# Resolução *Puzzle* 8 utilizando algoritmos de busca

Gustavo Aparecido de Souza Viana

***Abstract*— Os algoritmos de busca são utilizados em diversos segmentos, com intuíto de fazer uma busca simples ou também na tentativa de resolver um problema. Portanto, neste trabalho será analisado o comportamento dos algoritmos de BFS (*breadth-first search*), DFS (*depth-first search*), *hill-climbing* e *A** aplicados em uma árvore n-ária com objetivo de achar a solução do jogo *Puzzle* 8. Foi possível concluir que dependendo da ordem que é gerado os novos estados (possíveis jogadas), ou seja, se adicionar primeiro a árvore o movimento esquerdo e depois direito ou vice-versa, isso tem um grande impacto na resolução do DFS.**

## I. INTRODUÇÃO

Existem uma série de algoritmos de busca aplicados em diversos tipos de estruturas de dados, árvore, lista encadeada simples ou dupla, grafo e entre outros. O objetivo desses algoritmos é executar a busca de uma forma otimizada e eficiente em termos de tempo de resposta.

Podemos citar uma séries de exemplos onde a utilização do DFS e BFS, A* e hill climbing pode ser aplicada, sendo um deles, a detecção de ciclo em um grafo, resolução de jogos (como o *Puzzle* 8) e além de outros.

Partindo dos algoritmos de busca citados previamente, o objetivo deste trabalho é aplicar o algoritmo BFS, DFS, A* e hill climbing para resolver o problema do jogo *Puzzle* 8 chegando na solução final com o melhor caminho percorrido e comparar o tempo de resposta de cada algoritmo.

## II. CONCEITOS FUNDAMENTAIS

Nesta seção, será apresentado os conceitos básicos de árvore n-ária, *depth-first search*, *breadth-first search*, *Hill-climbing* e *A**. Outros conceitos que foram utilizados na implementação, como lista ligada dinâmica, fila e pilha dinâmica podem ser encontrados no artigo de [2].

### A. Árvore n-ária

A árvore é uma estrutura de dados que permite você organizar seus dados de forma ordenada ou não, podendo ser binária ou não e sendo alocada de forma dinâmica, permitindo escabilidade.

A árvore funciona com encadeamento de nós, onde o nó possui o seu valor ou objeto, ponteiro para o nó pai e uma lista de ponteiros para os nós filhos, permitindo o encadeamento de nós como podemos ver na Figura 1.

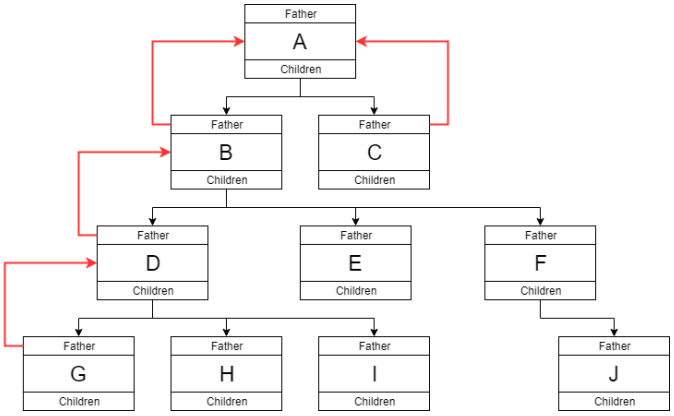


Fig. 1: Representação da árvore

A operação de inserção e remoção poder ser feita no em qualquer parte da árvore, portanto essa inserção pode acontecer na raíz com apenas 1 elemento na árvore, no início, no meio e no fim da árvore. Na hora da implementação deverá ser tratado cada cenário de inserção e remoção separadamente, pois necessita reconfigurar os ponteiros de "pai e filhos" do nó que será removido e inserido. A inserção e remoção tem complexidade de ordem *O(log N)*.

