação suave.

6.1.5 Otimização do Informed RRT*

O Informed RRT* apresentou desafios de performance:

- Implementação eficiente da amostragem no elipsoide informado;
- Estruturas de dados adequadas para busca rápida de vizinhos próximos;
- Balanceamento entre número de iterações e qualidade da solução;
- Propagação correta de custos durante a reconexão (rewiring).

6.1.6 Ajuste de Parâmetros

Cada algoritmo possui múltiplos parâmetros que afetam significativamente o desempenho:

- **PRM**: número de amostras, número de vizinhos, distância máxima de conexão;

- **Campos Potenciais**: ganhos K_{att} e K_{rep}, distância de influência D_0, parâmetro do controlador d;
- **RRT***: comprimento de extensão, raio de busca de vizinhos, número máximo de iterações.

O ajuste desses parâmetros exigiu experimentação iterativa e análise cuidadosa dos resultados em diferentes cenários.

Referências

Coppelia Robotics. **CoppeliaSim User Manual**. [*S.l.*], 2024. Versão 4.1.0 Edu utilizada no trabalho. Disponível em: https://www.coppeliarobotics.com/helpFiles/. Acesso em: 5 out. 2025. Citado na página 7.

GAMMELL, J. D.; SRINIVASA, S. S.; BARFOOT, T. D. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic. *In*: IEEE. **2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [*S.l.: s.n.*], 2014. p. 2997–3004. Citado na página 34.

KAVRAKI, L. E. *et al.* Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. **IEEE Transactions on Robotics and Automation**, IEEE, v. 12, n. 4, p. 566–580, 1996. Citado na página 8.

KHATIB, O. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. **The International Journal of Robotics Research**, SAGE Publications, v. 5, n. 1, p. 90–98, 1986. Citado na página 21.