Relatório da aplicação do Algoritmo A* para a resolução do jogo Puzzle 15 utilizando diferentes heurísticas

Gabriel T. H. Santos[1], **Gustavo H. F. Cruz**[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá, 12 de abril de 2021

ra107774@uem.br, ra109895@uem.br

Resumo. *Neste relatório será descrito uma análise para resolução do jogo Puzzle 15 utilizando da implementação do Algoritmo A*, focando na análise comportamental do algoritmo para cinco propostas de heurísticas, foram utilizados dez casos de teste e feitas comparações dos resultados obtidos através de duas abordagens diferentes para o problema.*

1. Introdução

Este trabalho visa analisar e contemplar uma visão sobre a aplicação do algoritmo A* para a resolução do jogo *Puzzle* 15, utilizando de cinco heurísticas e as comparando em busca de uma conclusão da mais efetiva.

2. O Problema

O jogo *Puzzle* 15 ou Jogo do 15, em português, é um quebra-cabeça em tabuleiro, contendo quinze peças numeradas de um a quinze e um espaço vazio. O jogo consiste na organização destas em suas respectivas posições, podendo variar a ordenação por coluna ou linha. O problema surge quando se tem as peças bagunçadas e se deve ir trocando de lugar o espaço em branco com cada peça adjacente em busca do objetivo final. Para o trabalho, desejamos saber o número mínimo de movimentos partindo de qualquer tabuleiro, desde que esse seja válido, para se chegar ao tabuleiro final.

3. Algoritmo A*

O algoritmo A* é um algoritmo da área de grafos usado para realizar a busca do **melhor** caminho entre um vértice de entrada, o estado inicial do problema, até outro(s) vértice(s), sendo o(s) estado(s) final(is) do problema. O diferencial dele, quando comparado à algoritmos de busca em grafos como o *Dijkstra* e/ou *Floyd Warshell*, é que o caminho entre ambos não é conhecido de antemão e o A* deve explorar os possíveis caminhos toda vez que encontra um novo vértice. A busca entre **todas** as possibilidades de caminhos é uma opção nada desejável, computacionalmente falando, uma vez que a quantidade de combinações possíveis pode impactar de maneira exponencial no tempo da solução, ou seja, para cada vez que um novo ramo é explorado, um novo nível da árvore é gerado e para explorar o nível todo, dependendo da aplicação do problema, pode ser uma tarefa quase impossível (ainda computacionalmente falando). Por este motivo que o algoritmo faz uso de heurísticas e ferramentas para 'podar' ramos que não são promissores, isto é, ramos que se afastam da solução final desejada.

Existem dois cenários possíveis para a aplicação das heurísticas no algoritmo. A primeira é a possibilidade de uma heurística admissível, isso significa que para todo ramo

que o algoritmo optar por explorar será um ramo que melhora a sua solução atual, ou em outras palavras, ele pega um caminho cujo valor é **menor ou igual** ao ideal (neste caso, a quantidade de movimentos mínimos reais para chegar na resposta), se aproximando assim do estado final. O segundo cenário possível, é quando temos uma heurística não admissível e, neste caso, o ramo que o algoritmo vai explorar não tem garantia alguma de estar mais próximo da solução a cada nível explorado.

3.1. Adaptação do Algoritmo ao Problema

O problema no qual o A* é aplicado, e o foco deste trabalho, é o jogo *Puzzle* 15. Quando aplicado no jogo, o ambiente de aplicação do algoritmo é mudado e alterações/melhorias no código podem ser realizadas. O primeiro ponto a ser levantado é que no caso do jogo, todas as heurísticas são admissíveis e o algoritmo não precisa considerar a possibilidade de heurísticas não admissíveis, o que nos permite alterar partes do código e, consequentemente, melhorar o tempo de execução. Além disso, para a aplicação do algoritmo no jogo, os vértices se tornam o estado do tabuleiro e os sucessores são gerados a partir da possibilidade de movimentos da peça branca do tabuleiro (representada com o número zero para a implementação adotada).

3.2. Heurísticas Diferentes

Para a solução do problema foram implementadas três lógicas de heurísticas diferentes, o que resultou em três diferentes heurísticas 'principais'. Além delas, foram geradas outras duas, a partir da combinação das respostas geradas. A problemática de diferentes heurísticas busca fornecer diferentes métodos para o algoritmo chegar a solução.

