UNIVERSIDADE FEDERAL DE PELOTAS (UFPEL)
CENTRO DE DESENVOLVIMENTO TECNOLÓGICO (CDTEC)

MESTRADO EM COMPUTAÇÃO

FUNDAMENTOS DE INTELIGÊNCIA ARTIFICIAL (PPGC)

Relatório comparativo entre algoritmos de busca aplicados no problema do Quebra-cabeça Deslizante (8-puzzle)

Yan Ballinhas Soares

3 de abril de 2019

1 Introdução

Algoritmos de busca são utilizados em diversos casos onde é necessário encontrar uma solução de forma automatizada, pesquisando dentre uma série de alternativas até encontrar a solução do problema proposto. Exemplos de cenários onde esse tipo de algoritmo é aplicado são: jogos, pesquisa de rotas, escalonamento de tarefas e outros problemas computacionais.

No desenvolvimento da cadeira de Fundamentos de Inteligência Artificial do PPGC da UFPel, foi proposto o desenvolvimento de 4 algoritmos de busca, sendo eles: busca em largura, busca em profundidade, busca em profundidade iterativa e o algoritmo A*. Os mesmos foram aplicados no problema do Quebra-Cabeça Deslizante (8-Puzzle), com diferentes tamanhos de tabuleiro e níveis de embaralhamento.

O repositório com o código desenvolvido, os scripts de teste e os resultados utilizados neste relatório pode ser acessado aqui

Na Seção 2 é descrito brevemente o funcionamento de cada algoritmo e os resultados individuais obtidos com cada um. Na seção 3, é feita uma comparação de tempo de execução e uso de memória entre os 4 algoritmos com entradas padronizadas e a discussão de seus resultados, além das dificuldades encontradas para a implementação dos mesmos.

2 Algoritmos de busca

Essa seção busca explicar brevemente o funcionamento de cada algoritmo de busca e os resultados obtidos. Para a aplicação dos algoritmos, é considerada uma estrutura de árvore, onde a raíz é o tabuleiro embaralhado e cada nodo tem como filho os possíveis movimentos a serem realizados com aquela configuração de tabuleiro. Além disso, os resultados apresentados nessa Seção são baseados em 3 execuções distintas de cada algoritmo, com diferentes níveis de embaralhamento e diferentes tamanhos de tabela. Todas as tabelas são embaralhadas aleatoriamente, por isso alguns resultados são discrepantes em relação aos outros.

Por questões práticas, caso o algoritmo demore mais de 10 minutos para achar uma solução, a solução é definida como não encontrada (valores nulos nas tabelas).

2.1 Busca em Largura (BFS)

A Busca em Largura (Breadth-First Search - BFS) explora os nodos da árvore por nível (horizontalmente). Ou seja, explora todos os nodos de um nível, da esquerda para a direita, antes de passar para o próximo nível (filhos dos nodos já explorados). Um exemplo de funcionamento se encontra na Figura 1.

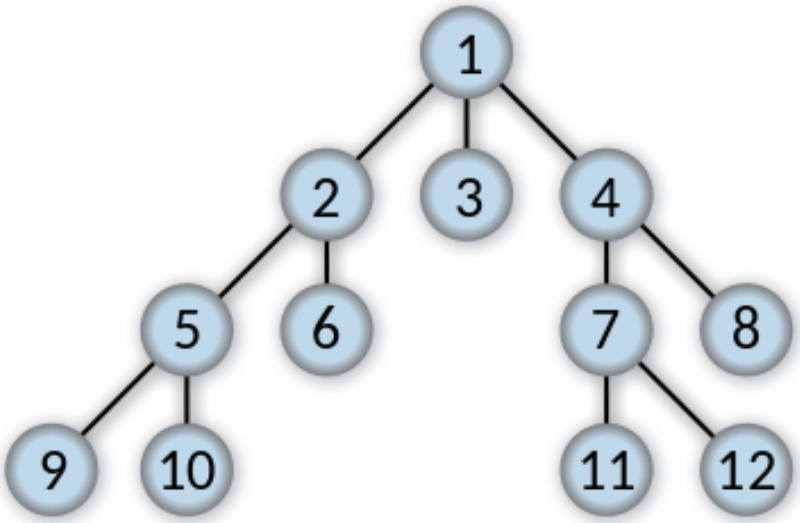

Figura 1: Exemplo de funcionamento do Algoritmo BFS

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	0.001s	0.002s	0.001s
100 vezes	0.03s	4.679s	72.159s
1000 vezes	374.392s	82.154s	90.485s
10000 vezes	297.937s	414.172s	277.110s

Tabela 1: Tempo de execução da Busca em Largura com Tabuleiros 3x3

.

