Problema de Caminho no MMS

Grupo Ômega

Miguel Fialho, Luis Paz, Gabriel Pragana,
Luis Tavares e Filipe Godoy

Dezembro de 2025

Abstract

Este artigo explora a implementação de uma simulação de robô para resolução de labirintos. Examinamos o algoritmo *Modified Flood Fill* (Preenchimento por Inundação Modificado) como uma solução de busca de caminho (*pathfinding*), detalhando as configurações de estado e os parâmetros de inicialização necessários para que um *micromouse* navegue com sucesso até o centro de um labirinto.

1 Introdução

1.1 O Problema

O problema é direto e fundamental na robótica móvel: baseia-se em um labirinto desconhecido onde um robô simulado deve navegar autonomamente de uma posição inicial até alcançar o centro.

1.2 A Solução

Para que o robô alcance o centro do labirinto, ele deve seguir um algoritmo determinístico. Para os fins deste artigo, o algoritmo escolhido é o **Flood Fill Modificado**. Este algoritmo define o estado do robô como uma configuração específica composta pelos seguintes componentes:

- As coordenadas do robô na grade (x, y).
- A orientação do robô (Norte, Sul, Leste, Oeste).

O algoritmo inicializa dois conjuntos principais de dados:

1. **Conjunto Objetivo (G):** Representa o destino alvo. Neste caso, são as coordenadas centrais do mapa (representado como uma matriz).
2. **Conjunto Processado (P):** Um conjunto inicialmente vazio que armazena as configurações que já foram processadas, garantindo que o robô não calcule o mesmo estado repetidamente de forma desnecessária.

1.3 Software e Ferramentas

Conforme mencionado no título, utilizaremos o simulador *Micromouse* disponível no repositório GitHub do mackorone. A lógica de controle e navegação será implementada inteiramente na linguagem Python.

2 Codificação e Desenvolvimento

2.1 Apresentação

Esta seção dedica-se a demonstrar e explicar o código desenvolvido para solucionar o problema descrito. O código será dissecado em seções funcionais, detalhadas nas subseções a seguir.

2.2 Configuração Global

```
import API
import sys
import numpy as np
from collections import deque

# CONFIGURACAO GLOBAL
maze_size = 16
# Representacao matricial do mapa (Paredes)
walls = np.zeros((maze_size, maze_size), dtype=int)
# Matriz de Distancias (Flood Fill)
flood = np.full((maze_size, maze_size), -1, dtype=int)

def log(string):
    sys.stderr.write("{}\n".format(string))
    sys.stderr.flush()
```

Listing 1: Main.py

As primeiras linhas de código dedicam-se à importação das bibliotecas essenciais: `API` (a biblioteca de comandos do simulador MMS), `sys` e `numpy`. O **NumPy** é uma biblioteca fundamental aqui, pois permite a manipulação eficiente de matrizes e dados numéricos, otimizando o desempenho do algoritmo.

O labirinto no MMS é representado como uma grade de 16×16. Inicializamos duas variáveis globais críticas, `walls` e `flood`, que atuam como **mapas internos** para o robô executar o algoritmo.

- A matriz `flood` é inicializada especificamente para o Algoritmo *Flood Fill*. Todos os seus elementos começam com o valor -1, indicando que são **células não visitadas** ou com distância desconhecida.
- A matriz `walls` é inicializada com zeros e serve como o mapa "físico" da memória do robô, registrando onde as paredes foram detectadas.

2.3 Função Set Wall (Mapeamento de Paredes)

```
1  def set_wall(x: int, y: int, direction: int):
2      """
3      Marca a parede na celula atual e a parede simetrica na celula
       vizinha.
4      direction: 0=N, 1=E, 2=S, 3=W
5      """
6      # 1. Operacao Bitwise para marcar a celula atual
7      walls[y, x] |= (1 << direction)
8
9      # 2. Calcula as coordenadas do vizinho
10     nx, ny = x, y
11     if direction == 0: ny += 1
12     if direction == 1: nx += 1
13     if direction == 2: ny -= 1
14     if direction == 3: nx -= 1
15
16     # 3. Marca na celula vizinha (se valida e dentro do mapa)
17     if 0 <= nx < maze_size and 0 <= ny < maze_size:
18         # A direcao oposta e calculada via aritmetica modular
19         opposite_dir = (direction + 2) % 4
20         walls[ny, nx] |= (1 << opposite_dir)
```