3.3. Heurística 1

A primeira heurística analisa cada posição do estado atual (tabuleiro atual) e conta quantas destas estão fora da sua posição ideal, ou seja, diverge da peça na mesma posição no tabuleiro de resposta.

3.4. Heurística 2

A segunda heurística conta quantas posições estão fora da sua posição ideal, numericamente falando, analisando então sequencialmente, por exemplo, um tabuleiro que tiver todas as suas peças movimentadas uma peça para frente (a primeira no lugar da segunda, a segunda no lugar da terceira...) teria apenas uma peça considerada como errada (neste caso, o zero que seria a primeira peça e não se esperaria nada na frente dele).

3.5. Heurística 3

A terceira heurística é a aplicação do conceito da distância *Manhattan* no tabuleiro do jogo, ou seja, para cada peça fora do lugar é contado quantos movimentos são necessários para chegar ao seu local correto, considerando que o caminho esteja livre.

3.6. Heurística 4

A quarta heurística é as três anteriores com pesos aplicados a elas, onde a soma final destes pesos é igual a um. Os pesos usados na aplicação implementada foram 0.09, 0.16 e 0.75 para, respectivamente, as heurísticas um, dois e três.

3.7. Heurística 5

A quinta e última heurística é o maior valor dentre os gerados pelas heurísticas um, dois e três para o estado sendo analisado.

4. Resultados

Os resultados foram gerados a partir de duas implementações/abordagens diferentes e são comparadas e analisadas a seguir.

4.1. Abordagens

Foram realizadas duas abordagens diferentes na implementação do A*, ambas na linguagem *C++*.

A primeira abordagem foi considerar os conjuntos dos vértices abertos e fechados como uma árvore *Heap* mínima. A busca pelo vértice de custo mínimo é constante, porém, a busca de um valor dentro do conjunto, é O(n), onde n é o tamanho do conjunto dos abertos. O principal aspecto desta abordagem é a ênfase no processamento, deixando o maior trabalho do algoritmo a cargo do processador.

Na segunda abordagem é implementada, assim como a primeira, uma árvore *Heap* mínima para o conjunto dos abertos, porém em conjunto com uma tabela *Hash* para guardar a existência de um vértice. O principal diferencial é a troca entre tempo de processamento por memória, uma vez que, com a tabela *Hash*, a checagem pela existência de um vértice é constante.

4.2. Configurações Utilizadas

Configurações da máquina

S.O	Debian 10 (buster)
Processador	Ryzen 5-3500u CPU @ 2,1Ghz - 3,7 GHz
Memória RAM	8 GB's
Memória SSD	280 GB's
Placa de Vídeo	Integrada AMD Radeon RX Vega 8

4.3. Tabela Comparativa de Tempos e Memória

Foram usadas as seguintes entradas testes para o algoritmo:

C1: 0 2 9 13 3 1 5 14 4 7 6 10 8 11 12 15
C2: 3 2 1 9 0 5 6 13 4 7 10 14 8 12 15 11
C3: 2 1 9 13 3 5 10 14 4 6 11 15 7 8 12 0
C4: 9 13 10 0 5 2 6 14 1 7 11 15 3 4 8 12
C5: 4 3 2 1 8 10 11 5 12 6 0 9 15 7 14 13
C6: 9 13 14 15 5 6 10 8 0 1 11 12 7 2 3 4
C7: 10 6 2 1 7 13 9 5 0 15 14 12 11 3 4 8
C8: 6 2 1 5 4 10 13 9 0 8 3 7 12 15 11 14
C9: 10 13 15 0 5 9 14 11 1 2 6 7 3 4 8 12
C10: 5 9 13 14 1 6 7 10 11 15 12 0 8 2 3 4

Nas tabelas seguintes são descritos os resultados obtidos, observando a abordagem sem o uso de *Hash* como a abordagem 1 e com o uso de *Hash* como a abordagem

2 e o custo de tempo e memória para cada caso em cada uma das heurísticas. Casos rodados sem êxito foram adicionados à tabela como '+'. Os resultados parcialmente calculados, ou seja, que chegaram próximo a resposta, foram colocados com os símbolos de desigualdade.

