Documentação Trabalho Prático I de Inteligência Artificial: Algoritmos de busca em árvore para resolver o n-Puzzle

Ronald Davi Rodrigues Pereira
ronald.pereira@dcc.ufmg.br

Universidade Federal de Minas Gerais

26 de Abril de 2019

1 Introdução

O jogo do 8-Puzzle[1], também chamado de "Racha-Cuca", é um famoso quebra-cabeças de 8 peças. No Brasil, este jogo virou uma verdadeira febre por por volta do ano de 1955. Era difícil uma criança e mesmo um adulto, não possuir a sua caixinha com as peças deslizantes. Ele se trata de um quebra-cabeças de oito peças, composto por uma placa oca de metal com quinze quadrados que trocam de lugar, todos gravados com números, letras ou figuras. O objetivo é arranjar as peças em ordem, da esquerda para a direita, de cima a baixo.

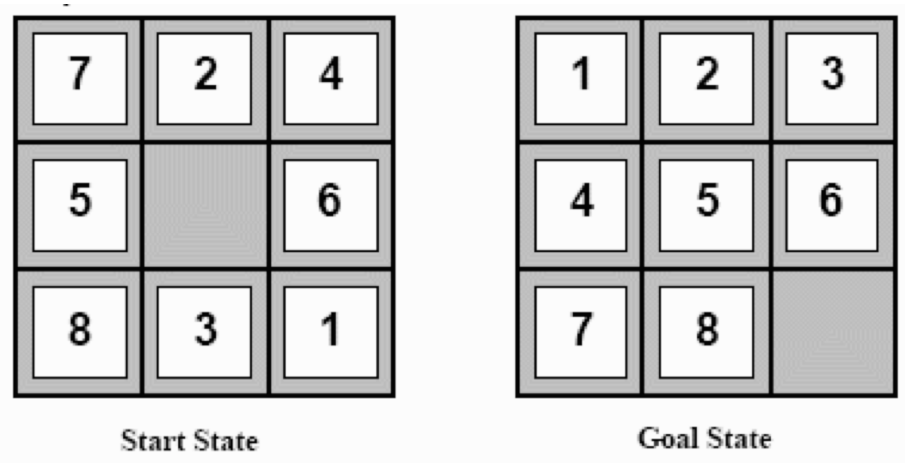

Figura 1 – Exemplo de estado inicial e final para uma instância do 8-puzzle

O objetivo desse trabalho consistiu em implementar e comparar diferentes métodos de busca apresentados durante o curso de Inteligência Artificial, aplicando-os a um problema proposto. Sabe-se também que o problema de encontrar uma solução com o menor número de passos no caso geral, denominado n-Puzzle, é NP-Difícil[2]. No entanto, foi realizado uma tentativa de se resolver esses casos com o algoritmo tido como ótimo nesse trabalho: *A* Search* (Busca A-estrela).

2 Modelagem e Implementação

A implementação foi realizada na linguagem Python 3.7.3, contendo um arquivo *solver.py* que resolve os n-puzzles dados para ele como entrada e um outro arquivo *generator.py* que gera um n-puzzle resolvível com os argumentos passados para ele.

2.1 Estados

Cada estado é composto por uma classe do tipo *StateNode*, que possui em seu interior:

n Dimensão do n-puzzle

puzzle Representação do puzzle do estado atual

father Apontador para o estado anterior, de modo a conseguir caminhar inversamente na árvore

up Apontador para o estado seguinte, movendo o espaço uma posição para cima, caso possível

down Apontador para o estado seguinte, movendo o espaço uma posição para baixo, caso possível

left Apontador para o estado seguinte, movendo o espaço uma posição para esquerda, caso possível

right Apontador para o estado seguinte, movendo o espaço uma posição para direita, caso possível

Abaixo podemos ver um esquema simplificado da árvore de estados com apenas um nível de expansão:

