# Comparação de heurísticas aplicadas no algoritmo A*

**Allan Diamante de Souza, 105423**

**Felipe Diniz Tomás, 110752**

[1]Departamento de informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

Modelagem e otimização Algoritmica – 6903

Ra110752@uem.br, Ra105423@uem.br

12, Abril 2021

## 1. Introdução

Publicado pela primeira vez em 1968 pelo grupo de pesquisadores do Instuto de pesquisa de Stanford[1], o A-estrela (A*) é um algoritmo de busca de caminho que utiliza uma combinação de aproximações heurísticas para resolver determinados tipos de problemas. A busca é realizada em um grafo começando de um vértice inicial, tendo como destino um vértice final. É frequentemente usado em muitos campos da ciência da computação devido à sua integridade, otimização e eficiência[2], sendo suas principais aplicações voltadas a encontrar rotas de deslocamento entre localidades, além de ser preferencialmente utilizado em problemas de quebra-cabeça, como é o caso do N-Puzzle.

Por ser um algoritmo heurístico, um dos principais modificadores no desempenho do algoritmo A* é a heurística aplicada, onde sua variação faz com que a execução siga diferentes caminhos. No entanto é notado que o algoritmo pode gerar um grande número de nós e que sua arvóre de busca pode crescer exageradamente, resultando em um grande uso de memória em determinadas situações, mas como destacado por Zeng and Church (2009) continua sendo o melhor solução em muitos casos.

Sendo assim, o objetivo deste trabalho é analisar o comportamento do algoritmo A* utilizando 5 heurísticas diferentes, para resolver o problema do quebra-cabeça 15-puzzle, observando o tempo de execução e uso de memória entre elas. As heurísticas aplicadas são: o número de peças foras de lugar (de acordo com o tabuleiro destino), o número de peças fora de ordem na sequência numérica das 15 peças (seguindo a ordem das posições no tabuleiro destino), o somatório da Distância Manhattan para cada peça fora do lugar, a combinação linear entre heurísticas e o máximo valor entre heurísticas.

## 2. O problema

O jogo 15-Puzzle, também conhecido como jogo das 15 peças, trata-se de um quebra-cabeças de quinze peças, composto por um tabuleiro com 15 números, que

---

[1] https://en.wikipedia.org/wiki/A*_search_algorithm

[2] https://link.springer.com/chapter/10.1007%2F978-3-642-02094-0_7

trocam de lugar através de um espaço vazio. O objetivo é arranjar as peças em ordem, da esquerda para a direita, de cima a baixo[3]. É um problema popular para modelagem de heurísticas e utilização das mesmas.

O problema consiste em dado uma configuração inical qualquer do tabuleiro, é necessário descobrir a quantidade mínima de movimentos sequênciais para chegar à configuração final do tabuleiro final. A seguir é possível observar um exemplo de uma configuração inicial para o tabuleiro 1, e o tabuleiro 2 final.

| | | | |
|---|---|---|---|
| 5 | | 9 | 13 |
| 1 | 2 | 10 | 14 |
| 3 | 6 | 7 | 11 |
| 4 | 8 | 12 | 15 |

**Tabuleiro 1. Tabuleiro embaralhado**

| | | | |
|---|---|---|---|
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 | |

**Tabuleiro 2. Tabuleiro final**

## 3. Algoritmo A*

O algoritmo A* é um algoritmo para Busca de Caminho, que realiza a busca do caminho em um grafo de um vértice inicial até um vértice final. O processo utiliza uma estrutura de árvore de nós sucessores originados de nós pais, que se expande indefinidamente a cada iteração até alcançar o nó objetivo.

Para determinar qual nó sucessor seguir, o algoritmo realiza o cálculo da função $f(n)$ em cada iteração. Essa função irá determinar através de um valor qual nó sucessor chega mais próximo do nó final, guiando assim as próximas iterações do algoritmo. Esta função é dada por:

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

No qual:

- $g(n)$ é o custo do nó inicial até o nó $n$
- $h(n)$ é uma aproximação fornecida pela função heurística do custo do nó $n$ até o nó objetivo.

[3] https://zenodo.org/record/979689

## 4. Heurísticas

Como explicado anteriormente o algoritmo A* deve aplicar determinada heurística a fim de encontrar o caminho. Através delas é possível identificar a eficiência de cada uma, e como a mesma afeta o processo de execução.