A operação de busca tem como complexidade de ordem *O(log N)*. Consequentemente, a operação de atualização tem mesma complexidade da busca, pois necessita buscar o elemento antes de atualizar.

### B. Depth-first search - DFS

Como o próprio nome diz *depth-first* foca na busca em profundidade, portanto a essência do algoritmo é visitar sempre o primeiro filho de cada nó, como podemos ver abaixo na Figura 2.

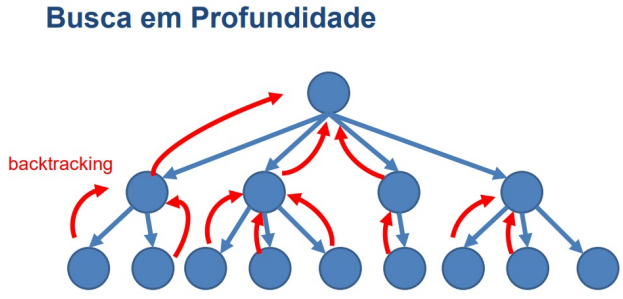


Fig. 2: Representação de busca DFS - busca em profundidade

### C. Breadth-first search

*Breadth-first* foca na busca em largura, portanto a essência do algoritmo é visitar sempre os nós vizinhos como se

fossem fileiras baseadas na altura da árvore, como podemos ver abaixo na Figura 3.

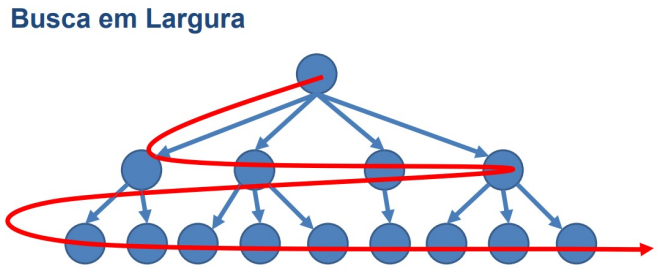


Fig. 3: Representação de busca BFS - busca em largura

*D. Comparação entre Depth-first search e Breadth-first search*

A Figura 4 representa a comparação entre os dois algoritmos.

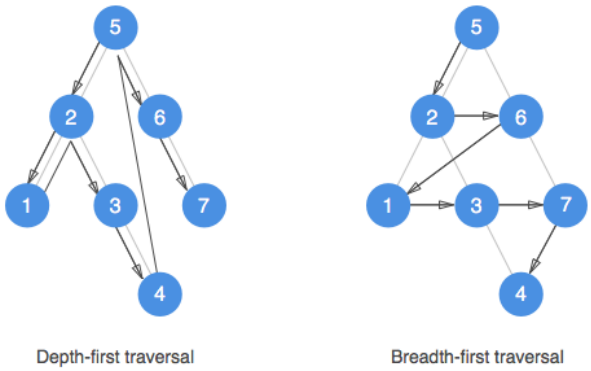


Fig. 4: Comparação entre Depth-first search e Breadth-first search

*E. Hill-climbing*

O algoritmo de Hill-climbing necessita de uma heuristica para que a exploração dos nós sejam priorizadas, portanto a essência do algoritmo é explorar sempre o nó filho com menor valor de heuristica baseada no nó pai o gerou, e assim sucessivamente. Importante resaltar que caso não exista nenhum valor de heuristica o algoritmo falha. Abaixo na Figura 5 exemplo de Hill-climbing.

