DCC028 Inteligência Artificial - TP1

Vinícius Teixeira Mello

May 7, 2019

1 Introdução

O objetivo do trabalho é a aplicação de diversos algoritmos de busca para resolver instâncias do jogo conhecido como 8-puzzle. O jogo consiste em um "tabuleiro" em grade três por três contendo números de 1 a 8 e um espaço vazio. O objetivo é colocar o tabuleiro na configuração da imagem abaixo.

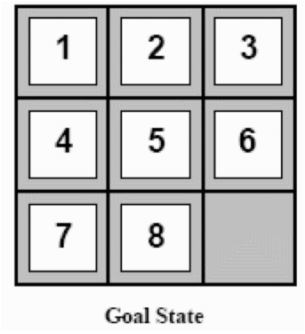

Figure 1: Estado objetivo

Os movimento permitidos são trocar o espaço vazio de lugar com qualquer uma de suas posições adjacentes. Vale a pena observar que esse jogo pode ser generalizado para grades maiores (4x4, 5x5, ...) mais por ser um problema NP-Difícil, a busca em arvore fica rapidamente inviável.

1.1 Modelagem do Problema

Os algorítimos descritos desse relatório foram implementados usando a linguagem C++. as Estruturas Dados básicas utilizadas nos algorítimos foram:

- **State:** Representa uma configuração do jogo através de um vetor de $n * *2$ posições, onde o -1 corresponde ao vazio. Um estado também armazena um vetor com todos os movimentos feitos para se chegar nele. Tem como métodos a geração de estados vizinhos, a conferencia se o estado é terminal e o calculo de heurísticas para algorítimos com informação.
- **Game:** Representa uma instância a ser solucionada e é composto basicamente por um State representando a configuração inicial. Todos os algorítimos apresentados aqui são implementados como métodos dessa estrutura.
- **Solution:** Estrutura que armazena informações sobre a solução do problema. Contém informações de tempo de execução, estados visitados e se o estado terminal foi encontrado ou não. Todos os Algorítimos de resolução retornam uma estrutura dessas.
- **TabuList:** Estrutura auxiliar que armazena todos os estados já visitados em uma execução do algorítimo. Implementada utilizando estrutura em árvore que permite conferir se um estado e repetido em $O(n)$ onde n é o tamanho do vetor que armazena a configuração de um estado. Também armazena nos nós folhas em quantos passos um determinado estado foi encontrado para garantir solução ótima nos algorítimos baseados em busca em profundidade. Utiliza uma estrutura auxiliar para representar os nós da árvore (Node).

1.2 Algorítimos

Os três primeiros algorítimos implementados foram os chamados "busca sem informação". A característica principal desse tipo de algorítimo é a utilização de busca em árvore de possíveis estados geradas a partir do estado inicial e das possíveis ações permitidas no jogo. Essa busca é chamada de sem informação pois não utiliza nenhuma estimativa para a qualidade de um estado em relação a sua sua "proximidade" do estado terminal. No caso do 8-puzzle os algorítimos *Uniform-cost search* e *Breadth-first search* tem comportamento igual, já que o custo corrente de um estado será sempre igual a sua profundidade na árvore de busca. O que diferencia as duas abordagens é a estrutura utilizada para armazenar os nós a serem visitados, enquanto o *Uniform-cost search* utiliza um fila de prioridade com base no custo corrente (numero de movimentos até o estado), o *Breadth-first search* usa uma fila FIFO. Além desses dois, também foi implementado o *Iterative deepening search* que faz buscas em profundidade (utilizando uma pilha para armazenar os nós a serem visitados) com profundidade limitada. A cada iteração o algorítimo reinicia a busca permitindo uma maior profundidade até que a solução seja encontrada. Em todos os casos um estrutura em árvore (TabuList) foi utilizada para armazenar estados já visitados e evitar a abertura exagerada de nos na árvore de busca. No caso do *Iterative deepening search* a árvore também armazena o número de passo necessários para chegar em um estado, e caso um caminho menor para aquele estado seja encontrado esse valor é atualizado e o algorítimo leva essa nova rota em consideração.

