ANÁLISE DE ALGORÍTMOS: Trabalho 2

Por David Beyda e Luiz Carlos Rumbelsperger

# Tarefa 1:

Essa tarefa pede um procedimento para construir um grafo onde os nós representam os estados do jogo, e as arestas representam se é possível ir de uma configuração para a outra em apenas um passo, seguindo as regras do jogo.

Para isso, decidimos representar o grafo como uma lista de tuplas, onde em cada tupla, o primeiro elemento guarda o estado representado pelo nó, e o segundo elemento guarda uma lista de vizinhos daquele nó:

G = [(estado, [vizinhos])]

Para gerar esse grafo, geramos primeiro todas as permutações possíveis dos números de 0 a 8 (0, 1, 2, 3, 4, 5, 6, 7, 8). Esses são todos os estados possíveis do jogo. Podemos facilmente calcular o número de estados (ou número de nós do grafo) por análise combinatória. Esse número é 9! = 362880.

Em seguida, para cada estado, analisamos onde está a casa vazia, e construimos os seus vizinhos. Então, guardamos seus vizinhos na tupla que representa esse estado. Podemos agora conferir o número de nós do grafo, pelo comprimento da lista G. O número está correto, e coincide com o resultado teórico (9!).

Além disso, podemos calcular também as arestas do grafo. Sabemos que o número de arestas é 2\*(somatório dos graus de todos os nós). podemos calcular o somatório dos graus dos nós da seguinte forma:

1. Se o espaço vazio está no meio do grafo, podemos mover 4 peças diferentes para o centro, ou seja, ele possui 4 vizinhos. além disso, isso funciona para qualquer permutação das outras peças, ou seja, funciona para 8! nós. Ou seja, temos 4 \* 8! possibilidades.

2. Se o espaço vazio está em uma das 4 quinas do jogo: podemos mover apenas 2 peças, mas isso funciona para 4 quinas, e 8! nós em cada quina. Logo, temos 2 \* 4 \* 8! possibilidades.

3. Se o espaço vazio não está no meio nem nas quinas, está colado nas paredes do tabuleiro, ocupando a posição do meio nas paredes. Logo, pode-se mover 3 peças, e isso funciona para 4 posições desse tipo, e 8! nós em cada. Logo, temos 3 \* 4 \* 8!

Somando tudo: 8! \* (1\*4 + 2\*4 + 3\*4) = 24 \* 8!

O número de arestas, então é isso divido por dois: 12 \* 8! = 483840.

Esse número também pode ser confirmado no procedimento. Basta percorrermos toda a lista G, somando o número de vizinhos de cada nó, e no final dividir esse número por 2.

A tarefa 1 pede também um exemplo de nós que estão conectados por uma aresta. Esses seriam:

[0 1 2]           [1 0 2]

[3 4 5]    e    [3 4 5]

[6 7 8]         [6 7 8]

E pede também um exemplo de nós que não tem aresta entre eles:

[0 1 2]            [1 2 0]

[3 4 5]     e     [3 4 5]

[6 7 8]            [6 7 8]

# Tarefa 2:

Nessa tarefa, é pedido o código principal da BFS (o loop interno da BFS), e que ele seja utilizado para contar as componentes conexas do nosso grafo. Para isso, fizemos um vetor de tamanho 9!, para guardar quais nós foram visitados. Além disso, definimos uma função hash para, a partir de um estado, descobrir o índice dele no vetor de visitados. Assim, basta iterar sobre os nós do grafo, rodando BFS caso o nó não tenha sido visitado ainda, e contar quantas vezes a BFS foi chamada. Além disso, a BFS deve marcar no vetor de visitados cada estado que ela visitou.

A função de BFS é a seguinte:

def BFS(G, s, visited):

queue = [s]

# distance from s, parent.

visited[hash(s)] = (0, None)

while queue:

u = queue.pop(0)

hu = hash(u)

for v in G[hu][1]:

hv = hash(v)

if not visited[hv]:

visited[hv] = (visited[hu][0] + 1, hu)

queue.append(v)

Ela guarda no vetor de visitados o tamanho do caminho até aquele nó, e quem foi o pai que originou esse caminho. Isso nos servirá mais a frente, na tarefa 3.

Além disso, constatou-se que o grafo possui duas componentes conexas. Isso significa que, seguindo as regras do jogo, não é possível atingir todas as configurações possíveis. Isso é interessante porque normalmente quando se compra esse jogo, ele já vem montado. Logo, se quebrarmos uma peça do jogo, e a encaixarmos em outro lugar, de forma que o novo estado esteja na outra componente conexa, não será mais possível solucionar o jogo apenas seguindo as regras.

# Tarefa 3:

A tarefa 3 pede para encontrarmos a configuração na qual o caminho mais curto para a solução do jogo é o maior possível, ou seja, o estado a partir do qual é "mais difícil" de se solucionar o jogo.

Para isso, basta rodarmos a BFS a partir da configuração que representa a solução final do jogo. Depois disso, no vetor de visitados, apenas para os nós visitados, procuramos o(s) nó(s) com a maior distância desde a solução.

A partir desses nós, podemos saber quem é o "pai" dele no caminho da solução, e o "pai do pai", e ir jogando toda a linhagem de pais em uma lista, que representa o passo a passo da solução. Assim, pode-se também visualizar a solução depois.

Feito esse procedimento, encontramos duas configurações iniciais mais difíceis, que necessitam do mesmo número de passos: 31 passos.

Os problemas são:

[8 6 7]

[2 5 4]

[3 0 1]

E

[6 4 7]

[8 5 0]

[3 2 1]

onde 0 representa o espaço vaizo.