Nível de Embaralhamento	1ª Execução	2ª Execução	3ª Execução
10 vezes	7	21	9
100 vezes	21	31441	342095
1000 vezes	776751	380651	409679
10000 vezes	657149	795211	626011

Tabela 2: Número de nodos visitados na Busca em Largura com Tabuleiros 3x3

.

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.014s	0.001s	0.009s
100 vezes	-	-	-
1000 vezes	-	-	-
10000 vezes	-	-	-

Tabela 3: Tempo de execução da Busca em Largura com Tabuleiros 4x4

.

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	48	4	40
100 vezes	719567	738856	735458
1000 vezes	702638	643888	683280
10000 vezes	668018	746456	756740

Tabela 4: Número de nodos visitados na Busca em Largura com Tabuleiros 4x4

.

2.2 Busca em Profundidade (DFS)

A Busca em Profundidade (Depth-First Search - DFS) explora os nodos da árvore até o final de cada ramo (verticalmente). Ou seja, explora todos os nodos, da raíz até as folhas antes de voltar e testar os outros nodos (até encontrar uma folha e voltar novamente e assim repetir o processo, até que

toda a árvore seja explorada). Um exemplo de funcionamento se encontra na Figura 2.

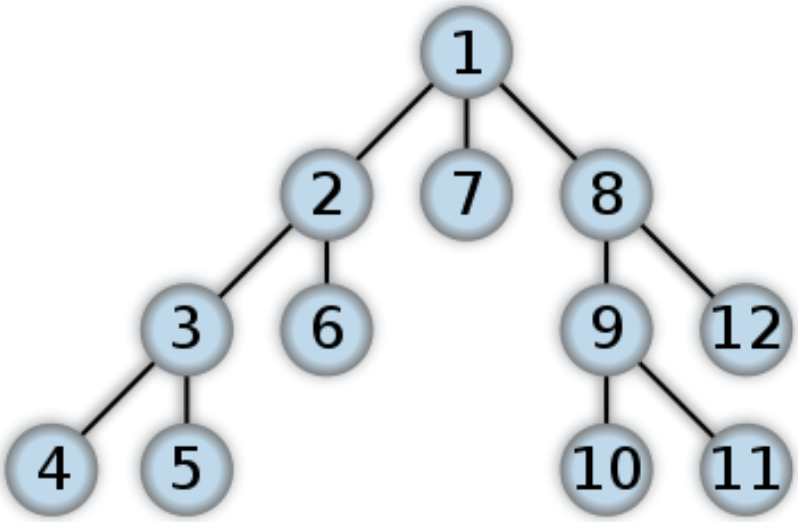

Figura 2: Exemplo de funcionamento do Algoritmo DFS

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	0.001s	0.0001s	0.004s
100 vezes	42.981s	52.885s	-
1000 vezes	-	26.575s	101.03s
10000 vezes	-	1.374s	3.284s

Tabela 5: Tempo de execução da Busca em Profundidade com Tabuleiros 3x3

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	11	1	33
100 vezes	240363	287489	994454
1000 vezes	1007864	165593	455901
10000 vezes	963933	7605	19511

Tabela 6: Número de nodos visitados na Busca em Profundidade com Tabuleiros 3x3

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.007s	0.069s	0.011s
100 vezes	-	-	75.567s
1000 vezes	-	-	-
10000 vezes	-	-	-

Tabela 7: Tempo de execução da Busca em Profundidade com Tabuleiros 4x4

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	17	161	34
100 vezes	704840	752559	257488
1000 vezes	754437	752301	757330
10000 vezes	690598	715758	690802

Tabela 8: Número de nodos visitados na Busca em Profundidade com Tabuleiros 4x4

2.3 Busca em Profundidade Iterativa

Este algoritmo complementa o algoritmo anterior. O algoritmo anterior poderia vir a ocupar muito espaço de memória, visando isso, esse algoritmo vai iterando a altura máxima da árvore a ser explorada. Quando toda a árvore é explorada até o nível determinado, o nível é iterado e a árvore gerada até o nível seguinte. Acaba sendo mais lento devido ao processo de ter que gerar a árvore repetidamente, porém soluciona o problema de muitos nodos abertos em memória ao mesmo tempo.