A segunda classe de algorítimos utilizados nesse trabalho foi a "busca com informação". Esses algorítimos são análogos ao sem informação, porém fazem uso também de uma heurística que irá estimar o quão próximo um determinado estado está do estado terminal. O primeiro algoritmo dessa classe é o *Greedy best-first search* que visita os estados baseados exclusivamente no valor da heurística. O *A* seach* por outro lado usa a informação do custo corrente somado a estimativa da heurística para definir a ordem de visitação. Vale ressaltar que o *A* search* utiliza os critérios do *Greedy best-first search* e *Uniform-cost search* somados para escolha da ordem de visitação. No caso da heurística utilizada ser admissível, o *A* search* tem garantia de otimalidade, enquanto o *Greedy best-first search* não tem a mesma garantia.

Por fim um algorítimo de "Busca local" denominado *Hill Climbing* foi implementado. Esse algorítimo consiste em se mover sempre para o melhor vizinho do estado corrente, desde que o novo estado tenha um valor da heurística melhor ou igual ao do estado corrente. Nesse caso não se tem garantia que uma solução será encontrada, mas em compensação não é necessário armazenar a árvore de busca, e a um resultado consegue ser obtido rapidamente.

1.3 Heurísticas Utilizadas

Para implementação das buscas com informação e busca local duas diferentes heurísticas foram utilizadas. A primeira se baseia em uma distância simples entre o estado atual é o estado terminal calculada com base no número de peças fora da posição correta. A segunda heurística é uma distância entre o estado atual e o estado terminal calculada com base na distância de manhattan. No contexto de 8-puzzle essa distância pode ser vista com ou número mínimo de movimentos necessários para colocar uma peça na posição correta (considerando que uma peça pode movimentar para qualquer posição adjacente). Ambas as heurísticas são admissíveis pois são limites inferiores para o verdadeiro valor do custo para se chegar no estado terminal. É fácil provar que a distância simples sempre será menor ou igual a distância de manhattan, e essa por sua vez será sempre menor que o número de movimento necessários para chegar ao estado terminal, pois ela considera o custo de uma versão relaxada do problema original.

1.4 Experimentos computacionais

Para testar os algorítimos da prática instâncias fornecidas no problema foram resolvidas usando todos os algoritmos e informações sobre número de nós visitados, número de ações até solução e tempo foram armazenadas e abaixo estão presentes gráficos e tabelas mostrando o resultado