Tabela 1. Abordagem 1 - H1

Caso	Tempo	Memória
1	0,187s	320,0 KiB
2	0,104s	240,0 KiB
3	0.006s	144,0 KiB
4	0,678s	728,0 KiB
5	+	+
6	+	+
7	+	+
8	$\geq$ 2d	$\pm$ 144,0 MiB
9	$\geq$ 2d	$\pm$ 144,0 MiB
10	$\geq$ 2d	$\pm$ 144,0 MiB

Tabela 2. Abordagem 2 - H1

Caso	Tempo	Memória
1	0,023s	304 Kib
2	0,023s	300 Kib
3	0,004s	144 KiB
4	0,073s	832 kib
5	$\geq$20m	$\geq$ 8 GiB
6	0m43,699s	291,2 MiB
7	$\geq$20m	$\geq$ 8 GiB
8	0m57,112s	425 MiB
9	0m1,903s	13,2 MiB
10	0m15,691s	96,3 MiB

Tabela 3. Abordagem 1 - H2

Caso	Tempo	Memória
1	0,348s	470 Kib
2	0m4,565s	860 Kib
3	0,012s	144 Kib
4	0m15,945s	1,7 Mib
5	+	+
6	+	+
7	+	+
8	+	+
9	+	+
10	+	+

Tabela 4. Abordagem 2 - H2

Caso	Tempo	Memória
1	0,062s	578,0 KiB
2	0,131s	1,1 MiB
3	0,007s	144,0 KiB
4	0,356s	3,0 MiB
5	$\geq$15m	$\geq$ 8 GiB
6	10m14,931s	3,2 GiB
7	$\geq$15m	$\geq$ 8 GiB
8	3m22,188s	1,3 GiB
9	0m14,465s	90,9 MiB
10	1m34,058s	628,6 MiB

Tabela 5. Abordagem 1 - H3

Caso	Tempo	Memória
1	0.015s	148,0 KiB
2	0,004s	144,0 KiB
3	0,007s	144,0 KiB
4	0,035s	248,0 KiB
5	0,019s	148,0 KiB
6	0,042s	248,0 KiB
7	0,484s	472,0 KiB
8	0,316s	452,0 KiB
9	0,024s	248,0 KiB
10	0,164s	368,0 KiB

Tabela 6. Abordagem 2 - H3

Caso	Tempo	Memória
1	0,004s	148,0 KiB
2	0,006s	144,0 KiB
3	0,004s	144,0 KiB
4	0,008s	144,0 KiB
5	0,134s	1,1 MiB
6	0,086s	832,0 KiB
7	0,490s	4,0 MiB
8	0,241s	2,0 MiB
9	0,020s	308,0 KiB
10	0,196	1,6 MiB

Tabela 7. Abordagem 1 - H4

Caso	Tempo	Memória
1	0,061s	252,0 KiB
2	0,097s	300,0 KiB
3	0,013s	144,0 KiB
4	0,016s	144,0 KiB
5	0,022s	244,0 KiB
6	0m4,314s	1016,0 KiB
7	0m15,014s	1,8 MiB
8	0,036s	688,0 KiB
9	0,391s	464,0 KiB
10	0m57,582s	4,1 MiB

Tabela 8. Abordagem 2 - H4

Caso	Tempo	Memória
1	0,007s	144,0 KiB
2	0,005s	144,0 KiB
3	0,004s	144,0 KiB
4	0,015s	144,0 KiB
5	0m1,376s	10,8 MiB
6	0,195s	1,8 MiB
7	0m3,608s	24,1 MiB
8	0,459s	3,8 MiB
9	0,017s	304,0 KiB
10	0,353s	3,1 MiB

Tabela 9. Abordagem 1 - H5

Caso	Tempo	Memória
1	0,009s	144,0 KiB
2	0,259s	392,0 KiB
3	0,007s	148,0 KiB
4	0,013s	144,0 KiB
5	0,014s	148,0 KiB
6	0m14,183s	1,9 MiB
7	2m20,417s	5,9 MiB
8	0m3,637s	984 KiB
9	0,207s	456,0 KiB
10	0m20,119s	2,2 MiB

Tabela 10. Abordagem 2 - H5

Caso	Tempo	Memória
1	0,012s	144,0 KiB
2	0,013s	148,0 KiB
3	0,005s	144,0 KiB
4	0,041s	304,0 KiB
5	0,631s	4,6 MiB
6	0,393s	2,7 MiB
7	0m11,603s	66,5 MiB
8	0,635s	4,5 MiB
9	0,035s	300,0 KiB
10	0m2,230s	12,9 MiB