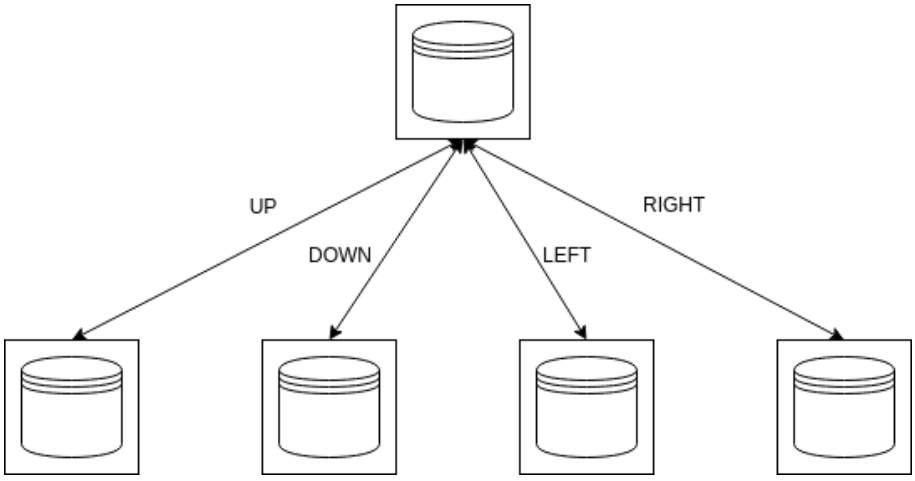

Figura 2 – Exemplo de um nível de expansão do nodo StateNode

2.2 Função Sucessora

A função sucessora de cada nodo se encontra dentro da própria classe do objeto (*StateNode.expand*). Essa função implementa a movimentação do espaço vazio do tabuleiro em uma posição, caso seja possível e a expande sob demanda dos algoritmos de busca. Essa expansão foi realizada para ser sob demanda pelo simples fato que o número de estados (contendo repetições não-triviais) de qualquer entrada é potencialmente infinito. Sendo assim, muitas das vezes o algoritmo não conseguiria nem mesmo começar a executar, pois a construção da árvore nunca terminaria.

3 Algoritmos

Nessa seção, serão apresentados os diferentes algoritmos implementados no projeto, bem como as principais diferenças sobre eles.

3.1 Breadth-First Search

Algoritmo clássico de busca em árvore por largura, consiste em expandir todos os nodos presentes em um determinado nível antes de expandir os nodos do próximo nível. Utiliza de uma fila (FIFO - *First In, First Out*) para representar a sua fronteira, de modo a inserir os novos nodos expandidos no final. Possui complexidade de tempo e espaço em $O(b^d)$, sendo b o *branching factor* (quantos nós são expandidos de cada nó pai em média, que nesse problema são aproximadamente 3) e d o nível da árvore em que se encontra a solução. No entanto, mesmo com essa complexidade elevada, ele é um algoritmo completo (encontra sempre uma solução, caso ela exista) e é ótimo (a solução encontrada é sempre a de menor custo), pois o custo de cada passo é sempre não-decrescente. Porém, esse algoritmo, quando comparados com os demais, é um dos que mais realiza expansões de nodos desnecessários (que não levam à melhor solução), o que reduz bastante a sua eficiência comparativamente.

3.2 Iterative Deepening Search

Adaptação do algoritmo clássico de busca em árvore por profundidade (*Depth-First Search*), consiste em expandir todos os nodos diretamente passando ao nível seguinte até uma profundidade máxima i, que é iterativa a cada ciclo da busca, combinando os benefícios do *Breadth-First Search* e *Depth-First Search*. Utiliza de uma pilha para representar a sua fronteira (LIFO), de modo a inserir os novos nodos expandidos no topo. Possui complexidade de tempo em $O(b^d)$ e complexidade de espaço em $O(bd)$, sendo b o *branching factor* (quantos nós são expandidos de cada nó pai em média, que nesse problema são aproximadamente 3) e d o nível da árvore em que se encontra a solução. Ele também é um algoritmo completo (encontra sempre uma solução, caso ela exista) e é ótimo (a solução encontrada é sempre a de menor custo), pois o custo de cada passo é sempre não-decrescente. Porém, esse algoritmo varia bastante em sua eficiência, pois caso a solução esteja mais à esquerda da árvore de soluções, é encontrada rapidamente, enquanto caso esteja mais à direita, várias expansões desnecessárias são realizadas, mesmo que a sua utilização de espaço seja a melhor entre os algoritmo utilizados nesse projeto.

3.3 Uniform-Cost Search

Algoritmo semelhante ao *Breadth-First Search*, consiste em expandir o nodo de menor custo até o momento. Utiliza de uma fila de prioridades (*heap*), de modo a sempre selecionar o nodo que possui o menor custo acumulado em sua fronteira. Possui complexidade de tempo e espaço em $O(b^{1+C^*/\epsilon})$, sendo b o *branching factor* e C^* o custo da solução ótima. No n-puzzle, como o custo mínimo e máximo de cada passo é 1, $\epsilon = 1$, isso resulta em uma complexidade resultante de $O(b^d)$, sendo d o nível da árvore em que se encontra a solução. Ele também é um algoritmo completo (encontra sempre uma solução, caso ela exista) e é ótimo (a solução encontrada é sempre a de menor custo), pois o custo de cada passo é sempre não-decrescente e ele sempre segue o caminho do menor custo possível. No entanto, mesmo sendo um dos melhores algoritmos de busca sem informação, ele ainda realiza muitas expansões de nodo desnecessárias, diminuindo significantemente sua eficiência.

