UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC865 - Projeto e Análise de Algoritmos

8-PUZZLE

17 de Outubro de 2018

Luis G A Diniz - dinizluis@dcc.ufmg.br
Matrícula: 2018718066
Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Conteúdo

Introdução

O objetivo do presente trabalho é implementar um algoritmo que retorne a solução ótima para o jogo 8-Puzzle, de forma que o problema seja modelado e resolvido com conceitos de Teoria dos Grafos estudados em sala de aula.

O Jogo

O jogo consiste em um puzzle que se inicia num tabuleiro hxh com apenas uma posição livre. Cada célula pode conter um número entre 0 e $h^2 - 1$, onde 0 representa uma posição vazia. A cada rodada, a partir do estado atual, um movimento é realizado (cima, baixo, esquerda ou direita) e um novo estado é gerado. Há diversas versões do jogo (N-Puzzle, onde $N = h^2 - 1$), a versão 8-Puzzle será tratada neste trabalho.

O objetivo do jogo é chegar num estado em que os números estejam ordenados da esquerda para direita e de cima para baixo: {0 1 2}, {3 4 5}, {6 7 8}. Foi convencionado que cada movimento tem custo um, portanto a solução ótima consiste em achar o menor número de movimentos possíveis para, a partir de um estado inicial, ganhar o jogo.

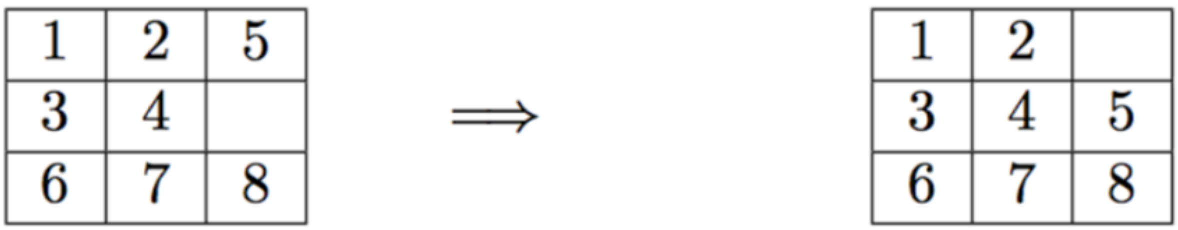

Figura 1: Transição no jogo 8-puzzle: o estado 2 (destino) é resultado do movimento para cima a partir de 1 (origem).

Modelagem e Solução

Modelagem

Cada instância do jogo foi abstraída como um grafo direcionado acíclico, ou simplesmente uma árvore de busca. A partir de um estado inicial os filhos são criados como os possíveis estados gerados a partir dos movimentos possíveis. Dessa forma um nó da árvore pode ter 1, 2 ou 3 filhos.

A linguagem de programação escolhida foi Python e apenas uma classe foi criada. A classe *board_class.py* contém a estrutura necessária para tratar o problema. Há apenas três atributos na classe, são eles, *conf* (lista com os algarismos na ordem em que aparecem no tabuleiro), *status* (indica se a configuração do tabuleiro é final ou não) e *pai* (aponta para o pai do nó, raiz aponta para *None*).