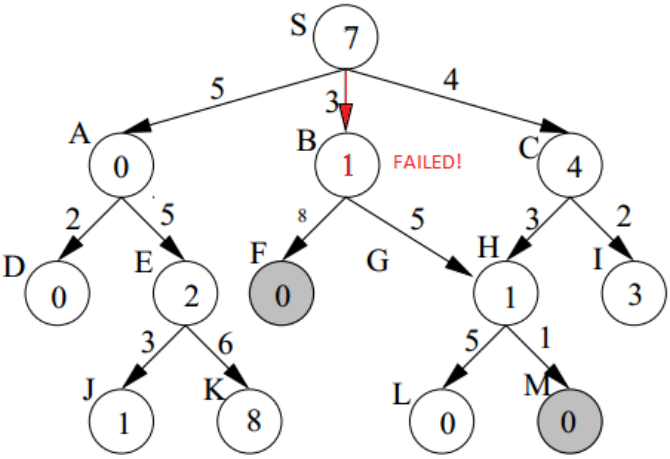


Fig. 5: Representação de busca Hill-climbing

*F. A**

O algoritmo de A* necessita de uma heuristica para que a exploração dos nós sejam priorizadas, portanto a essência do algoritmo é visitar o menor valor de heuristica em relação ao nó pai de todos os nós filhos que estão disponíveis para exploração. Abaixo na Figura 6 exemplo de A*.

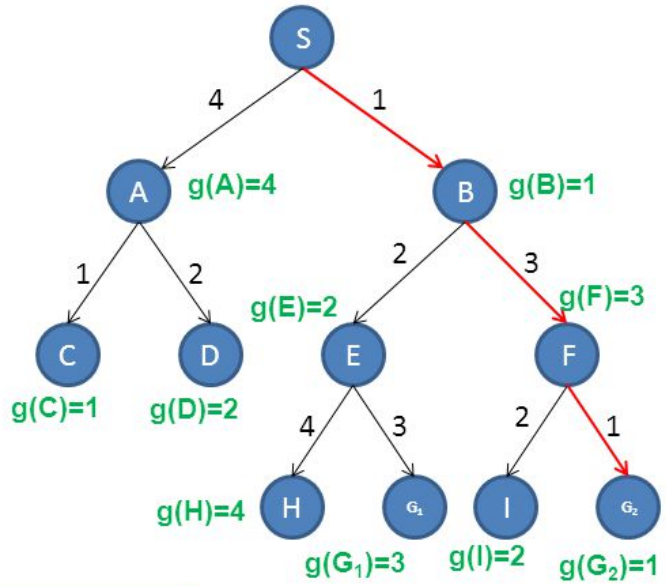


Fig. 6: Representação de busca A*

## III. METODOLOGIA

Nesta seção será apresentado a metodologia utilizada para implementar árvore, DFS, BFS, hill climbing e A*, em subsessões diferentes, incluindo pseudocódigos e UML das classes utilizadas implementadas em C++. O código fonte pode ser encontrado em https://github.com/apparecidoo/ProgramacaoCientifica.

*A. Árvore n-ária*

Foi criada uma classe template chamada *Tree* que representa uma árvore, onde o T do template é passado para o tipo

do conteúdo que será armazenado por cada nó. Esta classe é composta por propriedades e métodos.

As propriedades são *root* (nó que representa a raiz da árvore), *id* (chave primária para identificar cada nó que será criado), *queue_bfs_list* (lista de nós a serem explorados pelo bfs), *stack_dfs_list* (lista de nós a serem explorados pelo dfs), *explored_list* (lista de nós que já foram explorados).

Os métodos são *set_child_properties* (seta as propriedades do filho baseado no nó pai), *compare* (compara o conteúdo de dois nós), *print_content* (exibir conteúdo do nó), *print_node* (exibe as propriedades e o conteúdo do nó), *back_tracking* (exibe todos os nós pais). Nenhum dos mesmo apresentará pseudocódigo pois todos são métodos virtuais que serão sobreescritos na classe Puzzle.

### B. Puzzle

A classe *Puzzle* herda métodos da classe *Tree* onde os métodos de busca são reimplementados pois conforme os nós são explorados mais possibilidades (inserindo mais nós) são disponibilizadas para exploração. O construtor da classe possui o objetivo do seu puzzle e o tipo do mesmo, como por exemplo *Puzzle8* para que na chamada da busca, o mesmo seja usado como referência de *Goal*.

