Universidade de São Paulo - ICMC
SCC0530 - Inteligência Artificial
Prof. Solange Oliveira Rezende

Busca de caminho em um Sliding Puzzle

Gabriel Simmel Nascimento	Nº USP: 9050232
Giovanna Oliveira Guimarães	Nº USP: 9293693
José Augusto Noronha de Menezes Neto	Nº USP: 9293049
Julia Diniz Ferreira	Nº USP: 9364865
Lucas Alexandre Soares	Nº USP: 9293265
Otávio Luis Aguiar	Nº USP: 9293518
Rafael Augusto Monteiro	Nº USP: 9293095

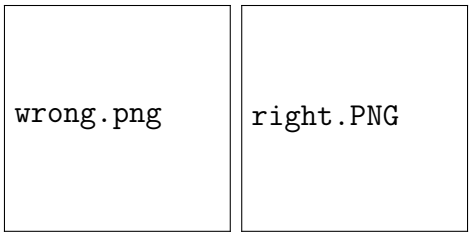
São Carlos - SP

8 de maio de 2018

Sumário

1 Introdução

O problema tratado neste projeto é a busca de um caminho para resolver um tabuleiro do jogo *Sliding Puzzle*. O jogo consiste em um tabuleiro N por N, onde N é um inteiro maior que 1, de peças deslizantes, que precisam ser arranjadas de forma a encontrar uma configuração específica. Para atingir o estado final, deve-se deslizar peças para o espaço vazio. A imagem 1 exemplifica um tabuleiro em seu estado inicial, enquanto a imagem 2 exemplifica o tabuleiro no estado final.



Uma solução para o problema é um caminho, isto é, um conjunto de passos que podem ser dados para atingir o estado alvo. Modelando o problema como uma busca em árvore, é possível utilizar algoritmos de busca para encontrar um caminho ótimo. Nas sessões seguintes, descreveremos mais profundamente a modelagem do problema e os algoritmos utilizados.

2 Métodos

2.1 Modelagem

Com o objetivo de encontrar um método genérico para solucionar os *puzzles*, implementamos um grafo no qual cada um de seus nós consiste nos estados possíveis de um tabuleiro qualquer. As Figuras 1 e 2, por exemplo, representam dois nós de um destes grafos, sendo a Figura 1 o nó inicial e a 2 o nó alvo (final).

Os nós adjacentes representam os estados que podem ser alcançados a partir do nó atual ao "movimentar"o espaço vazio do tabuleiro para qualquer posição válida. Ou seja, se a lacuna encontra-se em uma posição na qual há quatro possibilidades válidas de movimento, o vértice correspondente a configuração atual terá quatro nós vizinhos.

Embora trate-se de um problema aparentemente simples, o desenvolvimento de uma algorítimo capaz de apresentar uma solução para um caso genérico não é uma tarefa trivial, devido à complexidade do espaço de busca. Por exemplo, o caso de um jogo de dimensões 4x4:

- Cada uma das peças pode ocupar qualquer posição do tabuleiro;
- Para cada posição que uma peça ocupa, as demais também podem alternar-se entre os demais espaços disponíveis.

Dado isto, vemos que a complexidade do espaço de busca é fatorial e, por isso, mesmo para jogos cujas medidas são pequenas, o espaço obtido terá dimensões extremamente grandes. Tal como no caso apresentado, que possui um total de $2.092279e^{13}$ estados válidos.

Ainda, com o intuito de reduzir o espaço de busca, limitamos para casos onde o caminho mínimo possui no máximo 45 passos.

2.2 Algoritmos de Busca

Nesta sessão, serão descritos os algoritmos de busca implementados para a resolução do problema de busca apresentado acima.

2.2.1 A*

O A* é um algorítimo de busca informada, que percorre o espaço de busca a procura da solução que apresentar menor custo para o problema em questão. O algorítimo considera, dentre as possibilidades, primeiramente as que, ao que tudo indica, são a melhor opção no momento. A cada iteração, o A* determina quais de suas pré soluções deverão ser expandidas, o que é feito com base em uma estimativa do custo necessário para que o objetivo seja atingido. Em suma, o A* escolhe as soluções que minimizem a função:

$$f(n) = g(n) + h(n)$$

Onde n é o último nó adicionado à solução, $g(n)$ é o custo da solução desde o nó inicial até n e $h(n)$ uma heurística admissível que estima o custo da solução mais barata de n até o objetivo final.

Para o cenário de um *15 Sliding Puzzle Game*, a distância de Manhattan é uma heurística $h(n)$ admissível, isto é:

$$h(n) \leq h*(n)$$

Para todos os nós n, onde h^* é o custo real da melhor solução de n para o objetivo final.

Ao utilizarmos uma heurística admissível, a solução encontrada pelo A* será sempre ótima. Para $g(n)$, nosso algorítimo considera o custo de se realizar um movimento válido no tabuleiro, ou seja, partir do nó atual (estado imediato) para algum de seus vizinhos (estados válidos alcançáveis).

Com o objetivo de reduzir o número de operações realizadas pelo algorítimo, a cada passo realizado, é feita uma checagem para garantir que o A* nunca retorne ao estado do qual acabara de sair.