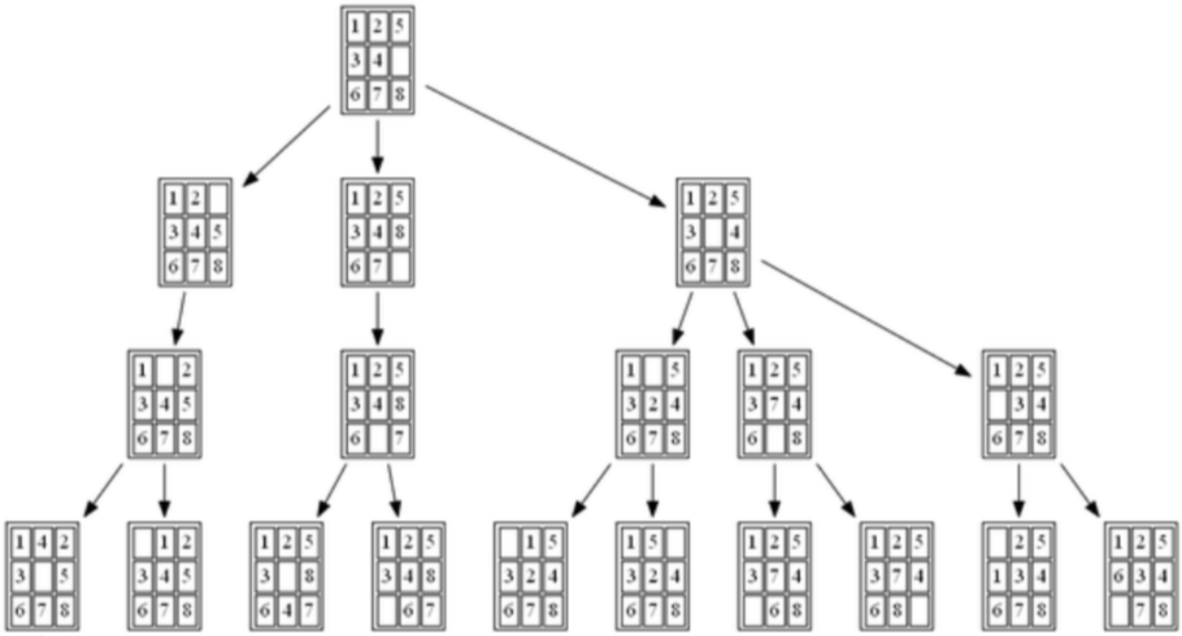

Figura 2: Exemplo de árvore de busca.

Solução

Com base na estrutura criada para representar o jogo, basta fazer a árvore de busca crescer (nível por nível) até chegar num estado final. Como o primeiro estado final é um nó folha é preciso apenas retornar a altura da árvore para que a solução ótima seja encontrada.

Para que a árvore crescesse nível por nível em tempo a heurística A* adaptada e aplicada. Alguns métodos auxiliares foram utilizados para implementar a heurística, abaixo segue pseudocódigo da solução aplicada.

O método *heuristicValue()* é capaz de calcular o valor heurístico de determinada configuração (armazenada no objeto board). Tal cálculo é realizado por meio da soma das distâncias de Manhattan de cada elemento no tabuleiro à sua devida posição (posição do estado final). Tal valor deve ser somado com o atual custo para chegar àquele estado (representado por g no algoritmo).

Algorithm 1 findMinCost(board)

```
se board é final então
    retorna 0
senão
    q = ∅
    r = (board.heuristicValue(), board) //r é uma tupla
    q = q ⋃ r
    nodes = 0 //Mantém número de nós expandidos
    enquanto q não é vazia faça
        u = min(q) //Escolhe o filho com menor valor heurístico para expansão
        Remove u de q
        u.haveChild()//Gera filhos de u (segue a ordem CBED)
        para i ← 1 até len(filhos de u) faça
            nodes+ = 1
            se o i-ésimo filho de u é final então
                retorna len(ancestrais de u)+1, nodes
            senão
                g = len(ancestrais de u)+1 //Representa o atual custo do nó
                h = i−ésimo_filho_de_u.heuristicValue() //Calcula o valor heurístico
                q = q⋃(h + g, i-ésimo filho de u)
            fim se
        fim para
    fim enquanto
fim se
```

Algorithm 2 heuristicValue(board)

```
h = 0
para i ← 1 até h^2 faça //h = 3
    h = h + board.manhattanDistance(board.conf, i)
fim para
retorna h
```

É importante ressaltar que a implementação da heurística foi adaptada em relação à versão encontrado na literatura [1], de forma à não usar duas listas para armazenar os vértices visitados e não visitados. Os nós já visitados são desprezados e não há impactado na solução pois sua função é garantir que não haja nós repetidos na árvore.