### 4.1. $h'_1(n)$ – Número de peças fora de lugar

Esta heurística consiste na contagem de peças fora do lugar tendo como base a configuração final. Considerando o exemplo do Tabuleiro 1 e o Tabuleiro 2, visto na seção 2, o a heurística retornará 8 peças fora do lugar. A complexidade desta heurística é linear já que percorrerá sequencialmente todo o tabuleiro verificando cada peça, logo como o tabuleiro permanece do mesmo tamanho, o tempo de execução é constante para cada chamada heurística.

### 4.2. $h'_2(n)$ – Número de peças fora de ordem de acordo com a sequência numérica

Esta heurística consiste na contagem de peças fora do lugar tendo como base a configuração final. Basicamente a heurística olha para cada peça e verifica se a próxima na sequência (de acordo com o tabuleiro final) é sua sucessora em ordem númerica. Por exemplo, em uma peça de valor 5, espera-se que apróxima na sequência seja a número 6, caso contrário contabiliza fora de ordem.

No Tabuleiro 1 visto na seção 2, tem como retorno 9 peças fora de ordem, sendo elas em amarelo:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | 3 | 4 | 0 | 2 | 6 | 8 | 9 | 10 | 7 | 12 | 13 | 14 | 11 | 15 |

**Tabela 1. Peças fora de ordem**

A complexidade da heurística é linear, já que deve percorrer todo o tabuleiro verificando as peças fora de ordem. A execução ocorre em tempo constante dado o tamnho fixo do tabuleiro.

### 4.3. $h'_3(n)$ – Distância Manhattan

Esta heurística consiste no cálculo da distância Manhattan entre cada peça e seu respectivo lugar na configuração final. Ou seja, para cada peça fora de seu lugar soma a distância Manhattam (quantidade de deslocamentos) para colocar em seu devido lugar.

De maneira mais formal, podemos definir a distância de Manhattan entre dois pontos num espaço euclidiano com um sistema cartesiano de coordenadas fixo como a soma dos comprimentos da projecção da linha que une os pontos com os eixos das coordenadas[4].

Por exemplo, num plano que contem os pontos $P_1$ e $P_2$, respectivamente com as coordenadas ( $x_1, y_1$) e $(x_2, y_2)$ é definido por:

$$|x_1 - x_2| + |y_1 - y_2|$$

[4] https://pt.wikipedia.org/wiki/Geometria_pombalina

Quando aplicado ao tabuleiro 15-puzzle, $P_1$ passa a ser a peça do tabuleiro desornedado e $P_2$ a mesma peça no tabuleiro final. Novamente a complexidade deste algorito depende do tamanho do tabuleiro.

### 4.4. $h'_4(n)$ – Combinação linear entre heurísticas

Nesta heurística ocorrerá a soma do resultado de heurísticas anteriores multiplicado por pesos, $\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}$ que totalizam 1.

$$p_1 * h'_1(n) + p_2 * h'_2(n) + p_3 * h'_3(n)$$

Onde foi definido:

- $\boldsymbol{p_1} = 0.3$
- $\boldsymbol{p_2} = 0.2$
- $\boldsymbol{p_3} = 0.5$

A complexidade depende das heurísticas anteriores, assim como seu tempo de execução.

### 4.5. $h'_4(n)$ – Máximo entre as heurísticas

Está heurística utilizará o máximo das heurísticas anteriores, portanto sua complexidade também depende das mesmas.

$$max\ (h'_1(n), h'_2(n), h'_3(n)).$$

## 5. Metodologia

Quando se trata de metodologia, foi necessário estabelecer recursos para que os testes possam ser feitos de forma justa. Esta seção dica-se a explicar os métodos e recursos utilizados no projeto.

### 5.1. Máquina

Foi utilizado uma máquina pessoal com as configurações a seguir[5]:

- Processador: Intel(R) Core(TM) i7-3770k CPU @ 3.50GHz;
- Memória RAM: 16.0GB;
- Disco: 100.6GB;
- GPU: RX580 8gb
- OS: Windows 10 Home

### 5.2. Linguagem

A linguagem utilizada foi o *python* 3.8[6] .

---

[5] https://towardsdatascience.com/colab-pro-is-it-worth-the-money-32a1744f42a8

[6] https://docs.python.org/

## 5.3. Métrica de tempo e memória

Para calcular o tempo de execução foi utilizado o módulo *time.time()*[7] e a verificação de memória através da biblioteca *sys*[8] utilizando o método *getsizeof()* dos conjuntos A e F combinados.

## 5.4. Estrutura dos dados

Ao implementar o algoritmo A* é necessário estabelecer uma estrutua de dados para armazenar adequadamente os conjuntos A e F, além da estrutura dos nós.

