Atividade Prática I

Trabalho sobre Métodos de busca

Pedro Santi Binotto [20200634][*1], Cauã Pablo Padilha [22100895][†2], Felipe Jun Hatsumura [19206699][‡3], e Gabriel Lemos da Silva [18200628][§4]

[1]Departamento de Informática e Estatística, Universidade Federal de Santa Catarina

24 de setembro de 2025

[*]pedro.binotto@grad.ufsc.br
[†]padilha.caua@grad.ufsc.br
[‡]fjhats@gmail.com
[§]glemoss.dev@gmail.com

Sumário

1 Q1

Proposta — enunciado na *Questão 1*:

> Quais os métodos ou funções principais e suas relações com o algoritmo A*?

1.0.1 Solução — Q1

I/O Primeiramente, os dados de entrada do programa são lidos e interpretados pela função `capture_input`. Os dados deverão ser informados no seguinte formato:

```
M
E0..EN
LEVEL
```

Onde:

- *M* refere-se ao tamanho (do lado) do tabuleiro (ex.: 3 para *8Puzzle* , 4 para *15Puzzle*, etc...);
- *E0..EN* referem-se aos números de $(N = M^2)$ de 0 até N (em qualquer ordem) que expressam o estado inicial do tabuleiro;
- *LEVEL* é um valor dentre:
  - **`L0:`** Para execução de *A** com heurística *nula* (custo uniforme);
  - **`L1:`** Para execução de *A** com heurística não admissível;
  - **`L2:`** Para execução de *A** com heurística *Manhattan [1]*;
  - **`L3:`** Para execução de *A** com heurística tunada;

À partir destes dados, serão produzidas as seguintes estruturas de dados, que possibilitarão o processamento do *puzzle*:

- **`g`** - Grafo que serve de estrutura de dados para o algoritmo, contendo apenas o estado inicial
- **`n`** - Tamanho do tabuleiro
- **`s`** - Estado inicial
- **`t`** - Estado alvo
- **`l`** - Heurística a ser utilizada

A* A função de busca (implementada sob `a_star(g, n, s, t, h)`) recebe, então, estes dados, e inicia o processamento a partir do nó `s` (estado inicial informado ao programa):

```
def a_star(
    g: Graph, n: int, s: State, t: State, h: HeuristicFunction,
        max_nodes: int = 400_000
) -> Result | None:
    """
    'A*' search algorithm
    ...
    """
```

Deste ponto em diante, o algoritmo de busca irá calcular para cada nó (nó da vez) as possibilidades de movimentos alcançáveis a partir do da configuração atual do tabuleiro, similarmente a uma busca em largura (*BFS*), porém utilizando uma fila priorizada (*Priority Queue*) baseada no *score* de cada movimento, que será referente à função f:

$$f(n) = g(n) + h(n)$$

Onde:

- $f(n)$ é o *score* geral;
- $g(n)$ é o valor de *custo* ou *função utilidade* atrelada ao nó;
- $h(n)$ é o valor de *heurístico* atrelado ao nó;

O algoritmo navegará o grafo seguindo a ordem da fila priorizada até encontrar o estado de solução (`t`) ou a fila ficar vazia, que sinalizará que o estado de entrada (`s`) não é solucionável.

Observação Deve ser observado que a implementação de `a_star` recebe também um sexto parâmetro "`max_nodes`", que agirá como uma salvaguarda, para evitar que, em casos de *puzzles* maiores (*15Puzzle*, *24Puzzle*), o programa consuma muita memória do *host*, abortando a busca uma vez que este limite é atingido.

