Solucionador do N-Puzzle: uma aproximação usando pesquisa em largura e em profundidade em grafos

Leandro L. A. Vieira,[1] **Mateus P. Silva**[2]

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-caf) – Florestal – MG – Brasil

(leandrolazaro[1], mateus.p.silva[2])@ufv.br

Resumo. *Este trabalho trata-se da análise de métodos de solução do jogo N-Puzzle, especificamente os de busca em largura e em profundidade utilizando grafos, com melhoria nos critérios de parada para encontrar a solução mais rápida do desafio. Analisar-se-a também tais técnicas em contexto amplo, tomando como exemplo prático os algoritmos propostos. Comprova-se que tais procedimentos não são satisfatórios mesmo com os incrementos feitos, principalmente o segundo citado.*

1. Introdução

A teoria dos grafos é um dos principais métodos de modelagem de algoritmos graças à sua simplicidade, o que a faz tangenciar a universalidade na solução dos mais variados problemas. Ademais, os avanços tecnológicos fazem com que o tamanho dos dados cresçam exponencialmente todos os anos, tornando obrigatório o estudo de formas mais eficientes de operações computacionais, especialmente as com grafos.

Dois dos algoritmos mais discutidos na literatura são os de busca em largura e em profundidade. Ambos apresentam a característica de *busca não-informada*, que é a capacidade de serem executados sem o conhecimento/geração do grafo inteiro. Tal peculiaridade torna-os úteis para grafos que não são possíveis de serem alocados em memória principal ou, ainda, que são infinitos.

Afim de tornar os testes mais fidedignos, tomamos como base o desafio de solucionar o quebra cabeças N-Puzzle, que pode gera grafos infinitos. Propomos algumas melhorias para os dois algoritmos, com finalidade de torná-los menos ineficientes.

2. O jogo N-Puzzle e sua modelagem em grafos

Trata-se de uma pequena caixa quadrada, com N+1 espaços cobertos por N peças também quadradas, que contem números, figuras ou desenhos, e um espaço vazio para que se possa movimentá-las. As peças devem ser ordenadas na caixa em sequência, deslocando-se as peças ocupando o espaço vazio, fazendo quantos movimentos quiser sem retirá-las da caixa, colocando-as na ordem e deixando o N+1 quadradinho vazio.

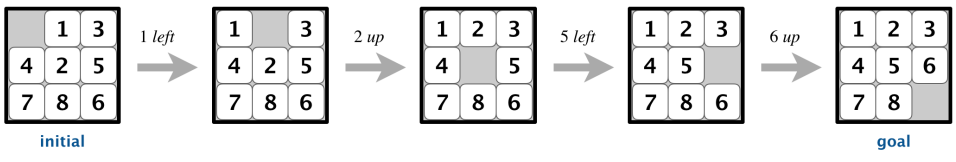

Figura 1. 8-Puzzle sendo resolvido

A modelagem de grafos foi feita tomando como vértices cada configuração do jogo, e como aresta cada movimento singular dos quadradinhos.

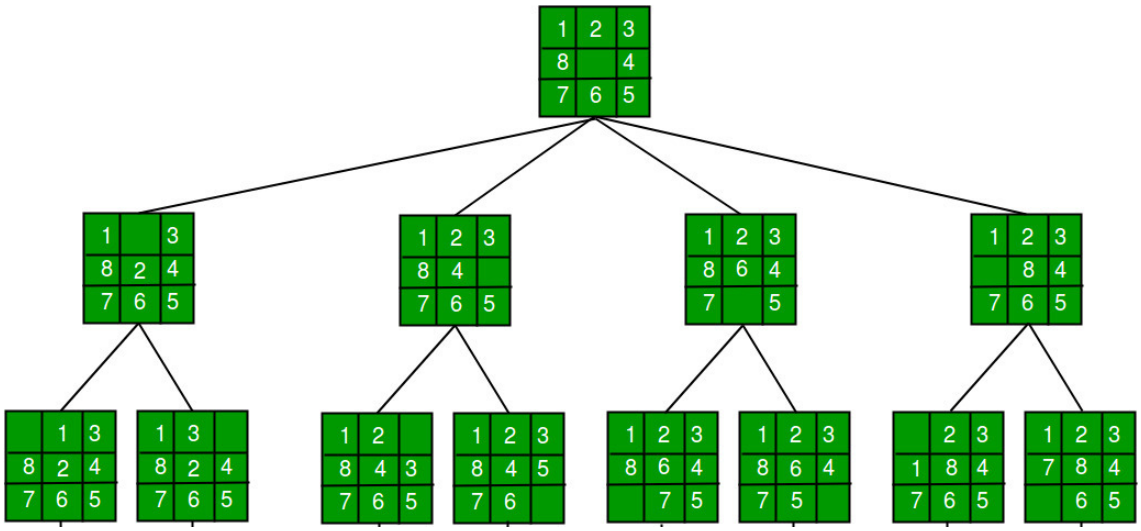

Figura 2. Grafo utilizado

3. Modelagem computacional

Como o grafo resultante é uma *árvore* infinita, a melhor estrutura de dados possível é a *árvore 4-ária*, estrutura que contém até quatro referências representando os quatro movimentos possíveis, além de uma para o puzzle gerador do nó, numa espécie de árvore duplamente encadeada.

