UNIVERSIDADE FEDERAL DE SANTA CATARINA

GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

SISTEMAS INTELIGENTES - INE5633

Lucas Broering dos Santos[1]

ATIVIDADE PRÁTICA 1

Florianópolis, Setembro de 2025

[1] Graduando em Sistemas de Informação; Contato: lucas.broering@grad.ufsc.br

Sumário

Introdução

O trabalho foi feito utilizando python, utilizei a lib **heapq,** para implementar a lista de prioridade com mais eficiência, ao invés de fazer uma lista ordenada na mão, fica mais limpo o código e eficiente. O pacote **json** para salvar a saída em json. A saída fica dessa forma:

Código

O código foi separado em arquivos, cada arquivo com suas respectivas classes ou funções. São elas:

- **puzzle_8.py**: Onde tem o **__init__** que encapsula um estado do tabuleiro e guarda o index do 0. A função **get_neighbors** gera estados vizinhos, para expandir os nós no A*. O **get_path** reconstrói o caminho da solução.

- **puzzle_solver.py**: O método **a_star_search(heuristic)** onde está o coração do trabalho. Ele recebe uma heurística. Mantém **g**, que é o custo do melhor caminho até aquele momento, e faz f = g + h para ordenar a fronteira. Como mostrado em aula, onde **h** é o valor da heurística. Nesse arquivo também tem um método chamado **uniform_cost_search**, onde eu chamo a função A* com heurística igual a 0.

Gerenciamento de Fronteira

```
if g_curr != g.get(tuple(state.board), float('inf')):
        continue
```

```
for neighbor in state.get_neighbors():
        neighbor_board_t = tuple(neighbor.board)
        tentative_g = g_curr + 1

        if neighbor_board_t in closed and tentative_g >= g.get(neighbor_board_t, float('inf')):
            continue

        if tentative_g < g.get(neighbor_board_t, float('inf')):
            g[neighbor_board_t] = tentative_g
            f_neighbor = tentative_g + heuristic(neighbor)
            counter += 1
            heapq.heappush(frontier, (f_neighbor, counter, tentative_g, neighbor))
```

O **if g_curr !=..** é utilizado para ignorar entradas obsoletas na lista de prioridade. Esse **If neightbor_board_t in…** evita reabrir nós que já foram explorados quando o caminho novo não é melhor. Já o **if tentative_g <..** só é atualizada a fronteira quando há um caminho melhor. Acho que seria esses checks que foram feitos para controlar a fronteira do A*.

Heurísticas

Implementei 4 heurísticas no trabalho, são elas:

- **misplaced_tiles(state, goal)**
    - Conta quantas peças estão fora do lugar. É admissível pois não superestima o custo real dos movimentos. Valores de (0..8).
- **manhattan_distance(state, goal)**

- Soma das distâncias Manhattan (|Δx|+|Δy|) de cada peça até sua posição alvo. Tem valores mais distintos.

- manhattan_distance_linear_conflict(state, goal)

- Quando duas peças estão na mesma linha e estão trocadas de uma forma que cada uma bloqueia a outra, adiciona dois passos extras.

- non_admissible(state, goal)

- Superestima, pois é a manhattan distance * 2, pode gerar soluções não tão boas, mas consegue às vezes acelerar a busca

Resultados

Caso Fácil

Algoritmo	Tamanho Caminho	Nós Visitados	Tempo (s)	Execução	Maior Fronteira
UCS	2	9	0.0000		8
A* admissível simples	2	3	0.0000		5
A* admissível avançada	2	3	0.0005		5
A* não admissível	2	3	0.0000		5

Caso Médio

Algoritmo	Tamanho Caminho	Nós Visitados	Tempo (s)	Execução	Maior Fronteira
UCS	5	41	0.0006		36
A* admissível simples	5	6	0.0000		7
A* admissível avançada	5	6	0.0000		7
A* admissível avançada + LC	5	6	0.0005		7
A* não admissível	5	6	0.0000		7

Caso Difícil

Algoritmo	Tamanho Caminho	Nós Visitados	Tempo (s)	Execução	Maior Fronteira

UCS	31	181439	1.8581	25134
A* admissível simples	31	143840	2.0205	24625
A* admissível avançada	31	20291	0.3040	9310
A* admissível avançada + LC	31	12363	0.4335	6235
A* não admissível	31	411	0.0060	277

Análise

- **UCS** foi o menos eficiente, pois não utiliza heurística e explora massivamente o espaço de estados (mais de **181 mil nós** em casos difíceis).
  *A admissível simples** (misplaced tiles) melhora um pouco em relação ao UCS, mas ainda expande muitos estados por ser uma heurística fraca.
- *A admissível avançada (Manhattan)** reduz drasticamente o número de nós visitados (ex.: de 143k para 20k no caso difícil), mostrando maior precisão.
- *A admissível avançada + conflitos lineares** ainda mais eficiente, reduzindo a fronteira máxima e o total de nós explorados.
- *A não admissível (2× Manhattan)** foi o mais rápido e expandiu pouquíssimos nós. Entretanto, por não ser admissível, **não garante a solução ótima em todos os casos** (apesar de, neste conjunto de testes, ainda ter encontrado o caminho ótimo de custo 31)

How to run

Rodar o arquivo **main.py** e escolher uma opção para rodar 1, 2 ou 3. Como o print abaixo:

.