Listing 2: Main.py

Esta é uma função central para todo o código. Em vez de criar múltiplas variáveis booleanas para identificar onde estão as paredes (Norte, Sul, etc.), utilizamos um **único número inteiro** para cada célula. Para isso, empregamos a codificação binária:

- Norte (0): 2^0 (Binário 0001)
- Leste (1): 2^1 (Binário 0010)
- Sul (2): 2^2 (Binário 0100)
- Oeste (3): 2^3 (Binário 1000)

Para realizar essas combinações, utilizamos o operador **Left Shift** (`1 << direction`), que desloca o bit '1' para a esquerda baseando-se na direção. O outro operador essencial é o **Bitwise OR** (`|=`). Este operador "mescla" os bits, garantindo que adicionemos uma nova parede à memória da célula sem apagar as paredes que já foram registradas anteriormente.

Das linhas 10 a 14, o código assegura a **consistência física** do mapa. É definido que uma parede detectada ao Norte da célula A é, obrigatoriamente, uma parede ao Sul da célula vizinha B. Antes de executar a marcação, o código verifica os limites da grade (`if 0 <= nx < maze_size...`) para prevenir erros de execução (*IndexError*) ao tentar acessar índices fora da matriz 16×16.

Finalmente, para marcar a parede no lado do vizinho, calculamos a **direção recíproca** usando aritmética modular:

$$\texttt{opposite\_dir} = (\texttt{direction} + 2) \pmod 4$$

Esta fórmula rotaciona eficientemente a orientação em 180° (ex: Norte torna-se Sul), garantindo que a barreira seja registrada corretamente em ambos os lados da fronteira.

2.4 Função Flood Update (Planejamento de Caminho)

```
def flood_update():
    """
    Atualiza os valores de distancia na matriz flood.
    """

    # Preenche a matriz com -1 (desconhecido)
    flood.fill(-1)
    # Cria a fila Q para processar as celulas
    queue = deque()

    # Definicao do Conjunto Objetivo G (o centro)
    for r in range(7, 9):
        for c in range(7, 9):
            flood[r, c] = 0
            queue.append((r, c))

    # Mapeamento: (Bit da Parede, Mudanca Linha, Mudanca Coluna)
    path = [(1, 1, 0), (2, 0, 1), (4, -1, 0), (8, 0, -1)]

    # Processa a fila (BFS)
    while queue:
        r, c = queue.popleft()
        dist = flood[r, c]

        for bit, dr, dc in path:
            nr, nc = r + dr, c + dc
            # Verifica existencia da celula dentro da matriz
            if 0 <= nr < maze_size and 0 <= nc < maze_size:
                # Verifica se NAO ha parede bloqueando (Bitwise AND
    )
                if (walls[r, c] & bit) == 0:
                    if flood[nr, nc] == -1:
                        flood[nr, nc] = dist + 1
                        queue.append((nr, nc))
```

Listing 3: Main.py

A função `flood_update` atua como o sistema de navegação do robô, calculando o caminho mais curto de cada célula até o objetivo. Ela implementa o algoritmo de **Busca em Largura (BFS - Breadth-First Search)** para gerar um gradiente de distâncias.

Inicialmente, a função reinicia a matriz `flood` com -1 (não visitado). A área alvo (o centro do labirinto, coordenadas 7,7 a 8,8) é explicitamente definida com distância 0 e adicionada à fila de processamento.

O algoritmo processa a fila iterativamente. Para cada célula extraída, ele examina os quatro vizinhos cardeais definidos na lista `path`. O passo crítico é a verificação de conectividade:

`if (walls[r, c] & bit) == 0:`

Esta operação **Bitwise AND** atua como um "porteiro". Ela consulta a matriz `walls` para verificar se existe uma barreira física na direção específica (`bit`). Se o resultado for 0 (sem parede) **E** o vizinho ainda for desconhecido (-1), o algoritmo atualiza a distância do vizinho para `dist` $+ 1$ e o adiciona à fila. Isso cria uma "rampa" numérica onde os valores diminuem à medida que se aproximam do centro.