O espaço vazio de cada jogo foi substituído pelo número zero para manter todos os dados do mesmo tipo - isto é, inteiros - e a posição do valor zero foi salva também para diminuição do processamento, embora sacrifique um pouco a memória principal. A altura da árvore é salva em cada um de seus nós, além do movimento utilizado para gerar aquele *filho*.

Além disso, é fácil perceber que pares de movimentos inversos criam a mesma configuração para o jogo, e a repetição de vértices é prejudicial aos algoritmos por serem dados redundantes que precisam de processamento também redundante, portanto esses nós foram removidos. O jogo possibilita nós idênticos em partes diferentes do grafo, porém tratar isso tonar-se-ia necessário alguma estrutura capaz de armazenar de forma eficiente permutações, o que até onde sabemos não encontra-se presente na literatura.

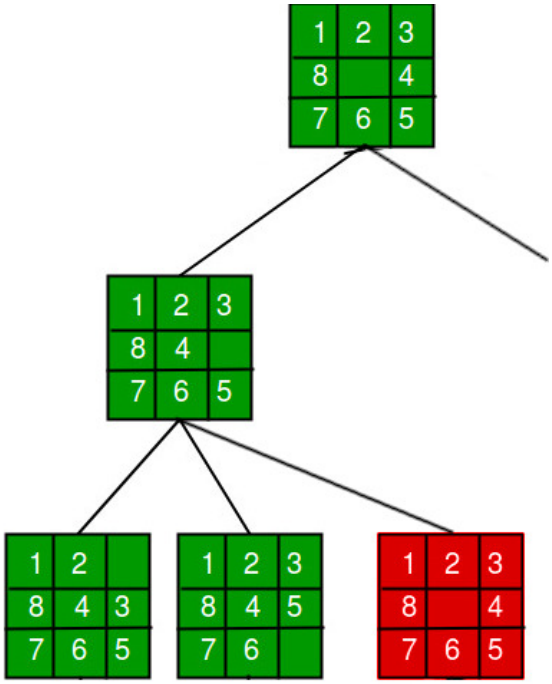

Figura 3. Note que o N-Puzzle resultado de um par de movimento inverso (esquerda-direita) é exatamente igual ao "avô"

3.1. Verificação de vitória e de solucionabilidade

Para verificar a vitória, foi implementado um algoritmo que primeiro verifica se o zero está na posição N,N - utilizando, para isso, os dados auxiliares previamente citados - e depois são verificadas as posições de todos os outros valores percorrendo o jogo.

Alguns N-Puzzles são insolucionáveis, pois o número de inversões necessárias são ímpares. Esse teste também foi implementado utilizando um método conhecido na literatura.

3.2. O caminho para a raiz dado um vértice

Ambos os algoritmos de busca retornam uma referência para o vértice vitorioso. Para isso, foi desenvolvido um método que retorna o caminho para a raiz dado um nó qualquer. Utilizando a estrutura *pilha*, todos os N-Puzzles desde o escolhido até a raiz são empilhados, tomando o percurso com a referência para o pai. Ao fim, toda a pilha é desempilhada, exibindo assim o caminho.

3.3. A busca em largura

O algoritmo de busca em largura foi implementado *ipsis litteris*, utilizando para isso a geração do grafo com os movimentos possíveis em tempo de execução. A estrutura auxiliar escolhida para armazenar os nós foi a fila, escolha extremamente comum na literatura.

A raiz do grafo é enfileirada, e enquanto a fila não estiver vazia, o nó é testado se é vitorioso. Caso seja, a referência para ele é retornada e a função termina. Caso o contrário, ele é desenfileirado e os *filhos* dele são gerados e enfileirados.

3.4. A busca em profundidade

O algoritmo de busca em profundidade foi implementado usando sua abordagem recursiva, tomando-se o cuidado de evitar tipos de dados muito grandes e/ou variáveis desnecessárias.

O método inicia-se assumindo que o menor caminho para a vitória no jogo conhecido precisa de infinitos movimentos e que a referência para esse jogo é nula. Antes de realizar a pesquisa em um nó, é perguntando se esse nó possui uma altura maior que o menor caminho para a vitória. Caso possua, a chamada recursiva é encerrada. Caso não, é testado se o jogo é vitorioso. Se sim, o menor caminho possível é atualizado para a altura do nó, e a referência para o jogo vitorioso é modificada. Se não, os *filhos* do nó são gerados, e a função é chamada recursiva para eles. Ao fim, é retornada a referência para o jogo vitorioso com o menor número de movimentos.