- O conjunto A: estados que já foram gerados, mas ainda não foram processados pelo algoritmo;
- O conjunto F: estados que já foram processados pelo algoritmo.

No projeto o nó foi estruturado como uma classe (*Node*), possuido o estado do tabuleiro (*state*), o pai que gerou o nó (*parent*), o custo de $g(n)$ (*gcost*) o custo de $h(n)$ (*hcost*) e uma função paro cálculo de $f(n)$, comno é mostrado na Figura 1.

```
class Node:
    def __init__(self, state, parent, gcost = 0, hcost = 0):
        self.state = state
        self.gcost = gcost
        self.parent = parent
        self.hcost = hcost

    def f(self):
        return self.gcost + self.hcost
```

**Figura 1. Classe Node**

A estrutura fundamental do código A* é a do conjunto A e F, que foi implementado em tabela hash, que existe por padrão em python com o uso de dicionário. As tabelas hash tem como chave uma *string* com o estado do tabuleiro (*iniTable.state*) e o nó (*iniTable*). Além disso, foi necessário um estrutura de heap para para armazenar o valor da função $f(n)$ e a respectiva chave do dicionário. Como mostrado na figura a seguir:

```
heap = [(iniTable.f(), iniTable)]
A = {str(iniTable.state) : iniTable}
F = {}
```

**Figura 2. Estrutura do conjunto A e F**

Através da biblioteca *heapq*[9] é possível utilizar a estrutura heap como heap minímo, para que seja possível recuperar o menor valor de $f(n)$ sem percorrer todo conjunto A. Buscar uma chave em dicionário tem tempo constante de $O(1)$ e encontrar

[7] https://pypi.org/project/ipython-autotime/

[8] https://docs.python.org/3/library/sys.html

[9] https://docs.python.org/3/library/heapq.html

o mínimo em um heap minímo pode ser feito em tempo constante, por fim verificar se um valor está em um dicionário tem tempo $O(n)$.

## 5.5 Analise das linhas 9 e 10 do algoritmo A* (versão II)

As linhas 9 e 10 são responsáveis pela otimização da árvore, ou seja, dado um estado do tabuleiro (nó), essa parte do código irá verificar se o estado já foi descoberto anteriormente e está no conjunto A, se sim, caso o caminho atual encontrado para esse nó for menor (considerando $g(n)$) do que o que está em A, irá removê-lo de A.

Esse processo do algoritmo influência diretamente no uso de recurso do mesmo. A memória não armazenará caminhos que foram encontrados com o maior $g(n)$ otimizando o armazenamento, conseqüentemente o desempenho de processamento também é afetado por não processar os caminhos desnecessário.

## 6. Casos de testes

Dez configurações iniciais para o tabuleiro foram fornecidas pelo professor, (onde 0 representa o espaço em branco), para a comparação das heurísticas. Os testes são:

| Testes | Configuração inicial |
|---|---|
| 1 | 0 2 9 13 3 1 5 14 4 7 6 10 8 11 12 15 |
| 2 | 3 2 1 9 0 5 6 13 4 7 10 14 8 12 15 11 |
| 3 | 2 1 9 13 3 5 10 14 4 6 11 15 7 8 12 0 |
| 4 | 9 13 10 0 5 2 6 14 1 7 11 15 3 4 8 12 |
| 5 | 4 3 2 1 8 10 11 5 12 6 0 9 15 7 14 13 |
| 6 | 9 13 14 15 5 6 10 8 0 1 11 12 7 2 3 4 |
| 7 | 10 6 2 1 7 13 9 5 0 15 14 12 11 3 4 8 |
| 8 | 6 2 1 5 4 10 13 9 0 8 3 7 12 15 11 14 |
| 9 | 10 13 15 0 5 9 14 11 1 2 6 7 3 4 8 12 |
| 10 | 5 9 13 14 1 6 7 10 11 15 12 0 8 2 3 4 |

**Tabela 2. Casos testes**

A partir das configurações iniciais poderá ser testado o consumo de tempo e de memória em cada heurística, comparando o desempenho das mesmas. O consumo de memória é interessante pois é um dos possíveis limitadores do algoritmo A*, logo rastrear a quantidade de memória sendo utilizada é essencial para medir a eficiência do algoritmo.

## 7. Resultado e análise

A seguir veremos o número de movimentos minímos para chegar ao tabuleiro final para cada caso de teste, ou seja o resultado esperado pelo algoritmo.

| Testes | Número de movimentos |
|---|---|
| 1 | 18 |
| 2 | 19 |
| 3 | 12 |
| 4 | 21 |
| 5 | 38 |
| 6 | 32 |
| 7 | 38 |
| 8 | 32 |
| 9 | 27 |
| 10 | 29 |

**Tabela 3. Casos testes**

Agora podemos partir para análise das métricas de memória e tempo.