2.5 Loop de Controle Principal

```
def main():
    log("Running...")
    API.setColor(0, 0, "G")
    API.setText(0, 0, "Start")

    # Variaveis do Robo
    x = 0
    y = 0
    orient = 0 # 0:N, 1:E, 2:S, 3:W

    # Calculo inicial do mapa
    flood_update()

    while True:
        # Deteccao de Paredes e Tomada de Decisao
        new_wall = False

        # Parede a Frente
        if API.wallFront():
            real_dir = orient
            # Atualiza parede se nao estiver marcada
            if (walls[y, x] & (1 << real_dir)) == 0:
                set_wall(x, y, real_dir)
                new_wall = True
                API.setWall(x, y, "nesw"[real_dir])

        # Parede a Direita
        if API.wallRight():
            real_dir = (orient + 1) % 4
            if (walls[y, x] & (1 << real_dir)) == 0:
                set_wall(x, y, real_dir)
                new_wall = True
                API.setWall(x, y, "nesw"[real_dir])

        # Parede a Esquerda
        if API.wallLeft():
            real_dir = (orient - 1) % 4
            if (walls[y, x] & (1 << real_dir)) == 0:
                set_wall(x, y, real_dir)
                new_wall = True
                API.setWall(x, y, "nesw"[real_dir])

        # Recalcula o mapa APENAS se houver novas paredes
```

```
        if new_wall:
            flood_update()

        # Debug: Mostra a distancia na celula
        API.setText(x, y, str(flood[y, x]))

        # Verifica se chegou ao objetivo
        if flood[y, x] == 0:
            log("CHEGUEI NO CENTRO!")
            API.setColor(x, y, "B") # Pinta de Azul
            break

        # Decide para onde ir (Greedy)
        best_dir = -1
        lowest_value = 9999

        # Verifica os 4 vizinhos (N, E, S, W)
        for d in range(4):
            # Se ha parede nesta direcao, ignora
            if (walls[y, x] & (1 << d)) != 0:
                continue

            # Calcula coordenadas do vizinho
            nx, ny = x, y
            if d == 0: ny += 1
            if d == 1: nx += 1
            if d == 2: ny -= 1
            if d == 3: nx -= 1

            # Verifica limites e valor do Flood Fill
            if 0 <= nx < maze_size and 0 <= ny < maze_size:
                val = flood[ny, nx]
                # Queremos ir para um valor MENOR que o atual
                if val != -1 and val < lowest_value:
                    lowest_value = val
                    best_dir = d

        # Execucao do Movimento
        if best_dir != -1:
            diff = best_dir - orient

            # Logica de rotacao minima
            if diff == 1 or diff == -3:
                API.turnRight()
            elif diff == -1 or diff == 3:
                API.turnLeft()
            elif diff == 2 or diff == -2:
                API.turnRight()
                API.turnRight()

            API.moveForward()

            # Atualiza posicao virtual e orientacao
            orient = best_dir
            if orient == 0: y += 1
            if orient == 1: x += 1
            if orient == 2: y -= 1
```

```
            if orient == 3: x -= 1
        else:
            # Se nenhum caminho for encontrado (Beco sem saida ou Erro)
            log("Preso! Algo deu errado.")
            break

if __name__ == "__main__":
    main()
```

Listing 4: Main.py

A função `main` executa o paradigma clássico da robótica: **Sentir-Planejar-Agir**. Ela opera em um loop infinito, mantendo o estado do robô (coordenadas x, y e orientação) e orquestrando a estratégia de navegação.

Para determinar o próximo movimento, o algoritmo itera pelas quatro direções cardeais possíveis. Ele valida os movimentos verificando os limites da grade e a existência de paredes (usando o mapa de bits). Entre os vizinhos válidos, ele seleciona aquele com o **menor valor de Flood Fill**. Essa abordagem "gananciosa" (*greedy*) garante que o robô siga estritamente o gradiente descendente em direção ao objetivo (distância 0).

Uma vez selecionada a direção ótima (`best_dir`), o robô deve alinhar-se fisicamente. A lógica de rotação calcula a diferença entre a direção alvo e a orientação atual:

$$\texttt{diff} = \texttt{best\_dir} - \texttt{orient}$$

Com base nessa diferença, o código executa as rotações necessárias de 90° (`turnRight`/`turnLeft`) ou 180° antes de mover-se para frente. Finalmente, o robô atualiza manualmente suas coordenadas virtuais (x, y) para manter a sincronia com o movimento físico no simulador.

References

[1] Mackorone. *MMS: Micromouse Simulator*. Repositório GitHub. Disponível em: https://github.com/mackorone/mms. Acesso em: Dez. 2025.

[2] P. V. F. Zawadniak et al. "Micromouse 3D simulator with dynamics capability: a Unity environment approach". *SN Applied Sciences*, vol. 3, Springer Nature, 2021.

[3] S. Rijal, R. Nepal, R. Lwagun, R. Pati, J. Bhatta. "Optimizing Tremaux Algorithm in Micromouse Using Potential Values". *International Journal of Advanced Engineering*, vol. 3, no. 2, Set. 2020.