Comparação de desempenho de diferentes heurísticas no algoritmos A*

Luiz Fellipe Machi Pereira[1], **Mateus Tenório dos Santos**[1]

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

ra103491@uem.br,ra99829@uem.br

1. Introdução

Descrito em 1968 por *Peter Hart, Niss Nilsson* e *Bertram Raphael* como *Algoritmo A*[1], o algoritmo A* (que é o algoritmo A utilizando de heurística apropriada para atingir um comportamento ótimo) trata-se de um algoritmo para realizar a busca de caminho utilizando uma combinação de aproximações heurísticas do algoritmo de Busca em Largura e do Algoritmo de Dijkstra. Ele realiza a busca em um grafo começando de um vértice inicial com destino o vértice desejado.

As aplicações do algoritmo A* vão desde encontrar rotas de deslocamento entre localidades a resolução de problemas, como a resolução de quebra-cabeças, que é onde o N-puzzle (ou no nosso caso, 15-puzzle) se encaixa. O único ”problema”do algoritmo A*, é que sua árvore de busca gerada pode crescer indefinidamente até que o estado desejado seja encontrado, como descrito por [Culberson and Schaeffer 1994], dessa forma, aumentando o consumo de memória e tempo.

Um dos principais modificadores no desempenho do algoritmo A* é a heurística utilizada, onde diferentes heurísticas levam o algoritmo tomar diferentes caminhos. O objetivo desse trabalho é analisar o comportamento do algoritmo A* utilizando 5 heurísticas diferentes, onde poderemos analisar como cada heurística impacta no tempo e no uso de memória. Para isso, utilizaremos do algoritmo A* para resolvermos o quebra-cabeça 15-puzzle, aplicando as seguintes heurísticas : número de peças foras de seu lugar, número de peças fora de ordem na sequência numérica das 15 peças(seguindo a ordem das posições no tabuleiro), somatório da Distância Manhattan para cada peça fora do lugar, combinação linear entre heurísticas e máximo valor entre heurísticas.

2. 15-puzzle

O 15-puzzle, também conhecido como jogo do 15, é um quebra-cabeça baseado em deslizar peças em um tabuleiro que encontram-se fora de ordem. O objetivo do jogo é colocar as peças em ordem, fazendo movimento deslizantes que usam o espaço vazio. Um exemplo do tabuleiro pode ser visto na Tabela 1 e na Tabela 2. Este jogo é um problema clássico para modelagem de algoritmos envolvendo heurísticas.

[1] Algoritmo A*, WikiPedia https://pt.wikipedia.org/wiki/Algoritmo_A⋆

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

Tabela 1. Tabuleiro desordenado

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

Tabela 2. Configuração final

3. Algoritmo A*

O algoritmo A* é um algoritmo heurístico de busca que visa encontrar um caminho para o nó objetivo com o menor custo. O algoritmo A* consegue isso através da manutenção de uma árvore (que cresce em uma borda a cada iteração) de caminhos originados desde o nó inicial. Esta árvore cresce indefinidamente até que o nó objetivo seja alcançado.

Para determinar qual caminho o algoritmo deve percorrer, é realizado o cálculo de uma função $f(n)$ em cada iteração do algoritmo. Essa função guia o A* dando um valor estimado do custo do nó n até o nó objetivo. A função $f(n)$ é dada por :

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

onde:

- n representa um estado/nó qualquer;
- $g(n)$ é o custo do nó inicial até o nó n dado;
- $h(n)$ é uma estimativa dada por uma função heurística do custo do nó n até o nó objetivo.

4. Metodologia

O algoritmo A*, bem como as heurísticas foram implementadas utilizando a linguagem Python. Para aferir a quantidade de memória utilizada e o tempo necessário para computação da resposta utilizou-se a ferramente *GNU time*. A máquina utilizada foi uma instância *VM*, disponível na plataforma Google Cloud, com 1 vCPUj. 125 GB de memória RAM e sistema operacional Ubuntu 19.04.

4.1. Estrutura de dados utilizada

O algoritmo A* durante seu processamento gera muitos estados, cada estado gerado é armazenado um tabuleiro e as informações de $f(n)$, $g(n)$, $h(n)$ e lista de sucessores. São três os conjuntos para armazenamento de estados, ou nós, gerados pelo algoritmo A*, sendo eles os conjuntos A, F e Γ.

