Documentação do Projeto: Solucionador de Labirintos com A*

1. Visão Geral do Projeto

Este projeto é um visualizador avançado de algoritmos de busca, projetado para demonstrar e comparar algoritmos de busca **não informada** e **informada**.

Diferente de um labirinto simples (com apenas caminhos e paredes), esta versão introduz o conceito de **terrenos com custos variados**. Células podem ser "lama", "areia" ou "água", cada uma com um custo de movimento diferente. O objetivo não é apenas encontrar *um* caminho, mas encontrar o caminho de *menor custo total* do início ao fim.

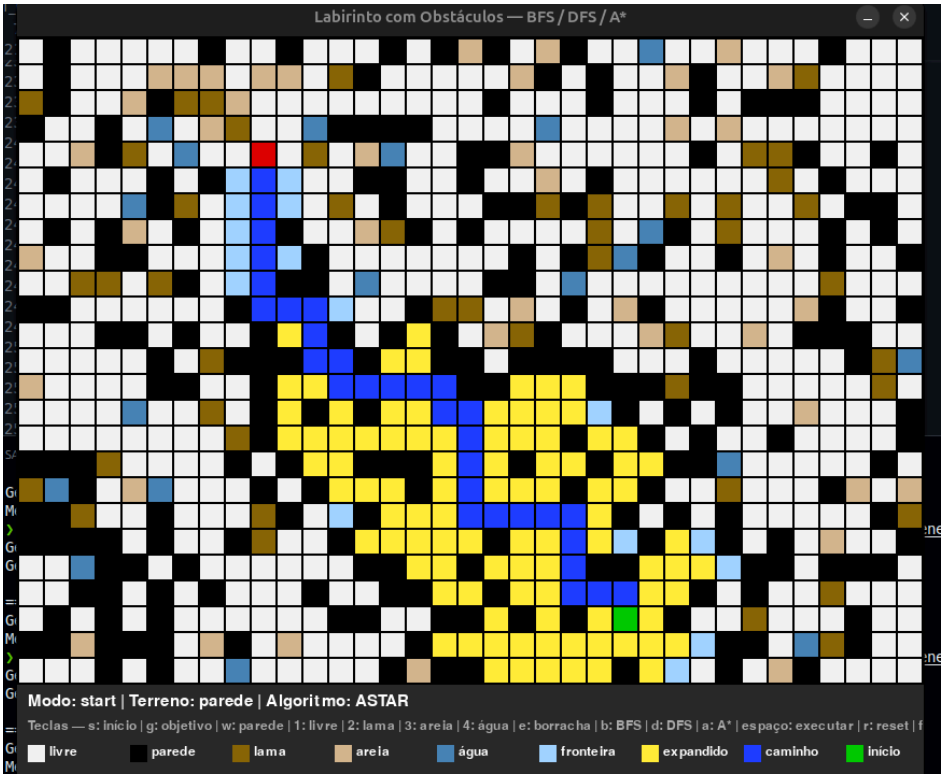

Figura 01: Interface do pygame

2. Como Foi Desenvolvido

O projeto foi desenvolvido em **Python** com a biblioteca **Pygame** para a interface gráfica.

- **Pygame:** Utilizado para desenhar a grade, os diferentes tipos de terreno (com cores distintas), os pontos de início/fim e para animar o processo de busca.

- **Estrutura de Dados (Labirinto):** O labirinto é uma matriz 2D onde cada célula armazena um tipo de terreno (Vazio, Parede, Lama, etc.), que está associado a um custo.
- **Fila de Prioridade (heapq):** Esta é a principal diferença em relação ao projeto anterior. Em vez de uma fila simples (FIFO) ou pilha (LIFO), este código usa uma **fila de prioridade**. Isso é essencial para os algoritmos de busca informada, pois permite que o algoritmo sempre explore primeiro o nó que parece ser o mais promissor (com menor custo ou prioridade).
- Note que para DFS e BFS o algoritmo ainda usa FIFO e LIFO. Foi adicionado um condicional para usar a heap quando A* for selecionado.

```
if self.algorithm in ("bfs", "dfs"):
    self.came_from[(sr, sc)] = None
    self.visited = {(sr, sc)}  # marcar origem como visitada p/ visual
    self.frontier.append((sr, sc))
else:  # A*      Gabriel_Castelo, ontem • A_star
    self.came_from[(sr, sc)] = None
    self.g_score[(sr, sc)] = 0
    h0 = self.heuristic((sr, sc), (gr, gc))
    self.f_score[(sr, sc)] = h0
    # heap de (f, tie, (r,c))
    heapq.heappush(self.open_heap, (h0, self.tie_counter, (sr, sc)))
    self.tie_counter += 1
    self.open_set_cells.add((sr, sc))
    # Em A*, consideramos "visitado" quando expandido (pop do heap)
```