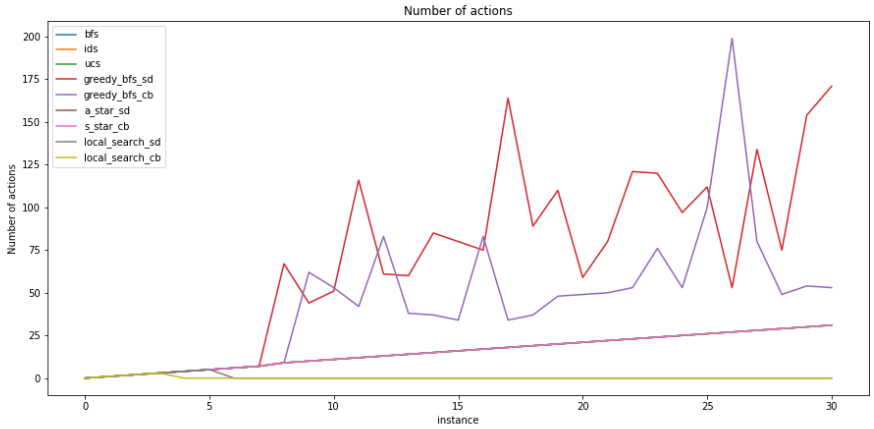

Figure 2: Ações até solução

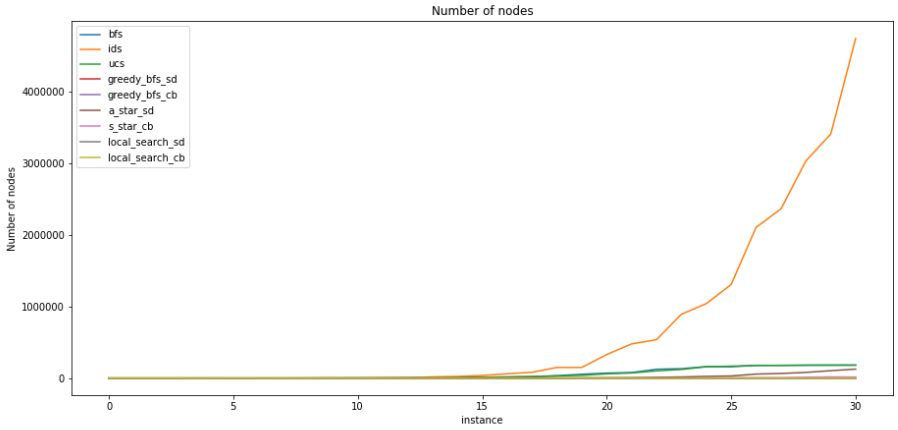

Figure 3: Nós visitados

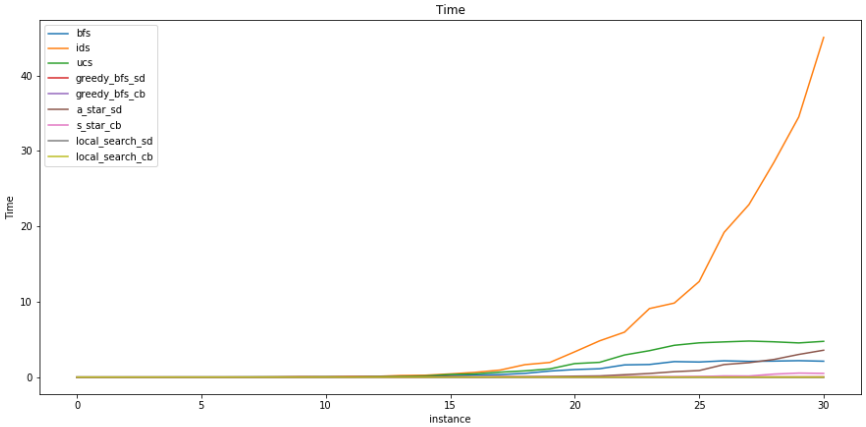

Figure 4: Tempod e execução

	bfs steps	bfs Nodes	bfs time	ids steps	ids Nodes	ids time	ucs steps	ucs Nodes	ucs time
0	0.0	0.0	0.000003	0.0	0.0	0.000006	0.0	0.0	0.000004
1	1.0	2.0	0.000214	1.0	2.0	0.000060	1.0	2.0	0.000134
2	2.0	9.0	0.000834	2.0	13.0	0.000304	2.0	8.0	0.000481
3	3.0	17.0	0.001490	3.0	19.0	0.000491	3.0	16.0	0.000827
4	4.0	33.0	0.002902	4.0	52.0	0.001149	4.0	27.0	0.001466
5	5.0	41.0	0.003967	5.0	111.0	0.002340	5.0	40.0	0.001960
6	6.0	71.0	0.005632	6.0	158.0	0.003558	6.0	52.0	0.002444
7	7.0	183.0	0.013172	7.0	236.0	0.005057	7.0	144.0	0.006125
8	9.0	359.0	0.012410	9.0	901.0	0.015517	9.0	416.0	0.015542
9	10.0	779.0	0.016349	10.0	1829.0	0.027447	10.0	615.0	0.021405
10	11.0	1253.0	0.023066	11.0	2209.0	0.032567	11.0	1215.0	0.034818
11	12.0	1445.0	0.024045	12.0	4237.0	0.063436	12.0	1174.0	0.030041
12	13.0	3336.0	0.051543	13.0	5937.0	0.094508	13.0	3102.0	0.075161
13	14.0	4286.0	0.062932	14.0	16730.0	0.215796	14.0	3808.0	0.096865
14	15.0	6048.0	0.090541	15.0	23759.0	0.252315	15.0	5989.0	0.157809
15	16.0	12725.0	0.204584	16.0	37831.0	0.428011	16.0	12024.0	0.355998
16	17.0	15253.0	0.274159	17.0	60530.0	0.635648	17.0	16235.0	0.436840
17	18.0	22058.0	0.346674	18.0	81753.0	0.935701	18.0	19704.0	0.659995
18	19.0	34387.0	0.500217	19.0	148790.0	1.656600	19.0	29953.0	0.833443
19	20.0	54476.0	0.809174	20.0	149265.0	1.947650	20.0	39163.0	1.093850
20	21.0	70371.0	1.005700	21.0	328760.0	3.359850	21.0	59675.0	1.797370
21	22.0	77693.0	1.112150	22.0	477005.0	4.816410	22.0	72189.0	1.965140
22	23.0	123807.0	1.623750	23.0	535631.0	5.967450	23.0	101138.0	2.942280
23	24.0	131526.0	1.677440	24.0	891462.0	9.094850	24.0	120664.0	3.509660
24	25.0	161048.0	2.060410	25.0	1038620.0	9.826000	25.0	157833.0	4.224400
25	26.0	158452.0	2.007340	26.0	1306176.0	12.686600	26.0	167149.0	4.549930
26	27.0	178088.0	2.167480	27.0	2102665.0	19.202600	27.0	176375.0	4.674800
27	28.0	178208.0	2.094210	28.0	2359511.0	22.891700	28.0	177624.0	4.786530
28	29.0	180601.0	2.130180	29.0	3027113.0	28.489100	29.0	180572.0	4.688250
29	30.0	181355.0	2.183550	30.0	3402601.0	34.519900	30.0	181310.0	4.538920
30	31.0	181440.0	2.113800	31.0	4736734.0	45.073900	31.0	181439.0	4.749540