3.4 A* Search

Algoritmo de busca com informação que utiliza da heurística *Manhattan Distance*. A escolha do nodo a ser investigado é feita considerando-se ambos o custo real de se chegar à solução atual ($g(n)$) e a estimativa do custo para se chegar à solução ótima ($h(n)$), resultando em uma estimativa do custo da melhor solução que passa pelo nó atual n expressa por $f(n) = g(n) + h(n)$. Nesse projeto, esse algoritmo é ótimo (a solução encontrada é sempre a de menor custo), pois a heurística é admissível e consistente, e é completo (encontra sempre uma solução, caso ela exista). Ele é também eficientemente ótimo, expandindo o menor número possível de nodos comparativamente com os demais algoritmos ótimos apresentados. No entanto, apesar disso tudo, a complexidade de tempo em relação ao número de nodos expandidos, dependendo da heurística utilizada, ainda pode ser uma função exponencial da solução e a complexidade de espaço ainda é elevada, pois todos os nodos da fronteira são mantidos em memória. Mesmo assim, se mostrou na maioria das vezes significantemente superior aos demais algoritmos implementados nesse projeto.

3.5 Greedy Best-First Search

Algoritmo de busca com informação que utiliza da heurística *Hamming Priority*. O termo *greedy* (guloso) significa que o método procura reduzir o custo imediato para alcançar o objetivo na expansão de cada nó, porém sem se preocupar com o custo total do caminho atual ($f(n) = h(n)$). Possui complexidade de tempo e espaço em $O(b^m)$, sendo b o *branching factor* (quantos nós são expandidos de cada nó pai em média, que nesse problema são aproximadamente 3) e m a profundidade máxima da árvore de busca. No entanto, esse algoritmo não é ótimo e nem completo, podendo entrar em loop, inclusive. É um bom algoritmo para servir de comparação *baseline* para outros algoritmos de busca com informação, mas pelo fato de depender fortemente da função heurística para seu sucesso, não é o mais recomendado para se utilizar.

3.6 Hill Climbing with Lateral Movements

Algoritmo de busca local que utiliza da heurística *Manhattan Distance*. Esse algoritmo guardam somente o estado corrente, investigam os vizinhos em busca de um estado melhor até que seja encontrada a solução, e o caminho encontrado até ela. Porém, esse algoritmo possui limitações bem significativas, como por exemplo: atingir um estado

que se mostre como um mínimo local, em que todos os vizinhos possuem custos maiores ou iguais a ele, mas não é uma solução do problema. No entanto, para que esse problema com custos maiores seja solucionado, é passo por parâmetro um argumento $k(k \in [1, \text{inf}))$, de modo que seja possibilitado movimentos laterais da função de custo (movimentos em que o custo se permanece inalterado). Mesmo assim, o algoritmo fica "preso"em um estado de mínimo local em um número significativo de vezes para soluções de custos relativamente elevados. Esse é um bom algoritmo para servir de comparação com os demais algoritmos, mas o fato de que ele possui essas limitações faz dele um dos algoritmos menos eficazes desse projeto.

4 Heurísticas

Nessa seção, serão apresentados as diferentes funções heurísticas implementados no projeto para sua utilização em algoritmos de busca com informação, bem como as principais diferenças sobre eles.

4.1 Hamming Priority

Essa função heurística representa o número de peças na posição errada do tabuleiro. Essa é uma função consistente (os custos são sempre crescentes) e admissível, porque cada peça deverá ser movimentada pelo menos uma vez, fazendo com que o custo real sempre seja maior ou igual ao estimado pela heurística ($C^*(n) \geq h(n)$). Essa heurística foi utilizada no algoritmo de busca com informação *Greedy Best-First Search*. Ela se mostrou uma heurística satisfatória, porém não gera a maior estimativa do menor custo possível para se chegar à solução ótima.

4.2 Manhattan Distance

Essa função heurística representa a soma das distâncias de cada peça às suas posições corretas no tabuleiro. Essa também é uma função consistente (porque os custos são sempre crescentes) e admissível, porque cada ação movimenta cada peça apenas um passo mais próximo do objetivo, fazendo com que o custo real sempre seja maior ou igual ao estimado pela heurística ($C^*(n) \geq h(n)$). Essa heurística foi utilizada no algoritmo de busca com informação *A* Search* e no algoritmo de busca local *Hill Climbing with Lateral Movements*. Ela se mostrou como a melhor heurística encontrada para o atual problema do n-puzzle, pois maximiza a estimativa do menor custo possível para se chegar à solução ótima.