Figura 02: Condicional para uso de heap ou lista simples

3. Algoritmos Utilizados

O foco principal deste projeto é o algoritmo **A* (A-Estrela)**, o algoritmo de busca informada mais proeminente.

Busca Informada: A* (A-Estrela)

O A* é um algoritmo inteligente que encontra o caminho de menor custo de forma muito eficiente. Ele faz isso balanceando dois fatores para decidir qual nó explorar em seguida:

1. **g(n) - Custo Real:** O custo exato acumulado para viajar do ponto inicial até o nó n. O código calcula isso somando os custos dos terrenos (lama custa 3, areia custa 5, etc.) ao longo do caminho.
2. **h(n) - Heurística:** Uma estimativa "otimista" do custo para ir do nó n até o objetivo. Este código usa a **Distância de Manhattan** como heurística, que calcula o número de passos horizontais e verticais necessários para chegar ao fim (ignorando paredes ou custos).

A prioridade de cada nó é calculada como f(n) = g(n) + h(n). Ao expandir sempre o nó com o menor f(n), o A* foca a busca na direção correta (graças ao h(n)) sem ignorar os custos reais do caminho (graças ao g(n)), garantindo a solução ótima.

Outros Algoritmos Incluídos

O código também permite a seleção de outros algoritmos para comparação:

- **BFS e DFS:** Também incluídos para demonstrar como falham em encontrar o caminho de *menor custo* (apenas o mais curto em passos, no caso do BFS).

4. Principais Funções

As funções *start_search(),step_search()* e os métodos *step_dfs(),step_bfs(),step_astar()*, são os responsáveis por executar de fato algoritmo.

```python
def step_search(self):
    if self.algorithm == "bfs":
        self._step_bfs()
    elif self.algorithm == "dfs":
        self._step_dfs()
    else:
        self._step_astar()      Gabriel_Castelo, ontem • A_sta
```

```python
def _step_bfs(self):
    if not self.frontier:
        self.animating = False
        return
    r, c = self.frontier.popleft()
    if (r, c) == self.goal:
        self._reconstruct_path((r, c))
        return
    # expande
    for nbr in self._neighbors4(r, c):
        if nbr not in self.came_from:
            self.came_from[nbr] = (r, c)
            self.frontier.append(nbr)
            self.visited.add(nbr)

def _step_dfs(self):
    if not self.frontier:
        self.animating = False
        return
    r, c = self.frontier.pop()
    if (r, c) == self.goal:
        self._reconstruct_path((r, c))
        return
    # expande
    for nbr in self._neighbors4(r, c):
        if nbr not in self.came_from:
            self.came_from[nbr] = (r, c)
            self.frontier.append(nbr)
            self.visited.add(nbr)
```

```python
def _step_astar(self):
    if not self.open_heap:
        self.animating = False
        return

    # pop menor f
    f, _, (r, c) = heapq.heappop(self.open_heap)
    if (r, c) in self.open_set_cells:
        self.open_set_cells.remove((r, c))
    # agora é um nó expandido
    self.visited.add((r, c))

    if (r, c) == self.goal:
        self._reconstruct_path((r, c))
        return

    current_g = self.g_score.get((r, c), float('inf'))
    gr, gc = self.goal

    for (nr, nc) in self._neighbors4(r, c):
        # custo de mover para o vizinho
        step = COST_MAP[self.grid[nr][nc]]
        tentative_g = current_g + step

        if tentative_g < self.g_score.get((nr, nc), float('inf')):
            self.came_from[(nr, nc)] = (r, c)
            self.g_score[(nr, nc)] = tentative_g
            h = self.heuristic((nr, nc), (gr, gc))
            fn = tentative_g + h
            self.f_score[(nr, nc)] = fn
            heapq.heappush(
                self.open_heap, (fn, self.tie_counter, (nr, nc)))
            self.tie_counter += 1
            self.open_set_cells.add((nr, nc))
```