### 7.2. Consumo de memória

O consumo de memória, de acordo com a heurística, é o principal fator limitante do algoritmo A*. Em algumas das heurística não foi possível executar todos os teste pois houve estouro de memória. A seguir a Tabela 4 indica a quantidade de memória utilizada em cada teste por cada uma das cinco heurísticas. "N/A" indica que houve estouro de memória.

| Testes | $h'_1(n)$ | $h'_2(n)$ | $h'_3(n)$ | $h'_4(n)$ | $h'_5(n)$ |
|---|---|---|---|---|---|
| 1 | 46 KB | 110.7 KB | 3.48 KB | 14.08 KB | 6.9 KB |
| 2 | 27 KB | 221.3 KB | 1.8 KB | 6.9 KB | 3.4 KB |
| 3 | 2.9 KB | 6.96 KB | 1.8 KB | 1.8 KB | 1.8 KB |
| 4 | 110 KB | 442.5 KB | 14.0 KB | 27.8 KB | 14 KB |
| 5 | N/A | N/A | 6.98 KB | 7.86 MB | 6.9 KB |
| 6 | 62.90 MB | N/A | 14.0 KB | 884.9 KB | 221.3 KB |
| 7 | N/A | N/A | 27.8 KB | 31.4 MB | 1.90 MB |
| 8 | 83.88 MB | 251.6 MB | 55.4 KB | 1.9 MB | 110.7 KB |
| 9 | 3.93 MB | 125.8 MB | 3.44 KB | 184.5 KB | 110.7 KB |
| 10 | 31.45 MB | 167.7 MB | 110.7 KB | 884.9 KB | 442.5 KB |

**Tabela 4. Consumo de memória**

### 7.2. Consumo de tempo

Quando se trata do consumo de tempo, como já explicado anteriormente, cada heurística pode influênciar muito no desempenho do algoritmo, fazendo com que uma seja computacionalmente viável e outra não. Essa diferença resulta em um comportamento diretamente ligado ao tempo, podendo demorar mais para chegar ao resultado esperado. É importante ressaltar que a prioridade da implementação neste projeto foi o tempo e não o consumo de memória. A seguir na Tabela 5 podemos ver os resultados de cada teste para cada uma das 5 heurísticas aplicadas.

| Testes | $h'_1(n)$ | $h'_2(n)$ | $h'_3(n)$ | $h'_4(n)$ | $h'_5(n)$ |
|---|---|---|---|---|---|
| 1 | 0.05 s | 0.1 s | 0.005 s | 0.019 s | 0.011 s |
| 2 | 0.008 s | 0.331 s | 0.006 s | 0.011 s | 0.008 s |
| 3 | 0.003 s | 0.009 s | 0.004 s | 0.006 s | 0.004 s |
| 4 | 0.146 s | 0.437 s | 0.015 s | 0.04 s | 0.026 s |
| 5 | N/A | N/A | 0.009 s | 10.115 s | 0.013 s |
| 6 | 1654.09 s | N/A | 0.019 s | 1.899 s | 0.354 s |
| 7 | N/A | N/A | 0.041 s | 43.304 s | 4.604 s |
| 8 | 3195.30 s | 6839.46 s | 0.056 s | 2.069 s | 0.158 s |
| 9 | 3.78 s | 3990.22 s | 0.004 s | 0.306 s | 0.19 s |
| 10 | 549.5 s | 5087.92 s | 0.125 s | 1.811 s | 0.783 s |

**Tabela 5. Consumo de tempo (em segundos)**

## 7.3 Analise entre as heurísticas

Este tópico analisará dentre os testes, qual método teve o melhor desempenho de tempo e o menor consumo de memória. A seguir veremos algumas tabelas de comparação de tempo médio e memória média entre cada heurística dos testes que não houveram estouro, visto na tabela 3 e 4 anteriormente.

| Tempo médio $h'_1(n)$ | Memória média $h'_1(n)$ |
|---|---|
| 4.25 vezes mais rápido que $h'_2$ | $h'_2$ utiliza 464% do espaço da $h'_1$ |
| 16,020 vezes mais lento que $h'_3$ | $h'_3$ utiliza 0.16% do espaço da $h'_1$ |
| 879 vezes mais lento que $h'_4$ | $h'_4$ utiliza 1.81% do espaço da $h'_1$ |
| 2,443 vezes mais lento que $h'_5$ | $h'_5$ utiliza 0.59% do espaço da $h'_1$ |