4. As escolhas para a implementação prática

Como tratam-se de algoritmos reconhecidamente custosos para a literatura, e estruturas relativamente complexas com regras bem definidas - tanto os nós da árvore como o jogo em si - escolhemos a linguagem C++, devido ao seu paradigma orientado a objetos e seu tempo de execução médio bastante satisfatório devida sua compilabilidade.

Primeiramente foi criada uma classe com todas as regras bem definidas do jogo, com a possibilidade de entrada de um jogo qualquer por meio de arquivo. Também foi implementada a possibilidade de criação de um jogo aleatório, tomando como base o jogo vitorioso e fazendo uma quantidade aleatória de movimentos possíveis, garantindo assim um jogo resolvível caso o original também seja.

```
class Slider
{
private:
    short int matrix[slidersize][slidersize];
    short int xZero, yZero;
protected:
    bool movableDown();
    bool movableUp();
    bool movableLeft();
    bool movableRight();

    void moveDown();
    void moveUp();
    void moveRight();
    void moveLeft();
    [...]
```

Depois, uma nova classe foi desenvolvida que herda as características da *Slider*, porém com os comportamentos de um nó de grafo.

```
class SliderGraph : private Slider
{
private:
```

```
    SliderGraph * up;
    SliderGraph * down;
    SliderGraph * left;
    SliderGraph * right;
    SliderMovementType moved;

    short int level;
    SliderGraph * father;

    [...]

public:

    [...]
    string widhtSearch();
    string depthSearch();
};
```

Utilizamos tanto a *fila* quanto a *pilha* disponibilizadas por padrão na linguagem, devida a alta eficiência das duas. Destaca-se ainda, do código, o método usado para a geração de filhos que evita repetições sem realizar nenhum tipo de pesquisa:

```
void SliderGraph::createChildren()
{
    short int childrenLevel = this->level + 1;

    if(this->moved != movedUp && Slider::movableDown())
    {
        this->down = new SliderGraph(*this);
        this->down->moveDown();

        [...]

    }

    if(this->moved != movedDown && Slider::movableUp())
    {
        this->up =  new SliderGraph(*this);
        this->up->moveUp();

        [...]

    }

    [...]
}
```

5. Testes executados e seus resultados

Foram executados testes em quatro configurações do jogo, as quais serão descritas a seguir, respectivamente *s1*, *s2*, *s3* e *s4*:

1	2	3
4	5	0
7	8	6

2	3	5
6	7	1
4	0	8

0	4	3
2	1	5
7	8	6

1	6	5
2	7	0
3	8	4

Todas essas configurações, com exceção de *s1*, foram geradas de forma aleatória. O número de passos mínimo para a solução são de, respectivamente: 1, 19, 8 e 25