- Conjunto A - Conjunto dos estados abertos: são os estados que já foram gerados, mas ainda não foram processados pelo algoritmo;
- Conjunto F - Conjunto dos estados fechados: aqueles que já foram processados pelo algoritmo. Pode ser tratado também como o conjunto dos estados que formam o caminho do estado inicial até o estado objetivo;
- Conjunto Γ - Conjunto dos estados sucessores: é o conjunto de estados sucessores de um determinado estado v.

Porém, como citado anteriormente, devemos tomar cuidado no momento de implementação, pois a árvore gerada pelo algoritmo A* cresce rapidamente, devido a grande quantidade de estados gerados por iteração. Portanto, é necessário utilizarmos de estruturas de dados que possuam bons tempos de acesso, inserção e remoção.

A estrutura escolhida para armazenar os conjuntos A e F foi a o dicionário, que é padrão da linguagem Python. Um dicionário permite armazenar um par chave-valor, passando a chave de acesso é possível realize várias operações com complexidade $\mathcal{O}(1)$ em relação ao tempo, sendo elas armazenar um item, recuperar um item pela chave, verificar se a chave está contida no dicionário e remover um item. Verificar se um objeto está entre os valores do dicionário possui complexidade $\mathcal{O}(n)$ em relação ao tempo. [2]

Contudo em cada iteração do algoritmo é necessário saber qual elemento possui o menor valor de $f(n)$. Para que não fosse necessário iterar sobre o conjunto A, operação feita em $\mathcal{O}(n)$, usou-se a estrutura heap mínimo, ou *heap queue*, que guarda uma cópia do conjunto mapeando o valor de $f(n)$ com o tabuleiro correspondente. Desta forma o acesso ao elemento com menor valor de $f(n)$ é constante, uma vez que este é o primeiro elemento da estrutura[3].

Finalmente, para armazenar o conjunto Γ utilizou-se da estrutura lista, uma vez que se faz necessário iterar sobre o conjunto na execução do algoritmo, sendo inevitável um tempo de execução linear.

4.2. Heurísticas

Esta sessão será dedicada a apresentação das heurísticas utilizadas na implementação do algoritmo A*.

4.2.1. $h'_1(n)$ - Número de peças foras de seu lugar

A heurística $h'_1(n)$ consiste na contagem das peças que estão fora de seu lugar na configuração final (tabela 2). No exemplo da tabela 1 a heurística retorna o valor 2 uma vez que as peças 14 e 15 estão fora de lugar. A complexidade é linear em relação ao tamanho do tabuleiro, uma vez que é necessário percorre-lo sequencialmente para encontrar quais peças estão fora do devido lugar. Neste caso o tempo de execução é constante devido a não variação do tamanho do tabuleiro.

[2]Time Complexities of the Dictionary's Functions: `https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/DataStructures_II_Dictionaries.html`

[3]Heap queue algorithm: `https://docs.python.org/2/library/heapq.html`

4.2.2. $h'_2(n)$ - Número de peças fora de ordem na sequência numérica

Esta heurística consiste na contagem das peças que estão fora de seu lugar na sequência numérica das 15 peças. Vamos supor a configuração inicial da tabela 3:

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

Tabela 3. Tabuleiro desordenado

Primeiro passo considerar a sequências de "posições" no tabuleiro conforme a sequência "numérica" esperada para o estado final, ou seja, iniciando na posição [0,0] (linha 0 e coluna 0) e terminando na posição com o zero (vazio). Levando isso em consideração, obteremos a seguinte sequencia :

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

Tabela 4. Sequência gerada a partir do tabuleiro desordenado

Agora, é analisada a sequência numérica. As células em destacadas são as posições que são contabilizadas para a heurística. Podemos ter duas ocorrências nesta heurística. A primeira é que a posição atual é inferior a posição anterior, e a segunda é que a posição atual não é a próxima posição em comparação com a anterior, ou seja, pos[atual] $>$ pos[anterior + 1]. É válido notar que após o 0, ou posição vazia, não é contabilizada uma ocorrência. Neste exemplo, a heurística deve retornar como valor 8.

A complexidade da Heurística é linear ao tamanho do tabuleiro. Como o tamanho do tabuleiro não varia, o tempo de execução é constante.

4.2.3. $h'_3(n)$ - Somatório da Distância Manhattan

Esta heurística consiste em realizar o cálculo da Distância Manhattan para cada peça encontrada fora de seu lugar, tendo como base o tabuleiro final esperado. Dessa forma, o valor da heurística é a soma de todas as Distâncias Manhattan aplicadas.