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.002s	0.002s	0.002s
100 vezes	23.004s	1.87s	39.517s
1000 vezes	18.261s	234.729s	116.03s
10000 vezes	44.067s	-	237.96s

Tabela 9: Tempo de execução da Busca em Profundidade Iterativa com Tabuleiros 3x3

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	13	17	15
100 vezes	56295	7697	114835
1000 vezes	40129	529158	336321
10000 vezes	144169	626976	510175

Tabela 10: Número de nodos visitados na Busca em Profundidade Iterativa com Tabuleiros 3x3

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.0006s	0.0002s	0.0031s
100 vezes	0.102s	-	-
1000 vezes	-	-	-
10000 vezes	-	-	-

Tabela 11: Tempo de execução da Busca em Profundidade Iterativa com Tabuleiros 4x4

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	4	1	15
100 vezes	399	189446	163179
1000 vezes	614248	216921	519567
10000 vezes	540140	164482	187758

Tabela 12: Número de nodos visitados na Busca em Profundidade Iterativa com Tabuleiros 4x4

2.4 Algoritmo A*

Diferentemente dos algoritmos anteriores, que são classificados como algoritmos sem informação, o algoritmo A* utiliza de heurísticas para decisão de qual nodo visitar a seguir. Neste trabalho, foram utilizadas 2 heurísticas diferentes: a heurística 1 considera o número de peças que estão fora do lugar e a heurística 2 calcula a distância de Manhattan no tabuleiro (o quanto as peças estão deslocadas em relação ao seu local correto). Essas heurísticas são consideradas h(n), sendo n o nodo em questão.

A decisão de qual nodo visitar a seguir é feita através do menor resultado de f(n) entre os nodos visitados naquele momento, que é dado por f(n) = h(n) + g(n), sendo g(n) o nível onde o nodo se encontra (raíz é considerada nível 0, seus filhos nível 1 e assim por diante).

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	0.0004s	0.0001s	0.00102s
100 vezes	0.011s	0.038s	0.102s
1000 vezes	-	0.965s	35.716s
10000 vezes	-	265.618s	19.962s

Tabela 13: Tempo de execução da Busca com Algoritmo A* e Heurística 1 em Tabuleiros 3x3

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	2	1	5
100 vezes	82	240	526
1000 vezes	60607	2326	15382
10000 vezes	60768	41026	11727

Tabela 14: Número de nodos visitados com o Algoritmo A* e Heurística 1 em Tabuleiros 3x3

Nível de Embaralhamento	**1ª execução**	**2ª execução**	**3ª execução**
10 vezes	0.001s	0.0006s	0.001s
100 vezes	18.385s	-	0.498s
1000 vezes	-	-	-
10000 vezes	-	-	-

Tabela 15: Tempo de execução da Busca com Algoritmo A* e Heurística 1 em Tabuleiros 4x4

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	5	2	6
100 vezes	8744	48251	1111
1000 vezes	43577	46219	42025
10000 vezes	42459	43966	46354

Tabela 16: Número de nodos visitados com o Algoritmo A* e Heurística 1 em Tabuleiros 4x4

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.0012s	0.0006s	0.001s
100 vezes	0.863s	0.64s	0.013s
1000 vezes	3.064s	0.692s	2.233s
10000 vezes	0.156s	4.069s	1.974s

Tabela 17: Tempo de execução da Busca com Algoritmo A* e Heurística 2 em Tabuleiros 3x3

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	6	3	9
100 vezes	2008	1703	81
1000 vezes	4226	1783	3546
10000 vezes	661	4898	3281

Tabela 18: Número de nodos visitados com o Algoritmo A* e Heurística 2 em Tabuleiros 3x3

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	0.001s	0.001s	0.001s
100 vezes	0.993s	9.326s	-
1000 vezes	-	-	-
10000 vezes	-	-	-

Tabela 19: Tempo de execução da Busca com Algoritmo A* e Heurística 2 em Tabuleiros 4x4