O pseudocódigo 1 representa o algoritmo criar novas possiblidades a partir de um nó, ou seja, cria novas possíveis movimentações de peças. O método tenta fazer as movimentações abaixo, caso consiga a mesma é adicionada, caso contrário é pulada

**Algorithm 1:** create_children_nodes(TreeNode* node) - Create Children Nodes

```
move_zero_left(node);
move_zero_down(node);
move_zero_right(node);
move_zero_up(node);
```

O pseudocódigo 2 representa o algoritmo busca Depth-first search.

**Algorithm 2:** search_dfs(T test) - Depth-first search

```
if !is_solvable(test) then
    return NULL;
end
stackList.push(root);
while stackList.isEmpty() do
    node = stackList.pop();
     exploredList.enqueue(node.content);
    if compare(node.content, goal) then
        return node;
    end
    createChildrenNodes(node);
    if node.hasChildren() then
        child = node.childrenNodes.getRoot();
        while child != NULL do
            if !explored(child.content.content) and
             !stackToExplore(child.content.content) then
                stackList.push(child.content);
            end
            child = child.nextNode;
        end
    end
end
return NULL;
```

O pseudocódigo 3 representa o algoritmo busca Breadth-first search.

**Algorithm 3:** search_bfs(T test) - Breadth-first search

```
if !is_solvable(test) then
    return NULL;
end
queueList.enqueue(root);
while queueList.isEmpty() do
    node = queueList.dequeue();
     exploredList.enqueue(node.content);
    if compare(node.content, goal) then
        return node;
    end
    createChildrenNodes(node);
    if node.hasChildren() then
        child = node.childrenNodes.getRoot();
        while child != NULL do
            if !explored(child.content.content) and
             !queueToExplore(child.content.content)
             then
                stackList.enqueue(child.content);
            end
            child = child.nextNode;
        end
    end
end
return NULL;
```

O pseudocódigo 4 representa o algoritmo busca A*.

**Algorithm 4:** search_a_star(T test) - A*

```
if !is_solvable(test) then
    return NULL;
end
list.addLast(root);
while list.isEmpty() do
    node = getNextNodeToExplore();
     exploredList.addLast(node.content);
    if compare(node.content, goal) then
        return node;
    end
    createChildrenNodes(node);
    if node.hasChildren() then
        child = node.childrenNodes.getRoot();
        while child != NULL do
            if !explored(child.content.content) and
             !listToExplore(child.content.content) then
                list.addLast(child.content);
            end
            child = child.nextNode;
        end
    end
end
return NULL;
```

O pseudocódigo 5 representa o algoritmo busca hill climbing.

**Algorithm 5:** search_hill_climbing(T test) - Hill climbing

```
if !is_solvable(test) then
    return NULL;
end
node = root; node_to_validate = node;
while node != NULL do
    exploredList.enqueue(node.content);
    if compare(node.content, goal) then
        return node;
    end
    createChildrenNodes(node);
    if node.hasChildren() then
        child = node.childrenNodes.getRoot();
         node_to_validate = node; node = NULL;
        while child != NULL do
            if !explored(child.content.content) and
             child.content.f_score <=
             node_to_validate.f_score then
                node = child.content;
            end
            child = child.nextNode;
        end
    end
end
return NULL;
```