4.4. Análise entre as heurísticas

Quando comparamos as heurísticas entre si, podemos observar que entre as três 'principais', a terceira é a que mais se sai bem, conseguindo tempos e custos de memória e execução muito mais baixos que as demais. A heurística dois, por outro lado, é a que tem os piores resultados. O fato da heurística três ser melhor muito se deve por conta dela formular uma resposta que considera cada posição individualmente do tabuleiro. A heurística dois entretanto é o oposto, chegando a casos extremos onde um caso como o apresentado anteriormente (cada número deslocado uma unidade para frente) tem um valor muito próximo da resposta desejada, quando na realidade está longe dela. A heurística um pode ser vista como o meio termo, ela considera cada posição individualmente, mas não faz operações ou cálculos extras que deem uma melhor precisão para o resultado final. Em relação às outras duas heurísticas, enquanto a heurística quatro tende a ser mais rápida quando se atribui mais peso a heurística três mais distribuídas para a um e dois, a heurística cinco mantém um bom desempenho por conta de também considerar muito as respostas provindas da heurística três, uma vez que é ela que provê, grande parte das vezes, o maior resultado entre as heurísticas.

4.5. Análise entre as abordagens

Podemos perceber que a abordagem dois chega aos resultados muito mais rapidamente que a um a medida que se aumenta as dificuldades, porém o consumo da memória do sistema cresce abruptamente para estes casos maiores. Em relação às limitações das abordagens, temos que, enquanto a abordagem um não conseguia processar casos 'difíceis' por conta do tempo de execução, a abordagem dois parava por conta da falta de memória principal.

5. Conclusão

Apesar de não ser uma regra, nota-se que, quanto melhor uma heurística diferenciar um 'estado bom' de um 'estado ruim', mais facilita o algoritmo a achar o caminho correto a ser seguido, bem como, quanto melhor for uma heurística, mais complexa tende a ser sua implementação. Em termos gerais, a dificuldade de implementar uma heurística mais complexa, para obter um ganho de desempenho melhor, vale a pena.

Além disso, notamos que verificar se um estado novo já existe ou não dentro dos abertos influencia diretamente no tempo de processamento. A ausência desta verificação ocasiona a verificação demasiada de estados não promissores, o que impacta negativamente no tempo de processamento de cada vértice. Porém, com a verificação adicionada, a velocidade do processamento é muito impactada e dependendo da abordagem utilizada pode gerar um gargalo no algoritmo. Vale ressaltar ainda que, em ambas as abordagens foram pensados maneiras mais eficientes de fazer as verificações, sendo que, na abordagem um, se aproveita de duas verificações em uma mesma busca, ou seja, ao passar pelo conjunto dos abertos, o primeiro *if* verifica o tabuleiro de cada e se este for igual ao do 'm' é setado uma variável auxiliar para facilitar para o terceiro *if*, em sequência verifica se o caminho deste é menor e, caso verdadeiro, é atualizado com o novo candidato, não necessitando remover e depois inserir logo em seguida. Na abordagem dois a verificação de um membro no conjunto é feita através do uso do *Hash* e por isso o tempo de acesso é constante.

Referências

[1] Constantino, Ademir A.. Algoritmos de Busca na Resolução de Problemas. 2020.

[2] Explicação do Algoritmo A* (A Star). Direção de Carlos Mingoto. 2017. Vídeo (23min). Disponível em: https://www.youtube.com/watch?v=o5_mqZKhTvw. Acesso em: 1 abr. 2021.

[3] A* (A Star) Search Algorithm - Computerphile. Direção de Computerphile. 2017. Vídeo (14min). Disponível em: https://www.youtube.com/watch?v=ySN5Wnu88nE. Acesso em: 31 mar. 2021.

[4] Prado, Sergio . Otimização de código em Linguagem C – Parte 1. Disponível em: https://sergioprado.org/otimizacao-de-codigo-em-linguagem-c-parte-1/. Acesso em: 1 abr. 2021.

[5] Prado, Sergio . Otimização de código em Linguagem C – Parte 2. Disponível em: https://sergioprado.org/otimizacao-de-codigo-em-linguagem-c-parte-2/. Acesso em: 1 abr. 2021.