2.2.2 IDA*

O IDA*, *Iterative-deepening-A**, performa, a cada iteração, uma busca de primeira profundidade. Ou seja, quando o custo total de uma solução, $f(n)$, excede um dado limite de profundidade, o IDA* interrompe, momentânea ou permanentemente, o processamento de tal solução. O cálculo deste limite tem início com a estimativa do custo no estado inicial da busca e aumenta a cada iteração do algorítimo. A cada passo do *loop*, o limite utilizado para o próximo é o custo mínimo dentre todos os valores que excedem limiar atual.

Tal como o A* descrito anteriormente, o IDA* também garante que a solução encontrada é a solução ótima, desde que seja escolhida uma heurística admissível para o contexto do problema.

2.3 Programa

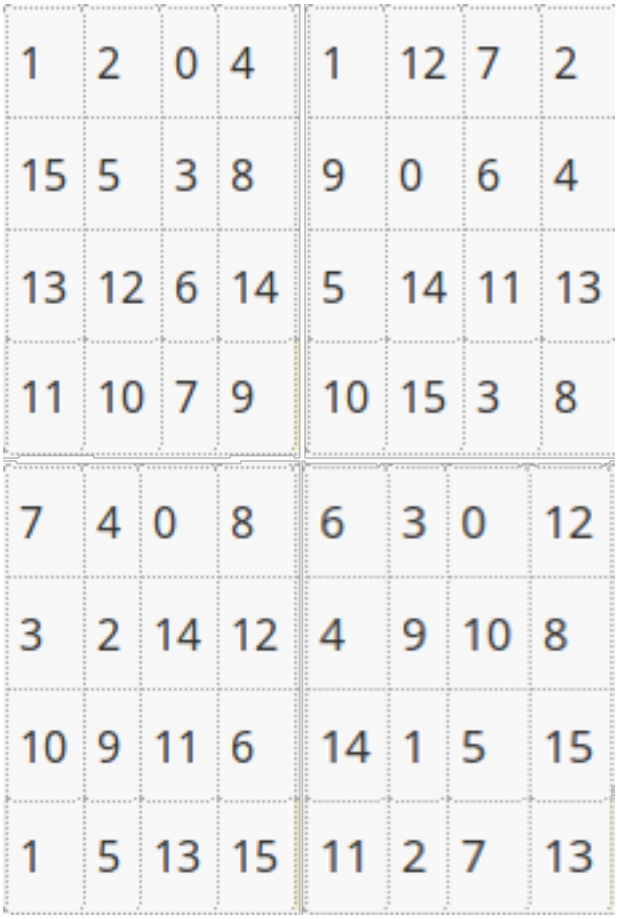
2.3.1 Descrição

Dois programas foram feitos em linguagem C++, um para cada algoritmo, cada um em um arquivo *.zip*. Ambos os programas recebem a mesma entrada: 16 valores inteiros separados por espaços e/ou quebra de linhas, e suas saídas são o tempo de execução, o número de iterações e a sequência de passos que resolve o tabuleiro. Para compilar e rodar os programas, um Makefile está incluso em cada arquivo *.zip*, basta executar `make` e `make run`.

3 Resultados

3.1 Execuções

A tabela 1 apresenta o tempo médio de dez execuções dos algoritmos para quatro tabuleiros com as seguintes configurações:



	Tempo de Execução	
	IDA*	A*
Média caso 1	0.001365	0.052758
Desvio caso 1	0.000234	0.003331
Média caso 2	0.597372	3.655886
Desvio caso 2	0.002716	0.045991
Média caso 3	0.001580	0.238017
Desvio caso 3	0.000033	0.008325
Média caso 4	2.736517	Memória da máquina excedida
Desvio caso 4	0.041999	Memória da máquina excedida

3.2 Discussão

Apesar de possuírem uma complexidade temporal equivalente, o IDA* possui uma menor complexidade espacial. Por manter uma fila com os nós ainda não explorados, o A* pode rapidamente consumir boa parte da memória disponível para execução do programa, pois, em seu pior caso, haverá um número exponencial de nós. Já o IDA* possui uma demanda linear de memória, visto que não armazena nenhum outro nó que não seja o atual. Como podemos ver na tabela, o A* não foi capaz de resolver um dos testes por dar erro de memória, mostrando

que, de fato, o A* consome muito mais memória. Este comsumo maior de memória implica em mais custos de tempo também devido à necessidade de manter e manejar os nós já visitados, tornando-o mais lento que o IDA* no caso geral.

A princípio, os algorítimos foram implementados em *python* em função da simplicidade da linguagem, contudo, o elevado tempo de execução dos casos testes tornou a abordagem inviável e, por fim, optamos por uma implementação em C++.

4 Conclusão

Por fim, pode-se concluir que, para nosso o problema, o IDA* mostra-se como uma alternativa superior ao A*, dado que seu consumo de memória é muito menor (linear x exponencial) e não há diferença significativa no tempo de execução das duas abordagens.

Além disso, notamos que *python* não é uma linguagem apropriada para este cenário específico, uma vez que, por ser de muito alto nível, ela não oferece recursos necessários para que os algorítimos desenvolvidos sejam executados em um intervalo de tempo aceitável.

5 Extra

No início, começamos a implementar os algoritmos em Python. Porém, a execução mostrou-se muito custosa, devido ao *overhead* de operações em listas e criação de objetos. Em seguida, reimplementamos os algoritmos em C++, que se mostrou muito mais rápido. Os scripts gerados foram incluídos na submissão, na pasta "Extra". Para sua execução, é necessário a biblioteca Numpy, além do Python3.