2 Q2

Proposta — enunciado na *Questão 2*:

> Como foi gerenciada a fronteira, ou seja, quais verificações foram feitas antes de adicionar um estado na fronteira? Explicar e mostrar os respectivos trechos de código:

2.0.1 Solução — Q2

A fronteira é gerenciada com uma fila priorizada (*Priority Queue*), inicializada com o estado inicial $[(h(s,t,n), g(s), s)]$:

Onde:

- s é a representação do estado referente ao score, na forma de uma tupla ordenada;
- t é a representação do estado solução, que representa o objetivo da busca;
- n é o tamanho de lado do tabuleiro;
- $g(n)$ é inicializado em 0;
- $h(n)$ é o resultado da função heurística sobre s, t, n;

```
visited_nodes: set[State] = set()
priority_queue: list[tuple[int, int, State]] = [(h(s, t, n),
    NODE_MIN_SCORE, s)]
upper_bound = len(priority_queue)
breadcrumb: Breadcrumb = {}
gs: dict[State, int] = {s: NODE_MIN_SCORE}
```

Estruturas de controle:

- **`visited_nodes`**: Conjunto que armazena estados já expandidos/visitados;
- **`priority_queue`**: Estrutura que mantém a fronteira (nós abertos), ordenada pelo valor de $f(n)$;
- **`upper_bound`**: Limite superior do tamanho da fronteira (para estatísticas);
- **`breadcrumb`**: Grafo mínimo que mantém as relações inversas da árvore principal, utilizado para retraçar o caminho escolhido de forma mais econômica em relação à memória;
- **`gs`**: Dicionário que armazena o menor custo de utilidade conhecido referente à cada estado;

Nós que já constam em `visited_nodes` são ignorados afim de evitar a formação de ciclos no grafo.

```
while priority_queue:
...
    curr_h, curr_g, curr_s = pq_pop(priority_queue)

    if curr_s == t:
        return ...

    if curr_s in visited_nodes:
        continue
    visited_nodes.add(curr_s)
```

Os "filhos" de cada nó são então calculados e explorados (via busca em largura) e submetidos às funções de score e fila priorizada; as métricas como `upper_bound` também são atualizadas:

```
for move in compute_moves(curr_s, n):
    g.add_edge(curr_s, move)
    next_g = curr_g + 1
    if move in visited_nodes and next_g >= gs.get(move,
       NODE_MAX_SCORE):
        continue

    if next_g < gs.get(move, NODE_MAX_SCORE):
        breadcrumb[move] = curr_s
        gs[move] = next_g
        f_neighbor = next_g + h(move, t, n)
        pq_push(priority_queue, (f_neighbor, next_g, move))
        if len(priority_queue) > upper_bound:
            upper_bound = len(priority_queue)
```

3 Q3

Proposta — enunciado na *Questão 3*:

> Descrição das heurísticas e comparação da faixa de valores e da precisão delas (no mínimo: dois casos difíceis, dois médios e um fácil); breve descrição sobre suas implementações.

3.0.1 Solução — Q3

Para atender os requisitos do trabalho foram feitas três heurísticas e o algoritmo sem heurísticas (custo uniforme – L0). As heurísticas escolhidas foram:

- **Heurística não admissível (L1)** - *Manhattan* x 2
- **Heurística admissível (L2)** - *Manhattan*
- **Melhor Heurística admissível (L3)** - *Manhattan* + Conflitos Lineares

A heurística de *Manhattan* calcula a distância em que cada peça está posicionada e da sua posição destino (a posição correta) sem considerar as diagonais, portanto considerando deslocamentos apenas na vertical e horizontal [1]. Essa heurística utilizada é considerada admissível pois não superestima o custo real, porém utilizá-la com o valor dobrado a torna não admissível pois superestima o custo real.

A utilização de *Manhattan* + Conflitos Lineares foi escolhida como a melhor heurística possível pois considera a distância de *Manhattan*, explicada anteriormente, e também leva em consideração a quantidade de peças na linha ou coluna correta mas em posição errada.

Apontamentos

- Custo uniforme sempre retornará $h(n) = 0$. Essa heurística fará a busca baseada apenas em $g(n)$ e explora muito mais estados, se não, todos;
- A heurística de *Manhattan* calcula a distância em que cada peça está posicionada e da sua posição destino (a posição correta) sem considerar as diagonais, portanto considerando deslocamentos apenas na vertical e horizontal. Essa heurística utilizada é considerada admissível pois não superestima o custo real, porém utilizá-la com o valor dobrado a torna não admissível pois superestima o custo real [1];
- A heurística não admissível irá retornar $h(n) * 2$; Isso superestima o custo real e pode otimizar o tempo de execução da busca, porém, retornando um caminho não perfeito;
- A utilização de *Manhattan* + Conflitos Lineares foi escolhida como a melhor heurística possível pois considera a distância de *Manhattan*, explicada anteriormente, e também leva em consideração a quantidade de peças na linha ou coluna correta mas em posição errada. Essa heurística ainda é admissível.