5 Soluções Encontradas e Discussão dos Resultados

Não foi encontrada nenhum solução em um espaço de tempo de execução de 8 horas para o *solution_31* proposto. Porém, foram realizados diversos outros testes com diversas outras entradas (inclusive para n-puzzles com $n > 3$). Essas soluções podem ser encontradas no caminho *src/output*. No entanto, para os arquivos de custo acima do *solution_8*, os algoritmos já apresentaram eficiência muito pequena, principalmente *Greedy Best-First Search* e *Hill Climbing with Lateral Movements*.

Antes que entremos nas análises, algumas considerações devem ser feitas:

1. O algoritmo *Hill Climbing with Lateral Movements* foi ignorado nessa discussão, pelo fato de que para soluções com custos mais elevados, ele não encontrava uma solução ótima na maioria significativa das vezes, sendo portanto uma comparação desnecessária entre os demais algoritmos.

2. O algoritmo *Greedy Best-First Search* não terminou de ser executado em um tempo de execução de 4 horas para o *solution 15*, portanto não foi colocado na comparação.

3. A preferência de utilização de uma tabela ao invés de um método mais visual como gráficos, que se deve ao fato de que, como as magnitudes dos valores comparados são muito diferentes, os gráficos gerados ficaram muito pouco informativos.

5.1 Soluções do n_puzzle.pdf

Algorithm	**Solution**	**Node Expansions**	**Solution Cost**	**Execution Time**
Breadth-First Search	1	3	1	0.001s
	5	51	5	0.020s
	8	154	8	0.164s
	15	6908	15	298.038s
Iterative Deepening Search	1	4	1	0.001s
	5	331	5	0.018s
	8	3418	8	0.096s
	15	10389031	15	376.812s
Uniform-Cost Search	1	3	1	0.001s
	5	51	5	0.036s
	8	154	8	0.237s
	15	6908	15	424.604s
A* Search	1	1	1	0.000s
	5	5	5	0.001s
	8	8	8	0.002s
	15	56	15	0.037s
Greedy Best-First Search	1	1	1	0.001s
	5	5	5	0.002s
	8	8	8	0.002s

Tabela 1 – Tabela de comparação entre os algoritmos para soluções do n_puzzle.pdf

Conforme pode-se observar na Tabela 1, o algoritmo *A* Search* obteve uma eficiência superior aos demais algoritmos, o que cresce significantemente conforme o custo da solução vai aumentando. Os algoritmos de busca com informação conseguiram um desempenho superior aos demais algoritmos apresentados também. Porém, para a solução de um 8-puzzle com solução ótima de custo 15, o *Greedy Best-First Search* nem mesmo conseguiu terminar de executar com 4 horas de execução corrida. Desse modo, podemos concluir que o melhor algoritmo implementado foi o *A* Search*, sem nenhuma dúvida.

5.2 NP-Difícil n-puzzle com A* Search

Todos as estruturas de dados e algoritmos foram projetados para executar com um número $n \in [1, \text{inf})$ para n-puzzle. Porém, conforme visto anteriormente, o melhor

algoritmo implementado nesse projeto foi o *A* Search* por uma margem de distância bem grande com relação à eficiência. Portanto, foram realizados diversos testes explorando a versão NP-Difícil do problema.

Version	n	Node Expansions	Solution Cost	Execution Time
8-puzzle	3	218	20	0.262s
15-puzzle	4	60	20	0.061s
24-puzzle	5	31	20	0.023s
143-puzzle	12	28	20	0.066s

Tabela 2 – Tabela de comparação entre soluções do A* Search para o diferentes n do n-puzzle

Podemos concluir, de acordo com a Tabela 2, que o algoritmo *A* Search* possui uma certa independência do tamanho do tabuleiro em si, não importando muito qual o n escolhido para o n-puzzle.

Version	n	Node Expansions	Solution Cost	Execution Time
15-puzzle	4	60	20	0.061s
		898	28	4.545s
		30155	30	54.697s

Tabela 3 – Tabela de comparação entre soluções do A* Search para o diferentes custos da solução ótima do n-puzzle

Podemos concluir também, de acordo com a Tabela 3, que o algoritmo *A* Search* possui uma forte relação com o custo ótimo da solução, de modo a aumentar significantemente o número de expansões e o tempo de execução do algoritmo.

6 Referências Bibliográficas

[1] WIKIPEDIA. 15 puzzle. https://en.wikipedia.org/wiki/15_puzzle. Acessado em 26 de Abril de 2019.

[2] RATNER, D.; WARMUTH, M. K. Finding a shortest solution for the n× n extension of the 15-puzzle is intractable. In: . c1986. p. 168–172.