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

Por exemplo, num plano que contem os pontos $P1$ e $P2$, respectivamente com as coordenadas $(x1, y1)$ e $(x2, y2)$, é definido por :

$$|x1 - x2| + |y1 - y2|$$

[4]Distância Manhattan e Geometria Pombalina, Wikipédia. `https://pt.wikipedia.org/wiki/Geometria_pombalina`

No caso desse trabalho, temos como P1 as coordenadas da peça fora do lugar e P2 as coordenadas da respectiva peça no tabuleiro de configuração final. Essa Heurística é linear no número de peças do tabuleiro.

4.2.4. $h'_4(n)$ - Combinação linear entre heurísticas

Nesta heurística é realizada uma combinação linear entre as heurísticas anteriores, onde a soma dos coeficientes p1, p2 e p3 deve ser igual a 1.

$$p1 * h1(n) + p2 * h2(n) + p3 * h3(n)$$

Onde $p1 + p2 + p3 = 1$

A escolha dos pesos é feita com base no seu comportamento em testes anteriores, e os pesos escolhidos foram $p1 = 0,4$, $p2 = 0,2$ e $p3 = 0,4$. A heurística é linear em relação ao tamanho do tabuleiro, porém, como é necessário computar as heurísticas anteriores

4.2.5. $h'_5(n)$ - Valor máximo entre heurísticas

É definida como o valor máximo entre as heurísticas 1, 2 e 3. A complexidade em relação ao tempo mantém-se igual a das heurísticas utilizadas.

4.3. Casos de teste e métricas

Para realizar a comparação de desempenho entre as diferentes heurísticas no algoritmo A* faz-se necessário a utilização de métricas para que possam ser confrontadas. Neste trabalho foram utilizadas as métricas consumo de tempo e consumo de memória.

O conjunto de caso de testes utilizados para execução e aplicação das métricas estão dispostos na Tabela 5.

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

Tabela 5. Casos de testes utilizados

Dado que a máquina utilizada para todos os casos de teste e heurísticas foi o mesmo nota-se que o consumo de tempo, ou seja, o tempo necessário para resolução do

15-puzzle com o algoritmo A*, está ligado intrinsecamente ao resultado informado pela heurística, uma vez que este resultado guiará o caminho a ser tomado na árvore de busca do algoritmo.

O consumo de memória torna-se uma boa métrica uma vez que, a quantidade de níveis gerados na árvore depende do valor retornado pela heurística, desta forma, quanto melhor a heurística menor será a quantidade de níveis gerado e menor será a quantidade de memória necessária para armazenar tal conjunto.

5. Resultados e discussões

Esta sessão será destinada a apresentar os resultados obtidos na execução dos casos de teste para cada heurísticas e anualizá-las utilizando as métricas na sub sessão 4.3.

Se a heurística h satisfizer a condição adicional $h(x) \leq d(x, y) + h(y)$ para cada aresta (x, y) do grafo (onde d denota o comprimento dessa aresta), então h é chamada monótona ou consistente. Com uma heurística consistente, A* garante encontrar um caminho ideal sem processar qualquer nó mais de uma vez e executar A* torna-se equivalente a executar o algoritmo de Dijkstra com o custo reduzido $d'(x, y) = d(x, y) + h(y) - h(x)$. Contanto que o a heurística seja consistente temos a garantia de um menor consumo de memória, dado que cada sucessor é processado uma única vez [5].

5.1. Consumo de memória

A implementação do algoritmos A* utilizada priorizou o consumo de tempo ao invés do consumo de memória, utilizando de estrutura auxiliares que possuíam cópias dos conjuntos A e F, desta forma um grande consumo de memória já era esperado. Os resultado com "−"apresentados na Tabela 6 indicam que a quantidade de memória disponibilizada, cerca de 120 Gb, não foi suficiente para suprir a necessidade do algoritmo e fornecer o resultado.

Teste	$h'_2(n)$	$h'_2(n)$	$h'_3(n)$	$h'_4(n)$	$h'_5(n)$
1	23,27 Mb	315 Mb	9,5 Mb	12,1 Mb	9,6 Mb
2	12,148 Mb	113 Mb	9,5 Mb	10,7 Mb	9,6 Mb
3	241 Mb	1,3 Gb	9,6 Mb	17,892 Mb	9,6 Mb
4	–	–	89,1 Mb	7,098 Gb	124,8 Mb
5	–	–	1,085 Gb	97,1 Gb	1,69 Gb
6	–	–	2,669 Gb	–	8 Gb
7	4,922 Gb	8,796 Gb	30,3 Mb	367 Mb	76 Mb
8	593 Mb	8,231 Gb	10,4 Mb	28 Mb	11 Mb
9	–	–	125,8 Mb	23,7 Gb	194 Mb
10	–	–	–	–	–

