1. Principais Desafios Encontrados

O desenvolvimento do algoritmo para o robô MicroMouse apresentou desafios técnicos que exigiram uma abordagem estruturada, dividida entre a comunicação com o simulador e a lógica de navegação autônoma.O primeiro grande obstáculo foi a **comunicação Inter-Processos (IPC)** entre o script Python e o simulador MMS. A biblioteca padrão de API fornecida operava de forma síncrona para todos os comandos, o que causava o travamento ("freezing") da execução quando comandos de visualização (como pintar células ou escrever texto) eram enviados sem que o simulador retornasse uma confirmação.

Além disso, a natureza do problema — **exploração de ambiente desconhecido** — exigia que o robô tomasse decisões em tempo real. Diferente de um algoritmo de busca estático (onde o mapa é conhecido a priori), o robô precisava lidar com a incerteza, atualizando sua rota a cada nova parede descoberta, evitando loops infinitos e becos sem saída.

2. Lógica da Solução: Algoritmo Flood Fill

Para resolver o problema de navegação, optamos pela implementação do algoritmo **Flood Fill (Método de Propagação de Ondas)**, fundamentado na teoria de busca em largura (BFS). A solução foi modelada em três etapas cíclicas:

- **Mapeamento Dinâmico (Matriz M):** O robô mantém uma representação interna do labirinto, atualizando as paredes (**walls**) apenas quando seus sensores as detectam fisicamente.
- **Propagação de Ondas (Matriz D):** A cada passo, o algoritmo recalcula uma matriz de distâncias. O objetivo (centro) recebe valor 0, e esse valor se propaga para os vizinhos livres incrementando em 1, criando um "gradiente" de distâncias que desvia dos obstáculos conhecidos[2].
- **Decisão de Movimento (Descida de Gradiente):** O robô, posicionado em uma célula de valor N, busca mover-se sempre para um vizinho acessível de valor N-1. Isso garante matematicamente a convergência para o objetivo pelo caminho mais curto conhecido até o momento

3. Mecanismos Facilitadores e Otimizações

Refatoração da API (`API.py`): Modificamos a camada de comunicação para distinguir comandos bloqueantes (movimento/sensores) de comandos não-bloqueantes (visualização). Isso permitiu pintar o mapa (`setColor`) e exibir os valores de distância (`setText`) em tempo real para depuração, sem comprometer a performance do robô.

Estrutura de Dados Eficiente: Utilizamos a `collections.deque` do Python para gerenciar a fila da busca em largura (BFS). Isso otimiza a propagação das ondas, mantendo a complexidade temporal baixa mesmo em labirintos complexos.

Orientação a Objetos: O código foi encapsulado na classe `MicroMouse`, mantendo o estado do robô (coordenadas `x`, `y`, orientação e mapas de memória) isolado da lógica de

controle, facilitando a manutenção e futuras expansões, como a implementação de uma *Speed Run*.

4. Observações: Porque Flood Fill e não Wall Follower (seguir a parede)? O Wall Follower é limitado e pode falhar em labirintos com ilhas ou loops complexos, nunca encontrando o centro. O Flood Fill garante matematicamente que o robô encontrará o objetivo se houver um caminho possível.