Nível de Embaralhamento	1ª execução	2ª execução	3ª execução
10 vezes	5	6	5
100 vezes	1711	6037	44212
1000 vezes	43055	44563	47012
10000 vezes	46260	45455	45915

Tabela 20: úmero de nodos visitados com o Algoritmo A* e Heurística 2 em Tabuleiros 4x4

3 Comparação entre os algoritmos

Para os resultados comparativos, foi gerada uma tabela 3x3 solucionável por todos os algoritmos para cada nível de embaralhamento. As tabelas utilizadas embaralhadas 200, 400, 600 800 e 1000 vezes podem ser vistas, respectivamente, abaixo:

$$\begin{bmatrix} 3 & 5 & 2 \\ 4 & 1 & 8 \\ 7 & 6 & \end{bmatrix}, \begin{bmatrix} 7 & 6 & 1 \\ 5 & 3 & 2 \\ 8 & & 4 \end{bmatrix}, \begin{bmatrix} 4 & 8 & 2 \\ & 6 & 7 \\ 5 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 & 6 \\ 3 & 5 & 8 \\ 2 & 1 & \end{bmatrix}, \begin{bmatrix} 8 & 1 & 6 \\ 3 & 7 & 2 \\ 5 & 4 & \end{bmatrix}$$

Nível de Embaralhamento	BFS	DFS	DFS Iterativo	A* com Heurística 1	A* com Heurística 2
200 vezes	10.88s	11.26s	16.14s	1.19s	0.13s
400 vezes	47.55s	45.81s	102.07s	14.58s	0.15s
600 vezes	69.18s	67.35s	114.76s	15.2s	0.2s
800 vezes	98.08s	106.83s	176.81s	31.65s	0.42s
1000 vezes	110.94s	129.27s	178.34s	40.95s	0.27s

Tabela 21: Tempo de execução para os diferentes algoritmos de busca

Nível de Embaralhamento	BFS	DFS	DFS Iterativo	A* com Heurística 1	A* com Heurística 2
200 vezes	69905	69905	69905 (18 níveis)	2140	567
400 vezes	253428	253428	253428 (21 níveis)	9695	656
600 vezes	310148	310148	310148 (21 níveis)	9743	799
800 vezes	401829	401829	401829 (22 níveis)	13889	1363
1000 vezes	470511	470511	470511 (22 níveis)	15362	999

Tabela 22: Número de nodos visitados pelos diferentes algoritmos de busca

Como é possível notar com os resultados apresentados, o algoritmo mais rápido e que consome menos memória é o A* (considerado um algoritmo otimizado pelo uso de informação, diferentemente dos anteriores). Ainda mais, é possível notar que o uso da heurística da distância de Manhattan se prova muito mais eficiente nesse caso que a heurística do número de peças fora do lugar.

Dentre os 3 outros algoritmos, o mais eficiente ainda é o DFS (Busca em Profundidade), em questão de tempo. Em todos os casos (nesses 3 algoritmos), a solução encontrada (número de nodos visitados) acabou sendo a mesma, porém, o DFS Iterativo encontraria outra solução caso ela aparecesse primeiro, consumindo assim ainda menos memória que os outros.

O Algoritmo A*, apesar de mais complexo de ser implementado, necessitando informações adicionais a serem acrescentadas em cada nodo para que seja possivel calcular a f(n) de cada um deles, acaba sendo muito superior em tempo de execução e uso de memória que os outros.

A maior dificuldade encontrada na implementação desses algoritmos foi, primeiramente, a adaptação a linguagem Python (na qual não era acostumado a programar e nunca havia programado orientado a objetos). Além disso, levou um tempo para conseguir pensar na lógica da implementação da ordem de exploração da árvore por cada algoritmo. Por fim, decidi utilizar um Array que ia armazenando os nodos, que iam sendo visitados e gerando seus filhos, que eram então anexados ao mesmo Array. As diferentes lógicas de utilização desse Array podem ser observadas no código.

Por fim, a execução deste trabalho me permitiu aprimorar um pouco a capacidade de programação com uma nova linguagem, além de exigir um raciocínio lógico para transpor o que foi dito em aula (teoria) para código (prática), capacidade necessária a ser desenvolvida por qualquer programador.