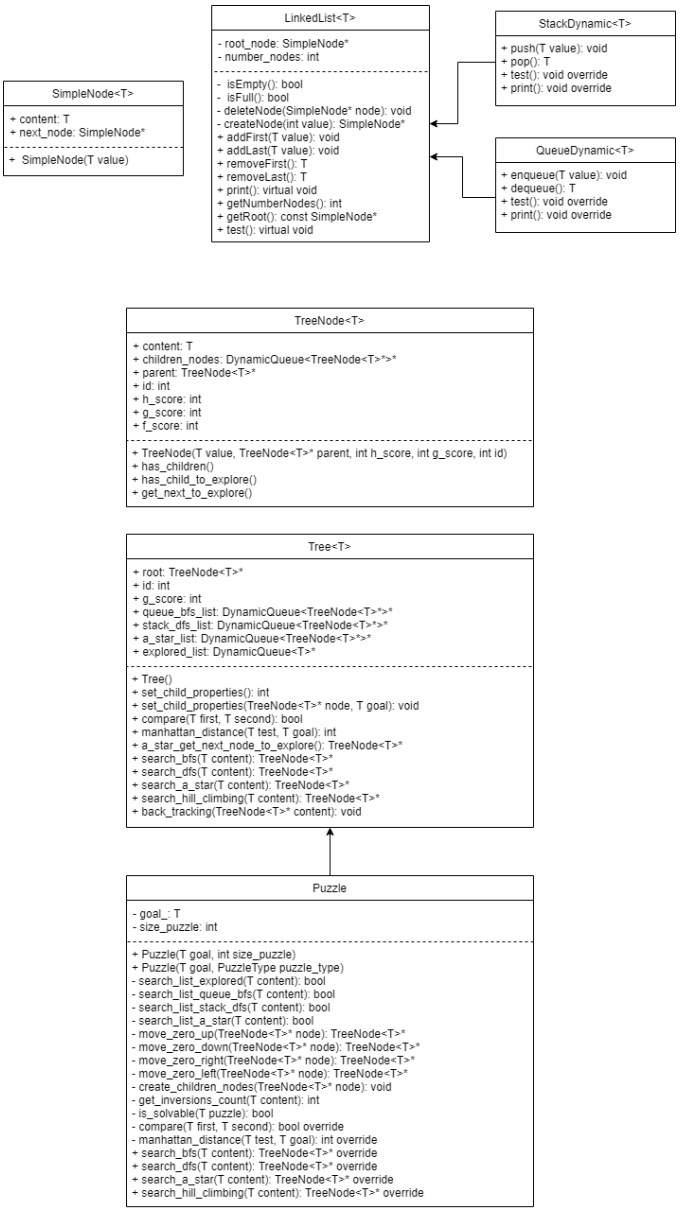


Fig. 7: Representação da UML completa

## IV. EXPERIMENTOS E RESULTADOS

Nesta Seção será apresentado os experimentos e seus respectivos resultados. Foram feitas duas iterações com ordens diferentes de movimentações de peça no momento de criação de novos estados, a primeira possui a sequência *left move, down move, right move e up move*, já a segunda, *down move, left move, right move e up move*. Os resultados das duas iterações estão na Figura 8 e Figura 9.

