Comparação de algoritmos de busca para resolver o 8-Puzzle

Victor Hugo Piontkievitz da Cruz[1]

[1]Universidade Tuiuti do Paraná
Curitiba – PR

{victor.cruz}@utp.edu.br

Resumo. *O trabalho atual se concentra em identificar as diferenças entre os algoritmos de busca cega e heurística, sendo o de Breadth-first Search (BFS) e Depth-first Search, e o A* e guloso, respectivamente, em relação aos recursos utilizados, como a memória e tempo, para a solução do problema do 8-Puzzle, que tem como objetivo ordenar 9 blocos enumerados, entre 0 (ou vazio) até 8, em um tabuleiro. O problema do 8-Puzzle é altamente estudado na área de Inteligência Artificial, sendo um exemplo utilizado e referenciado em materiais.*

1. Introdução

O 8-Puzzle é um jogo de tabuleiro, qual traz um problema conhecido na área de inteligência artificial e de soluções de puzzle, sendo utilizado como um exemplo qual pode ser resolvido utilizando algoritmos de busca. Ele é composto de blocos deslizantes, que devem ser posicionados de uma forma específica para se alcançar a vitória ou chegar no resultado desejado. São 9 blocos posicionados numa grade 3x3, desses quais 8 estão numerados com um número, indo de 1 até 8, e o bloco restante é vazio ou representado pelo 0. Normalmente, o objetivo é que eles estejam posicionados em ordem crescente, com o bloco vazio sendo posicionado por último. No entanto, as regras podem ser alteradas, colocando ordens específicas para se alcançar a vitória, diferente da casual. O estado inicial pode possuir os seus blocos em qualquer ordem, e o objetivo do jogo é chegar ao estado final por meio de deslizar o bloco vazio, o trocando com algum dos blocos adjacentes, quando possíveis, nas direções de cima, baixo, direita ou esquerda. Diversos algoritmos computacionais então existem, quais podem ser usados para resolvê-lo, e normalmente são realizadas comparações entre eles para ver quais são suas diferenças, avaliando o modo como operam e sua eficiência [JÚNIOR].

2. Busca Cega

Os algoritmos de busca cega, ou de busca não informada são aqueles que não tem informação sobre o seu espaço de busca, e resolvem a consulta de forma sistemática, sem qualquer conhecimento sobre conhecimento sobre o domínio [RUSSEL 2010] .

A Breadth-first search busca (BFS), ou a busca em largura, começa expandindo o nó raiz, e então todos os sucessores do nó raiz, seguido pela expansão de todos os sucessores dos sucessores do nó raiz, formando assim um tipo de busca por nível, em que todos os nós de um nível são expandidos antes de seguir para o próximo.

A Depth-first search (DFS), ou a busca em profundidade, sempre vai expandir o nó mais profundo na borda em que está da árvore, e segue expandindo até chegar um nó folha, que é o nível mais profundo. Quando alcançados, retornamos e expandimos o segundo nó mais profundo que ainda tem filhos.

3. Busca Heurística

Os algoritmos de busca heurística tentam encontrar resultados satisfatórios, não sendo necessariamente ótimos, os alcançando de acordo com as informações que possuem. A heurística pode ser baseada em aproximações e regras de dedução, que facilitam a sua busca [RUSSEL 2010] .

A busca gulosa sempre tenta expandir o nó que parece estar mais perto do objetivo que desejamos alcançar. A avaliação usada é apenas a função heurística $f(n) = h(n)$. No entanto, isso não quer dizer que ela realmente vai alcançar a melhor solução, pois pode ser levada a um beco sem saída.

A busca A*, é a forma mais popular em busca de caminhos. A avaliação usada por ela é a da combinação do custo para alcançar um nó e a do custo para ir do nó até o objetivo, tendo a função heurística de $f(n) = g(n) + h(n)$. Isso nos dá que $f(n)$ = custo estimado da solução de menor custo através de n.

4. Descrição da implementação e configuração dos testes

Os algoritmos de busca cega e busca heurística, sendo o BFS e DFS, e o de busca gulosa e o de A*, respectivamente, foram implementados para resolverem o problema do 8-Puzzle, e a linguagem utilizada para programação foi Python. Bibliotecas como o time, tracemalloc, csv e heapq foram utilizadas para facilitarem e aperfeiçoarem os algoritmos, bem como para medir o tempo e memória. Todos os algoritmos utilizam o mesmo padrão, tendo diferença apenas no que se refere ao modo em que fazem a busca, nos que possuem, na heurística utilizada.