Tabela 6. Consumo de memória por teste e heurística

Como é possível observar na tabela anteriormente citada, as heurísticas $h'_3(n)$ e $h'_5(n)$ foram as que produziram os melhores resultados em todos os casos de teste, sendo ainda a $h'_3(n)$ a melhor entre as duas. Isto é possível devido ao fato das heurísticas citadas

[5] A* search algorithm: `https://en.wikipedia.org/wiki/A*_search_algorithm`

serem consistentes, o que reduz o consumo de memória, uma vez que não serão gerados sucessores repetidos.

5.2. Consumo de tempo

Os casos simulados utilizam de diferentes níveis de recursos computacionais, como pode ser observado na subseção anterior, e por esse motivo eles afetam diretamente o comportamento das heurísticas. Portanto, é possível que um caso seja computacionalmente viável utilizando de uma das heurísticas, e computacionalmente inviável utilizando outra heurística. Utilizando da Tabela 7 como referência, podemos verificar o consumo de tempo aferido pelas cinco heurísticas utilizadas, em todos os casos testes, assim como verificar a eficiência de cada heurística.

Teste	$h'_1(n)$	$h'_2(n)$	$h'_3(n)$	$h'_4(n)$	$h'_5(n)$
1	0:01.09	0:21.44	0:00.05	0:00.24	0:00.06
2	0:00.52	0:06.95	0:00.02	0:00.11	0:00.03
3	0:15.36	1:30.23	0:00.02	0:00.65	0:00.04
4	–	–	0:05.67	9:47.30	0:09.98
5	–	–	1:17.86	4:21:43	2:26.80
6	–	–	3:14.37	–	12:10.71
7	5:44.46	10:42.72	0:01.43	0:28.89	0:05.88
8	0:40.68	9:46.86	0:00.08	0:02.14	0:00.16
9	–	–	0:08.25	34:58.53	0:16.40
10	–	–	–	–	–

Tabela 7. Consumo de tempo, h:mm:ss ou mm:ss

É possível observar melhores comportamentos nas heurísticas $h'_3(n)$ (Somatório da Distância Manhattan) e $h'_5(n)$ (Valor máximo entre heurísticas) em comparação com as heurísticas restantes. Além disso, notamos que as heurísticas $h'_1(n)$(Número de peças foras de seu lugar) e $h'_2(n)$(Número de peças fora de ordem na sequência numérica) conseguiram solucionar somente 50% dos casos de teste, e esse comportamento se dá, como visto na subseção anterior, pelo fato de a memória disponibilizada não foi suficiente para suprir a necessidade do algoritmo.

6. Conclusão

O algoritmo A* é um algoritmo de busca de caminhos, que utilizando uma função heurística, consegue se guiar na exploração do grafo. Porém, se utilizarmos de heurísticas que não sejam consistentes, o algoritmo passa a gerar árvores muito grandes, tornando o consumo de recursos grande demais e a execução do algoritmo inviável, como demonstrado em alguns casos na seção 5. Sendo assim, a busca pela eficiência no A* deve se pautar na utilização de heurísticas que possua estratégias para auxiliar na economia de tempo e memória.

Com base nos experimentos realizados, tivemos que a heurística $h'_3(n)$, Distância de Manhattan, foi a heurística que obteve melhores resultados em todos os testes, pois não só teve os menores tempos, mas também foi a que consumiu menor quantidade de

memória nesse conjunto de casos testes. Notamos portanto que o uso de uma heurística consistente, como é o caso da $h'_3(n)$, melhora a eficiência do algoritmo A*.

Para trabalhos futuros, podemos considerar um conjunto de testes mais amplo, onde podemos analisar melhor a heurística $h'_5(n)$, que também se trata de uma heurística consistente, que deveria dar resultados similares ou superiores que a $h'_3(n)$, já que utiliza o valor dominante entre as heurísticas $h'_1(n)$, $h'_2(n)$ e $h'_3(n)$. Além de um conjunto de testes maior e a utilização de um banco de dados de padrões, como demonstrado por [Culberson and Schaeffer 1996], um meio efetivo para diminuir significativamente a árvore de busca gerada pelo A*.

Referências

Culberson, J. and Schaeffer, J. (1994). Efficiently searching the 15-puzzle. Technical report.

Culberson, J. C. and Schaeffer, J. (1996). Searching with pattern databases. Technical report, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1.