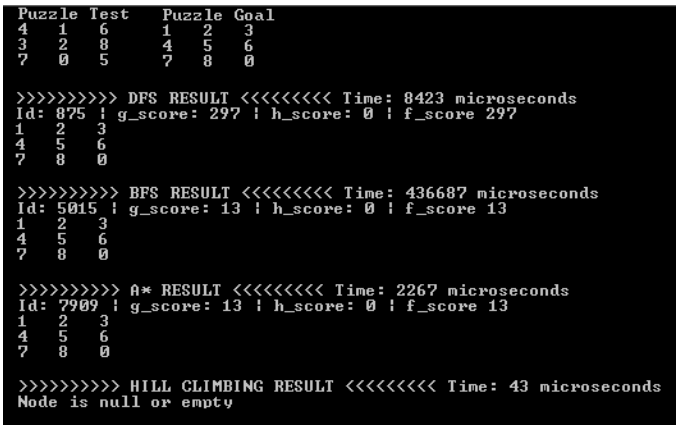


Fig. 8: Representação dos resultados do Puzzle na primeira iteração

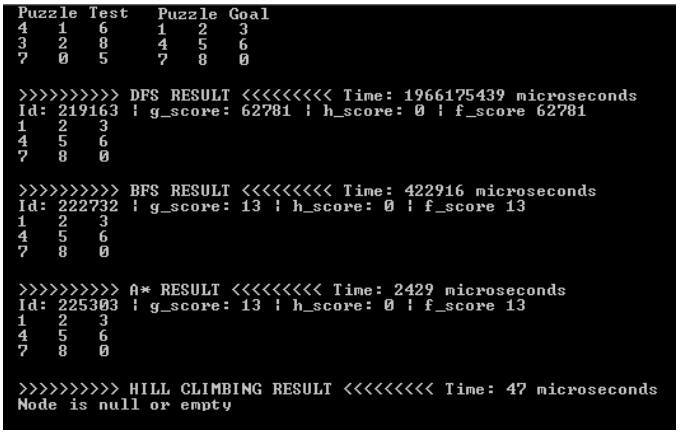


Fig. 9: Representação dos resultados do Puzzle na segunda iteração

## V. TRABALHOS RELACIONADOS

Nesta seção será apresentado alguns trabalhos que utilizaram listas ligadas, pilhas ou filas.

Em [6] o autores propõem transformar a estrutura de dados árvore em um vetor numérico multidimensional aproximado que codifica a informação da estrutura original. Os resultados evidenciaram a redução do custo computacional, e afirmaram que o método é adequado para acelerar o processamento de consultas de similaridade em grandes árvores com grandes conjuntos de dados.

Em [1] os autores descrem um sistema de otimização de árvore chamado *XGBoost*, com isso eles propõem um novo algoritmo de esparsidade para dados esparsos e esboço quantil ponderado para aproximar o aprendizado da árvore. Os resultados mostraram que os padrões de acesso ao cache, a compactação de dados e o sharding são elementos essenciais para a criação de um sistema escalonável de ponta a ponta para o aprimoramento de árvores.

Em [5] o autor propõe um estudo do algoritmo de busca *Depth-first search* DFS e de algoritmos lineares aplicados em grafo. O DFS é um algoritmos eficiente para encontrar os componentes fortemente conectados de um grafo direcionado e para encontrar os componentes biconectados de um grafo não direcionado.

Em [3] o autor faz um estudo sobre os algoritmos Dijsktra e breadth-first search, apresentando suas caracteristicas a partir da aplicação do mesmo em grafos.

Em [4] o autor faz um estudo sobre a aplicação de buscas locais e buscas globais, no caso buscas locais é utilizado o *hill-climbing*. Em conclusão, *hill-climbing* fornece uma estratégia poderosa para explorar espaços de busca combinatória. Em muitas aplicações, o hill-climbing supera a busca global.

Em [7] o autor propõe a comparação do algoritmo Djikstra e o A* para a resolução de caminhos para carros, onde os mesmos estão em movimento paralelamente e além de possuir obstáculos dinâmicos. Um algoritmo integrado *A-STAR-Dijkstra* é promovido para fazer múltiplos carros se movendo paralelamente sem colisão ou deadlock. Ambos os dois algoritmos são otimizados também.

## VI. CONCLUSÃO

Após a análise do conceito e resultados obtidos após a busca feita por *DFS* revelam que a ordem que os nós filhos são gerados tem grande impacto diretamente no tempo de resposta ou na busca em si, e já o *BFS*, *A** e *hill climbing* não possui esta característica. O algoritmo A* apresentou o maior rendimento na busca, evidenciando um tempo menor de execução da busca. No caso testado, o algoritmo *hill climbing* não conseguiu achar a solução. Para melhorar a qualidade dos testes, o próximo passo seria a comparação da eficácia com novos caso de teste, e assim, comparar a velocidade de resposta do mesmo evidenciando os resultados obtidos.

## REFERENCES

[1] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, 2016.
[2] G. A. de Souza Viana. Fila e pilha com lista dinâmica. 2019.
[3] N. Meghanathan. Breadth first search. 2017.
[4] B. Selman and C. P. Gomes. Hill-climbing search. 2006.
[5] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.
[6] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD Conference*, 2005.
[7] Z. Zhang and Z. Zhao. A multiple mobile robots path planning algorithm based on a-star and dijkstra algorithm. 2014.