Uma classe é utilizada para o Puzzle, com atributos para a grade, suas posições de acordo com a orientação, a profundidade da busca e o custo. O método *__lt__* é usado para a função de avaliação. Existe uma função que contém o objetivo final a ser alcançado, que pode ser alterado facilmente, mas para o teste foi utilizado o padrão de

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

qual é representado no código como $[[1, 2, 3], [4, 5, 6], [7, 8, 0]]$, sendo um tipo de lista que contém 3 listas, cada uma representando uma linha da matriz.

Outra função existente serve para validar se as posições horizontais e verticais estão dentro da grade do jogo, não sendo colocadas para fora do espaço 3x3. Para calcular a distância de cada bloco até a posição onde deveriam estar posicionados nas buscas heurísticas, é utilizada a distância de Manhattan, empregando as técnicas de *divmod* e *abs* do Python. Um método é usado para imprimir as posições de cada bloco em formato de matriz quando chamado e passado a grade como argumento.

O método do algoritmo então, de forma geral, recebe a grade e as posições como argumentos. Uma lista de visitados é criada utilizando o $set()$, devido a sua velocidade para encontrar os elementos inseridos ser maior que a de uma lista, pois utiliza uma *heap*. Para manter uma lista para os nós expandidos, o BFS utiliza a estrutura de fila, o DFS utiliza uma pilha, e o algoritmo guloso e o A* utilizam uma *heap*, através da biblioteca heapq, além de calcular o custo pela distância de Manhattan. Então, enquanto existem

elementos a serem expandidos, o nó atual se torna o próximo, é checado se o estado do tabuleiro é o estado final ou objetivo para poder finalizar a busca, e caso não seja, prosseguimos e tentamos trocar a posição do bloco. Caso seja válida, é realizada a troca no tabuleiro, e o estado é adicionado à lista de visitados, caso não esteja, e na estrutura de lista. Para as buscas heurísticas, é calculado a distância de Manhattan no processo. Para o "*main*"do código, é feito o *Parser* do arquivo em csv que possuí as instâncias. É pulada a primeira linha e então as restantes são lidas, removendo cada vírgula entre os números representando os blocos e os adicionando como elementos de uma lista. Uma lista é feita para cada linha, e então são adicionadas a uma segunda lista, que contém como cada elemento uma instância do 8-Puzzle, ficando da forma:

$$\big[[A1, A2...A9][B1, B2...B9]...[I1, I2...I9]\big]$$

Utilizando mais técnicas de listas, cada uma das instâncias é adaptada para o modo como o código aceita o 8-Puzzle, utilizando-se de 3 listas. Esse modo é:

$$\big[[A1, A2, A3], [A4, A5, A6], [A7, A8, A9]\big]$$

Após isso, encontramos a estrutura *for* que chama o algoritmo para cada instância encontrada. Inicializa as variáveis para cada chamada, ajusta a posição em que o bloco vazio está posicionado de acordo com o tabuleiro, imprime-o para visualização, e marcando a memória e o tempo iniciais, realiza a chamada do algoritmo de busca respectivo. Esse então retorna o número de passos e a profundidade em que o objetivo foi alcançado, caso tenha sido, e caso não, retorna os valores de -1 para ambos. Marcando a memória e tempo finais, junto com os retornos são adicionados em suas listas respectivas, cada um representando a instância para quais foram obtidos. Após todas as 10 instâncias terem sido realizadas, são mostradas as informações referentes a cada uma, sendo os passos levados para alcançar o objetivo, a profundidade, a memória utilizada em kBs e o tempo levado em segundos. As instâncias utilizadas podem ser vistas na tabela 1.

Tabela 1. Instâncias do 8-Puzzle.