4 Q4

Proposta — enunciado na *Questão 4*:

> Breve análise do desempenho da implementação com uma tabela comparativa (usando as informações da saída - itens a) a d)) das 4 variações implementadas (no mínimo: um caso difícil, um médio e um fácil para as abordagens 3 e 4 e um médio e um fácil para as abordagens 1 e 2):

4.0.1 Solução — Q4

Tabela 1: 8Puzzle

Nível Heurística	Dificuldade	Instância	Tempo (segundos)	Nós Visitados	Máx. Fronteira	Caminho
L0	Fácil	pzzl_0_8	0.517	8163	4485	17
L0	Médio	pzzl_3_8	0.036	8164	4485	17
L1	Fácil	pzzl_0_8	0.001	154	99	17
L1	Médio	pzzl_3_8	0.0008	127	81	19
L2	Fácil	pzzl_0_8	0.0009	194	121	17
L2	Médio	pzzl_3_8	0.001	202	126	17
L2	Difícil	pzzl_7_8	0.0003	47	37	13
L3	Fácil	pzzl_0_8	0.001	117	79	17
L3	Médio	pzzl_3_8	0.002	126	83	17
L3	Difícil	pzzl_7_8	0.0006	31	24	13

Tabela 2: 15Puzzle

Heurística	Dificuldade	Instância	Tempo (s)	Nós Visitados	Fronteira Max	Tamanho Caminho
L0	Fácil	pzzl_0_15	3.23	400000	378572	N/A
L0	Médio	pzzl_3_15	3.21	400000	378572	N/A
L1	Fácil	pzzl_0_15	4.69	400000	357639	N/A
L1	Médio	pzzl_3_15	4.72	400000	351535	N/A
L2	Fácil	pzzl_0_15	4.79	400000	347360	N/A
L2	Médio	pzzl_3_15	4.75	400000	353335	N/A
L2	Difícil	pzzl_7_15	4.98	400000	358550	N/a
L3	Fácil	pzzl_0_15	9.83	400000	344200	N/A
L3	Médio	pzzl_3_15	10.37	400000	360985	N/A
L3	Difícil	pzzl_7_15	10.52	400000	361953	N/A

5 Q5

Proposta — enunciado na *Questão 5*:

> Caso algum dos objetivos não tenha sido alcançado explique o que você faria VS o que foi feito e exatamente qual o(s) problema(s) encontrado(s), bem como limitações da implementação:

5.0.1 Solução — Q5

Para os *puzzles* maiores (4^2, 5^2), não foi possível fazer uma leitura exata das versões mais rudimentares/menos eficientes da busca (custo uniforme ou heurística não admissível), como a complexidade de tempo e memória era alta demais e o hardware não era capaz de encontrar a solução nestes casos.

Por este motivo, adicionamos um parâmetro de "`max_nodes`" (*default: 400000 nós*) à função de busca, que determina um número máximo de nós que podem ser explorados antes de abortar a operação, afim de evitar situações que resultariam no congelamento do sistema *host*, como aconteceu algumas vezes em desenvolvimento.

Assim, observa-se na Q4 que alguns dados nas tabelas estão ausentes, pois não conseguimos produzir um resultado satisfatório — isto é, uma solução para o *puzzle*.

Não identificamos se isso é indicativo de um problema na implementação das nossas heurísticas ou uma questão de ordem de magnitude.

6 Bibliografia

Referências

[1] Shrawan Kumar Sharma and Shiv Kumar. Comparative analysis of manhattan and euclidean distance metrics using a* algorithm. *J. Res. Eng. Appl. Sci*, 1(4):196–198, 2016.