Para que cada configuração do tabuleiro seja única na árvore de busca é verificado se o novo estado já existe durante a geração de novos filhos. Ou seja, o método auxiliar *haveChild()* garante que cada estado é único na árvore de busca.

ANÁLISE DE COMPLEXIDADE: N-PUZZLE

Para o algoritmo implementado iremos assumir 2 parâmetros para realizar a análise de complexidade do caso geral, são eles: d (quantidade de níveis da árvore) e b (fator de ramificação ou número máximo de sucessores imediatos do nó). Assumindo-se um fator de ramificação constante b tem-se uma árvore como na Figura 3.

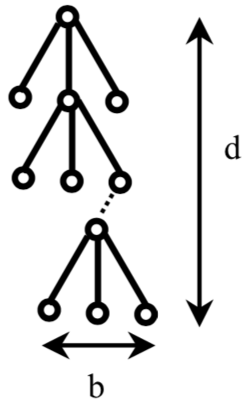

Figura 3: Árvore de busca com fator de ramificação b e profundidade d.

Se cada nó tem b filhos então a raiz ($d = 0$) tem 1 filho:

Raiz tem b filhos

Nível 1 tem $b * b$ filhos

Nível 2 tem $b * b * b$ filhos

...

Último nível tem $(b^{d-1}) * d$ filhos

Seja N o número de nós da árvore, temos que $N = 1 + b^2 + b^3 + ... + b^d$ *(I)* e pode-se escrever N como $bN = b + b^2 + b^3 + ... + b^{d+1}$ *(II)*. Subtraindo-se II de I tem-se:

$(1-b)N = b^{d+1} - 1$

$N = \frac{b^{d+1}-1}{1-b}$, que é o mesmo que $O(b^d)$ para complexidade de espaço no pior caso (em que todos os nós criados são analisados).

Dividindo-se o algoritmo capaz de calcular a solução ótima (Algoritmo 1) tem-se quatro blocos. O bloco 1 está compreendido entre as linhas 4 e 7, o bloco 2 entre as linhas 9 e 11, o bloco 3 na linha 15 e o bloco 4 entre as linhas 17 e 19. Pode-se reescrever o algoritmo conforme o Algoritmo 3 e concluir que a complexidade de tempo para a solução implementada é $O(b^{d+1}$.

Algorithm 3 Análise de complexidade findMinCost()

bloco 1 //Custa $c_1 + lg$(tamanho da fila)
enquanto q não é vazia **faça** // Custa b^{d+1} vezes no pior caso
  bloco 2 // Custa o tamanho da fila + lg(tamanho da fila) + d
  para i ← 1 **até** todos os filhos de u **faça** // Custa $c_2 * d$
    se é estado final **então**
      bloco 3 // Custo d
    senão
      bloco 4 // Custo d
    fim se
  fim para
fim enquanto

ANÁLISE DE COMPLEXIDADE: 8-PUZZLE

Durante a execução do algoritmo, com os testes providos pelo monitor, medidas de tempo, memória alocada e quantidade de nós analisados foram coletadas. A tabela abaixo mostra os resultados obtidos para os 9 casos de teste. A biblioteca *tracemalloc* foi utilizada para realizar as medidas de memória durante a execução do programa.

Tabela 1: Tempo de execução e quantidade de nós analisados por caso de teste.

In	Custo total	Tempo (segundos)	# nós examinados	Memória alocada
1	3	0.0070	6	1240 B
2	5	0.0307	15	1936 B
3	8	0.1303	34	6496 B
4	12	0.9230	97	21.1 KiB
5	14	1.5907	130	28.6 KiB
6	16	6.0482	272	61.9 KiB
7	20	14.1637	1833	428 KiB
8	22	22.6761	1977	462 KiB
9	26	72.8479	3615	845 KiB

Bibliografia

[1] KUNKLE, D. R. Solving the 8 puzzle in a minimum number of moves: An application of the a* algorithm. *Introduction to Artificial Intelligence* (2001).