Número da instância	Instância
1	[1,2,0,4,5,3,7,8,6]
2	[1,3,0,4,2,6,7,5,8]
3	[1,3,5,4,0,2,7,8,6]
4	[1,3,6,4,5,2,0,7,8]
5	[0,2,3,1,8,5,4,7,6]
6	[1,2,3,4,0,8,7,6,5]
7	[0,2,3,1,4,8,7,6,5]
8	[1,0,3,4,2,5,7,8,6]
9	[7,3,6,2,1,0,5,4,8]
10	[7,1,3,0,5,6,4,2,8]

5. Tabela comparativa com os resultados

A tabelas mostram a comparação entre os resultados obtidos para cada instância, para cada um dos algoritmos de busca utilizados. Os elementos representam o número de passos levados para alcançar o objetivo, a profundidade em que o objetivo foi alcançado, memória utilizada e tempo de execução.

A tabela 2 mostra os resultados do algoritmo BFS, a tabela 3 mostra os resultados do algoritmo DFS, a tabela 4 mostra os resultados do algoritmo guloso e a tabela 5 mostra os resultados do algoritmo A*.

Tabela 2. Resultados do BFS.

Instâncias	BFS			
Número	Passos	Profundidade	Memória (kBs)	Tempo (Segundos)
1	14	2	7.08	0.00039
2	40	4	9.344	0.00144
3	153	6	32.4	0.00451
4	321	8	46.868	0.010919
5	149	6	18.456	0.00526
6	163	6	20.696	0.00597
7	361	8	32.524	0.01061
8	30	3	1.688	0.00091
9	4540	13	116.148	0.14099
10	4157	13	117.044	0.10702

Tabela 3. Resultados do DFS.

Instâncias	DFS			
Número	Passos	Profundidade	Memória (kBs)	Tempo (Segundos)
1	5	2	4.072	0.00015
2	49	26	15.544	0.00133
3	111642	59034	132.764	4.43783
4	77807	42932	132.06	2.81859
5	167705	52436	132.572	6.52975
6	90055	49280	132.188	3.37016
7	25285	13922	128.988	0.78787
8	170323	49285	132.764	6.97913
9	1221	667	82.516	0.03187
10	100329	54253	131.74	3.67449

Precisa ser considerado que devido velocidade dos algoritmos e do hardware no qual os testes foram realizados, o tempo obtido se demonstrou muito breve, possuindo diversos zeros. A memória utilizada também se demonstrou pouca, por isso a escolha para fazer a representação usando kBs. Nos casos em que mais passos foram tomados e profundidades exploradas, no entanto, se demonstraram em maior capacidade.

6. Discussão sobre qual é a estratégia mais eficiente para o problema estudado

Observando os resultados obtidos, podemos perceber a diferença entre cada algoritmo. Em geral, as buscas cegas superaram os números nos campos disponíveis, demonstrando que para o problema do 8-Puzzle as buscas heurísticas podem ser mais recomendadas, mas não significando que vão ser melhores em todas as situações, pois existem particularidades no modo em que os algoritmos operam.

Tabela 4. Resultados da Gulosa.

Instâncias	Gulosa			
Número	Passos	Profundidade	Memória (kBs)	Tempo (Segundos)
1	5	2	4.072	0.00021
2	10	4	1.976	0.00040
3	15	6	1.976	0.00034
4	152	38	38.904	0.00522
5	13	6	1.176	0.00034
6	15	6	1.304	0.00031
7	18	8	1.624	0.00046
8	9	3	0.728	0.00017
9	28	13	2.584	0.00063
10	210	37	24.784	0.00571

Tabela 5. Resultados do A*.

Instâncias	A*			
Número	Passos	Profundidade	Memória (kBs)	Tempo (Segundos)
1	5	2	3.92	0.00019
2	10	4	1.976	0.00026
3	16	6	2.104	0.00041
4	26	8	3.608	0.00088
5	13	6	1.176	0.00044
6	24	6	2.488	0.00069
7	27	8	2.808	0.00064
8	9	3	0.728	0.00022
9	37	13	4.056	0.00097
10	122	13	17.944	0.00332

O algoritmo BFS apresenta um número interessante de passos. Em 8 das instâncias, foram dados entre 14 à 361 passos, mas nas outras duas foram dadas mais de 4000. Isso se deve ao modo como o BFS opera, pois cada profundidade a mais aumenta significativamente o número de passos, como pode ser observado. Os valores por perto de 300 vão até a profundidade 8, enquanto os valores acima de 4000 estão na 13. A memória usada e o tempo também aumentam do mesmo modo. Nos piores casos, chegou a usar cerca de 116 kBs, e levou 0.12 segundos para completar a busca.