	greedy_sd steps	greedy_sd Nodes	greedy_sd time	a*_sd steps	a*_sd Nodes	a*_sd time
0	0.0	0.0	0.000008	0.0	0.0	0.000008
1	1.0	1.0	0.000055	1.0	1.0	0.000062
2	2.0	2.0	0.000122	2.0	2.0	0.000141
3	3.0	3.0	0.000165	3.0	3.0	0.000187
4	4.0	4.0	0.000185	4.0	4.0	0.000211
5	5.0	5.0	0.000231	5.0	5.0	0.000261
6	6.0	7.0	0.000236	6.0	8.0	0.000321
7	7.0	8.0	0.000302	7.0	17.0	0.000630
8	67.0	192.0	0.006092	9.0	27.0	0.001013
9	44.0	983.0	0.030053	10.0	31.0	0.001174
10	51.0	594.0	0.016729	11.0	37.0	0.001418
11	116.0	1738.0	0.047917	12.0	62.0	0.002390
12	61.0	1046.0	0.030659	13.0	137.0	0.004976
13	60.0	322.0	0.009357	14.0	273.0	0.009228
14	85.0	603.0	0.018085	15.0	361.0	0.011475
15	80.0	1795.0	0.052031	16.0	572.0	0.017153
16	75.0	444.0	0.012970	17.0	804.0	0.021869
17	164.0	1567.0	0.047198	18.0	1093.0	0.027452
18	89.0	1132.0	0.033552	19.0	1491.0	0.038054
19	110.0	1381.0	0.038941	20.0	2510.0	0.064806
20	59.0	478.0	0.012888	21.0	4750.0	0.123509
21	80.0	364.0	0.009748	22.0	6043.0	0.160287
22	121.0	1217.0	0.031018	23.0	10759.0	0.321521
23	120.0	1216.0	0.030771	24.0	16326.0	0.483027
24	97.0	456.0	0.011868	25.0	25048.0	0.724402
25	112.0	1283.0	0.034968	26.0	30167.0	0.872857
26	53.0	272.0	0.006997	27.0	56657.0	1.669490
27	134.0	999.0	0.025468	28.0	65239.0	1.919140
28	75.0	931.0	0.024313	29.0	79806.0	2.344160
29	154.0	890.0	0.023755	30.0	104132.0	3.011480
30	171.0	1101.0	0.029302	31.0	125408.0	3.572050