**Tabela 5. Comparação da $h'_1(n)$ com as demais heurísticas**

| Tempo médio $h'_2(n)$ | Memória média $h'_2(n)$ |
|---|---|
| 4.25 vezes mais lento que $h'_1$ | $h'_1$ utiliza 21.55% do espaço da $h'_2$ |
| 68,020 vezes mais lento que $h'_3$ | $h'_3$ utiliza 0.04% do espaço da $h'_2$ |
| 3,730 vezes mais lento que $h'_4$ | $h'_4$ utiliza 0.39% do espaço da $h'_2$ |
| 10,37 vezes mais lento que $h'_5$ | $h'_5$Utiliza 0.13% do espaço da $h'_2$ |

**Tabela 6. Comparação da $h'_2(n)$ com as demais heurísticas**

| Tempo médio $h'_3(n)$ | Memória média $h'_3(n)$ |
|---|---|
| 16,020 vezes mais rápida que $h'_1$ | $h'_1$ utiliza 61,477% do espaço da $h'_3$ |
| 68,027 vezes mais rápida que $h'_2$ | $h'_2$ utiliza 285,271% do espaço da $h'_3$ |
| 18 vezes mais rápida que $h'_4$ | $h'_4$ utiliza 1,113% do espaço da $h'_3$ |
| 6.5 vezes mais rápida que $h'_5$ | $h'_5$ utiliza 362% do espaço da $h'_3$ |

**Tabela 7. Comparação da $h'_3(n)$ da com as demais heurísticas**

| **Tempo médio $\boldsymbol{h'_4(n)}$** | **Memória média $\boldsymbol{h'_4(n)}$** |
|---|---|
| 879.5 vezes mais rápida que $h'_1$ | $h'_1$ utiliza 5,525% do espaço da $h'_4$ |
| 3,734 vezes mais rápida que $h'_2$ | $h'_2$ utiliza 25,640% do espaço da $h'_4$ |
| 18 vezes mais lenta que $h'_3$ | $h'_3$ utiliza 9% do espaço da $h'_4$ |
| 2.78 vezes mais lenta que $h'_5$ | $h'_5$ utiliza 33% do espaço da $h'_4$ |

**Tabela 8. Comparação da $\boldsymbol{h'_4(n)}$ da com as demais heurísticas**

| **Tempo médio $\boldsymbol{h'_5(n)}$** | **Memória média $\boldsymbol{h'_5(n)}$** |
|---|---|
| 2,443 vezes mais rápida que $h'_1$ | $h'_1$ utiliza 16,984% do espaço da $h'_5$ |
| 10,377 vezes mais rápida que $h'_2$ | $h'_2$ utiliza 78,809% do espaço da $h'_5$ |
| 6.56 vezes mais lenta que $h'_3$ | $h'_3$ utiliza 28% do espaço da $h'_5$ |
| 2.78 vezes mais rápida que $h'_4$ | $h'_4$ utiliza 307% do espaço da $h'_5$ |

**Tabela 9. Comparação da $\boldsymbol{h'_5(n)}$ da com as demais heurísticas**

É perceptível que após todos os testes processados a $h'_2(n)$ (peças fora de ordem de acordo com a sequência numérica) é a mais custosa, tanto em tempo quanto em memória comparada as demais (Tabela 6). Já a heurística mais rápida e com menor consumo de memória é a $h'_3(n)$ (distância Manhattan) (Tabela 7). Também notou-se que tanto a heurística 1 quanto a heurística 2 não foram capazes de processar todos os testes, pois houve estouro de memória.

## 8. Conclusão

Conclui-se que a estrutura de dados arvore A* é influenciada pela heurística aplicada pra resolução do problema, porém caso utilizado uma heurística ineficiente o algoritmo gerará uma arvore muito profunda, consumindo muita memória e tempo.

Após analisarmos cada heurística aplicada na arvore A*, fica nítido a discrepância de desempenho entre as heurísticas, sendo a heurística 3 (distância Manhattan) a melhor delas. Isso ocorre porque a heurística 3 diferente das demais, contabiliza uma informação mais precisa de cada elemento do tabuleiro do que as outras heurísticas, melhorando a escolha dos seus sucessores.

Em trabalhos futuros pode ser realizado mais testes para melhorar a precisão das comparações entre as heurísticas.

## Referências

Zeng, W.; Church, R. L. (2009). "Finding shortest paths on real road networks: the case for A*". *International Journal of Geographical Information Science*.