O algoritmo DFS por sua vez se demonstrou o com pior eficiência entre todos os algoritmos. As profundidades exploradas chegam a quase 60000 no pior caso das instâncias, enquanto o seu maior número de passos foi de 170.323. O seu uso de memória, no entanto, não chega a ser diferente do que obtido nos casos piores BFS. A maioria de seus valores está por cerca de 13 kBs. Entre todos, foi o único algoritmo que levou mais de um segundo para encontrar alguma solução, chegando a levar até 6 no caso que demonstrou mais passos.

Observando a busca gulosa, podemos perceber que, em comparação aos algoritmos de busca cega, acabou chegando ao objetivo em um menor número de passos. A sua profundidade chegou a ser um pouco diferente da alcançada pelo A*, mas em 8 dos casos

apresentou a mesma. Sua memória então, igualmente, teve baixos números, ocupando cerca de 1 kBs e menos em 6 das situações, com o maior sendo 38.9 kBs. Assim, o tempo levado para chegar aos resultados se apresentou menor que todos da BFS, e apenas sendo maior que um valor da DFS, mas por apenas 0.06 milissegundos (ms).

O algoritmo de Busca A*, no entanto, é o que se apresentou mais consistente entre todos os estudados. Com o menor número de passos levados em geral, possui a sua profundidade semelhante à da busca gulosa, e de uma forma interessante apresenta a mesma que todas as instâncias do algoritmo BFS. Em relação a memória ocupada, demonstrou resultados maiores ou iguais que a busca gulosa em 7 situações, porém em relação aos piores casos, demonstrou ser um melhor algoritmo. Na pior situação da busca gulosa, foram ocupados 38.9 kBs, enquanto no A* foram apenas 3.6 kBs. O tempo levado apresenta comportamento semelhante a memória em comparação com a busca gulosa, superando o número em alguns casos.

7. Conclusão

Observando os resultados obtidos pela comparação dos algoritmos, podemos chegar à conclusão de que os algoritmos de busca heurística são mais adequados do que os algoritmos de busca cega para resolver o problema do 8-Puzzle. Isso se deve a heurística utilizada, que mensura um valor aproximado até o objetivo final desejado, possibilitando o algoritmo a fazer melhores escolhas, enquanto os de busca cega apenas podem expandir nós, sem saber se a situação está ficando melhor ou pior. O BFS se demonstrou ser melhor que o DFS, pois a chance de chegar ao objetivo caso poucos blocos estejam fora de suas posições é maior, sendo que o DFS pode acabar tomando um grande desvio em sua busca. Comparando o algoritmo A* e o guloso, podemos perceber que são muito semelhantes. Os seus resultados não diferem muito, e o guloso apresentou em mais casos um menor consumo de memória e de tempo. No entanto, o A* aparenta ser mais consistente em seus valores, apresentando um consumo de até 4 kBs de memória em 9 das situações. O guloso por sua vez, na pior situação, apresentou um resultado muito mais elevado, devido ao modo como age, sempre pegando o que aparenta ser a melhor opção. Caso instâncias diferentes fossem utilizadas, pode ser que esse comportamento se apresentasse frequente.

Dessa maneira, o algoritmo A* aparenta ser o mais recomendado entre os algoritmos de busca estudados. Embora o guloso apresente resultados melhores em mais situações, em uma situação mais difícil chega a apresentar bem piores, enquanto o A* certamente parece ser mais consistente entre seus valores. Os resultados obtidos então apontam o A* como uma melhor solução, mas por pouco. Para melhores conclusões, é recomendado que instâncias diferentes e mais complicadas do problema do 8-Puzzle sejam utilizadas, para que se possa identificar melhor as discrepâncias entre os algoritmos de busca heurística A* e o guloso no que se refere a suas eficiências em relação ao consumo de recursos.

Referências

JÚNIOR, N. F. e GUIMARÃES, F. G. Problema 8-puzzle: Análise de solução usando backtracking e algoritmos genéticos.

RUSSEL, Stuart J.; NORVIG, P. (2010). *Inteligência Artificial: Uma Abordagem Moderna.*, volume 3. São Paulo: Prentice Hall.