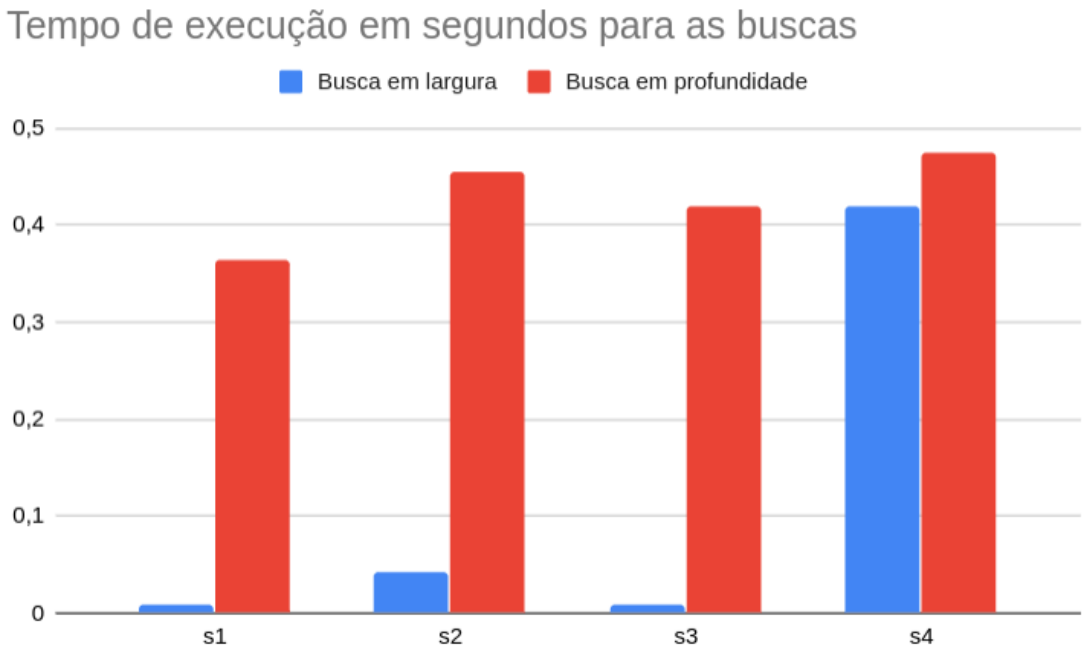

Figura 4. Tempo de execução em segundos no eixo Y. Quanto menos, melhor

Note que em todos os testes, a busca em largura se mostrou muito mais eficiente. O único teste em que os resultados foram próximos foi o *s4*. Isso aconteceu pois o algoritmo de busca em profundidade foi configurado para realizar prioritariamente recursões que favoreceram, por pura aleatoriedade, o teste criado. Entretanto, a probabilidade disso acontecer é muito pequena (uma em cada vinte e quatro, considerando que há quatro fatorial possibilidades de prioridade de recursão).

Embora a busca em largura tenha se mostrado mais eficiente, ela ainda mantém-se longe em questões de desempenho de outras abordagens de solução desse problema, como a que usa algoritmos genéticos.

6. Hardware usado

Todos os testes foram executados em um processador *I7-7700HQ*, 16GB de memória RAM DDR4 Adata, utilizando o sistema operacional *Kubuntu 19.04* com *kernel Linux 5.0.0-27-Ubuntu* (versões mais recentes disponíveis durante a elaboração do trabalho).

TLB/Cache details:	64-byte Prefetching Data TLB: 1-GB pages, 4-way set associative, 4 entries Data TLB: 4-KB Pages, 4-way set associative, 64 entries Instruction TLB: 4-KByte pages, 8-way set associative, 64 entries L2 TLB: 1-MB, 4-way set associative, 64-byte line size Shared 2nd-Level TLB: 4-KB / 2-MB pages, 6-way associative, 1536 entries. Plus, 1-GB pages, 4-way, 16 entries

Cache details				
Cache:	L1 data	L1 instruction	L2	L3
Size:	4 x 32 KB	4 x 32 KB	4 x 256 KB	6 MB
Associativity:	8-way set associative	8-way set associative	4-way set associative	12-way set associative
Line size:	64 bytes	64 bytes	64 bytes	64 bytes
Comments:	Direct-mapped	Direct-mapped	Non-inclusive Direct-mapped	Inclusive Shared between all cores

7. Pre-requisitos para execução dos algoritmos

- *GCC*
- *Linux* (recomendado)

Para repetir os testes, digite “make” para compilar e “make run” no terminal para abrir um menu auto-explicativo.

Note que, dependendo da dificuldade do jogo, talvez seja necessário aumentar o número de recursões do algoritmo de busca em profundidade, ou o de nós no de busca em largura. para isso, edite o arquivo “constants.hpp”. No mesmo arquivo, também é possível escolher o valor de N, ou seja, o tamanho do lado do jogo (por padrão, N=3).

8. Considerações finais

Comprova-se, então, que ambos algoritmos são ineficazes, mesmo com melhorias de critério de paradas propostos. É bastante óbvio, pela própria natureza dos algoritmos, que a busca em largura é mais eficiente para encontrar algum vértice com comportamento específico - no caso do jogo, um puzzle vitorioso - com a menor distância da raiz, pois procura a árvore de cima para baixo, portanto o primeiro vértice será automaticamente o desejado ou, no mínimo, será um dos desejados.

A evitação de repetições óbvias - de movimentos pares inversos - foi extremamente benéfica para os algoritmos, porém o grafo apresenta vértices iguais em posições bem distantes. Evitá-las de forma eficiente seria muito interessante. Recomenda-se a pesquisa de alguma estrutura de dados eficiente para salvar permutações, o que possibilitaria armazenar todos os N-Puzzles já testados, e assim com um simples teste de “contém” poderiam ser evitadas todas as repetições.

Foi visto na prática, também, o custo altíssimo que os algoritmos de busca têm. A escolha da linguagem de programação e das estruturas foi muito feliz. Acreditamos que, caso fosse escolhida uma linguagem interpretada e/ou de maior nível (como, por exemplo, *Python* ou *Java*), o tempo de execução seria extremamente prejudicado. Por outro lado, a escolha de uma mais próxima do código de máquina obrigaria-nos a implementar *filas* e *pilhas*, que provavelmente não seriam tão eficientes como as amplamente consolidadas disponíveis no *C++*. Entretanto, em trabalhos futuros seria interessante analisar o desempenho utilizando paralelismo, devido a natureza não bloqueante da geração dos filhos dos nós.