	greedy_cb steps	greedy_cb Nodes	greedy_cb time	a*_cb steps	a*_cb Nodes	a*_cb time
0	0.0	0.0	0.000005	0.0	0.0	0.000007
1	1.0	1.0	0.000038	1.0	1.0	0.000065
2	2.0	2.0	0.000090	2.0	2.0	0.000148
3	3.0	3.0	0.000121	3.0	3.0	0.000200
4	4.0	6.0	0.000193	4.0	6.0	0.000290
5	5.0	9.0	0.000258	5.0	8.0	0.000377
6	6.0	9.0	0.000216	6.0	10.0	0.000400
7	7.0	9.0	0.000247	7.0	10.0	0.000398
8	9.0	13.0	0.000368	9.0	21.0	0.000865
9	62.0	235.0	0.006238	10.0	20.0	0.000836
10	53.0	172.0	0.004436	11.0	23.0	0.000934
11	42.0	123.0	0.003160	12.0	22.0	0.000879
12	83.0	267.0	0.007035	13.0	36.0	0.001509
13	38.0	72.0	0.001862	14.0	76.0	0.003031
14	37.0	137.0	0.003611	15.0	96.0	0.003851
15	34.0	88.0	0.002292	16.0	120.0	0.004791
16	83.0	221.0	0.005799	17.0	177.0	0.006606
17	34.0	101.0	0.002630	18.0	201.0	0.006998
18	37.0	87.0	0.002287	19.0	158.0	0.005182
19	48.0	97.0	0.002525	20.0	206.0	0.006822
20	49.0	96.0	0.002524	21.0	390.0	0.012481
21	50.0	97.0	0.002536	22.0	786.0	0.022861
22	53.0	270.0	0.007136	23.0	1010.0	0.027761
23	76.0	285.0	0.007675	24.0	1130.0	0.030912
24	53.0	176.0	0.004652	25.0	2084.0	0.059021
25	100.0	337.0	0.009110	26.0	2519.0	0.070750
26	199.0	892.0	0.026615	27.0	5457.0	0.161848
27	80.0	217.0	0.006463	28.0	4914.0	0.149894
28	49.0	103.0	0.003037	29.0	12358.0	0.404309
29	54.0	155.0	0.004508	30.0	16738.0	0.537190
30	53.0	114.0	0.003189	31.0	15044.0	0.497143

	ls_sd steps	ls_sd Nodes	ls_sd time	ls_cb steps	ls_cb Nodes	ls_cb time
0	0.0	0.0	0.000003	0.0	0.0	0.000003
1	1.0	1.0	0.000026	1.0	1.0	0.000022
2	2.0	2.0	0.000036	2.0	2.0	0.000035
3	3.0	3.0	0.000047	3.0	3.0	0.000059
4	4.0	4.0	0.000076	0.0	3.0	0.000045
5	5.0	5.0	0.000089	0.0	4.0	0.000052
6	0.0	1.0	0.000006	0.0	3.0	0.000032
7	0.0	2.0	0.000032	0.0	6.0	0.000072
8	0.0	4.0	0.000049	0.0	4.0	0.000050
9	0.0	5.0	0.000066	0.0	3.0	0.000045
10	0.0	2.0	0.000029	0.0	6.0	0.000072
11	0.0	3.0	0.000036	0.0	3.0	0.000032
12	0.0	8.0	0.000124	0.0	4.0	0.000056
13	0.0	5.0	0.000076	0.0	3.0	0.000042
14	0.0	4.0	0.000068	0.0	4.0	0.000061
15	0.0	7.0	0.000102	0.0	3.0	0.000045
16	0.0	6.0	0.000089	0.0	4.0	0.000069
17	0.0	1.0	0.000006	0.0	3.0	0.000032
18	0.0	6.0	0.000088	0.0	4.0	0.000069
19	0.0	7.0	0.000107	0.0	4.0	0.000044
20	0.0	6.0	0.000090	0.0	5.0	0.000071
21	0.0	7.0	0.000115	0.0	6.0	0.000086
22	0.0	2.0	0.000030	0.0	3.0	0.000034
23	0.0	1.0	0.000007	0.0	3.0	0.000032
24	0.0	6.0	0.000091	0.0	3.0	0.000041
25	0.0	7.0	0.000106	0.0	5.0	0.000071
26	0.0	4.0	0.000057	0.0	5.0	0.000071
27	0.0	1.0	0.000005	0.0	1.0	0.000006
28	0.0	6.0	0.000089	0.0	6.0	0.000083
29	0.0	7.0	0.000106	0.0	5.0	0.000089
30	0.0	2.0	0.000029	0.0	2.0	0.000029

2 Conclusão

Observando os resultados apresentados na última secção, podemos confirmar que os algoritmos A*, busca de custo uniforme, aprofundamento iterativo e busca em profundidade conseguem chegar na solução ótima, enquanto os demais não tem essa garantia, e no caso da busca local na maioria das instâncias nem foi possível achar o estado terminal. Em relação ao tempo e ao número de nós visitados no entanto os melhores resultados são os da busca local e busca gulosa. O A* se mostrou o melhor algorítimo em geral, chegando na solução ótima em um tempo muito próximo aos dos algoritmos que não tem essa garantia. Em geral a heurística de distância de manhattan teve a melhor performance.