Resolução do problema 8-Puzzle Utilizando Algoritmos De Busca Cega e Busca Informada

Murillo Freitas Bouzon

***Resumo*—Um problema clássico de busca é o *8-Puzzle*, onde é necessário ordenar um quebra-cabeça movimentando apenas as peças adjacentes ao espaço vazio. É possível resolver este problema utilizando algoritmos de busca cega ou informada. Neste trabalho, foram implementados 2 algoritmos de busca cega e 2 algoritmos de busca informada utilizando estruturas de dados e uma hierarquia de classes para resolver o problema *8-Puzzle*. Os resultados mostraram que o método que conseguiu resolver em menor tempo foi o A*, resolvendo em 0.046 segundos.**

***Index Terms*—Algoritmos de Busca, Busca em largura, Busca em profundidade, A*, Subida de encosta, Jogo dos oito**

I. INTRODUÇÃO

Para resolver problemas onde é necessário chegar em um estado final a partir de um estado inicial definido, seguindo uma sequência de operações, foram propostos alguns algoritmos de busca. Esses algoritmos buscam pela solução visitando os estados no espaço de soluções, sendo feita em uma árvore de busca, onde cada estado é uma representação de uma solução no domínio do problema.

A navegação pelo espaço de busca é feita de acordo com as operações de transição possíveis de cada estado, podendo existir problemas onde a solução é inalcançável dependendo do estado inicial definido. Além disso, a ordem de expansão da árvore de busca e a velocidade em que a solução é encontrada difere para cada algoritmo de busca.

Uma abordagem utilizada para algoritmos de busca é a busca cega, onde os estados da árvore são expandidos de forma genérica, sem levar em consideração se o estado está próximo do objetivo do problema. Um exemplo de busca cega é a busca em largura que visita todos os dados de um nível L antes de visitar os estados do nível $L+1$. Outro exemplo é a busca em profundidade que segue um caminho específico para encontrar a solução e só altera o caminho se não for possível gerar novos estados a partir do estado em que se encontra, correndo o risco de entrar em *loopings* e nunca encontrar a solução final.

Outra abordagem de busca é a busca heurística, também chamada de busca informada, onde utiliza-se uma heurística, algum valor ou informação específica do problema que pode determinar a relevância de um estado e a prioridade em que novos estados são expandidos para chegar até o objetivo. Alguns exemplos desse tipo de busca são o A*, Subida de Encosta, Têmpera Simulada e Melhor-escolha.

Um problema de busca clássico é o problema das 8 peças, também conhecido como *8-Puzzle*. Sendo uma variação do jogo *15-Puzzle*, surgindo nos Estados Unidos no ano de aproximadamente 1880 [1]. Atualmente, pode ser resolvido facilmente com algoritmos de busca, cega quanto informada, sendo um problema simples para começar a aprender os princípios de algoritmos de busca.

Dito isso, este trabalho tem como objetivo resolver o problema *8-Puzzle* utilizando técnicas de IA, implementando os algoritmos de busca cega e informada utilizando estruturas de dados e comparando a performance entre eles.

II. FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados os conceitos relacionados aos algoritmos de busca, estruturas de dados e heurísticas utilizadas.

A. Heap

Heap é uma estrutura de dados, representada por uma árvore quase completa que representa a seguinte propriedade: Se u é um vértice pai de v, então o valor de u deve ser maior ou igual o valor de v (caso seja um *heap* de máximo), ou menor ou igual ao valor de v (caso seja um *heap* de mínimo). O vértice no topo do *heap* é chamado de raiz e é o elemento que tem a prioridade máxima ao ser removido. As operações de inserção e remoção têm complexidade de $\mathcal{O}\log(N)$, sendo N o número de elementos inseridos no *heap*. A Figura 1 apresenta um exemplo de *heap* de mínimo.

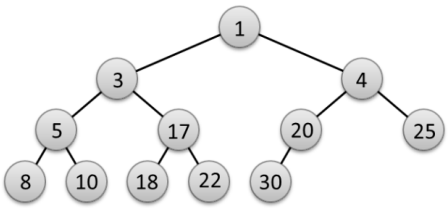

Figura 1: Heap de mínimo.

B. 8-Puzzle

O jogo consiste em um tabuleiro com 9 peças, sendo divididas entre 3 linhas e 3 colunas, onde 8 peças possui uma numeração de 1 a 8 e 1 única peça possui um espaço vazio. O objetivo é, a partir de uma configuração inicial, chegar até o estado final movimentando apenas as peças adjacentes ao espaço vazio. A Figura 2 mostra um exemplo de estado inicial e final.

C. Distância de Manhattan

A distância de Manhattan é uma métrica onde a distância entre dois pontos é a soma das diferenças absolutas de suas coordenadas. Pode-se dizer então que dado um estado U

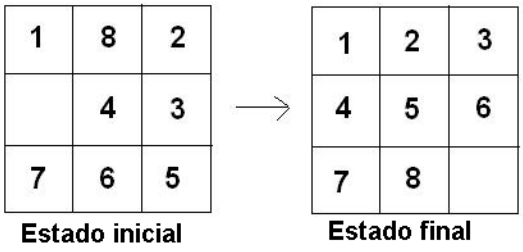

Figura 2: Exemplo de estado inicial e final do jogo 8-Puzzle.

do problema *8-puzzle* e o estado solução P, a distância de *Manhattan* entre eles é dada por:

$$dist = \sum_{i=1}^{N} |xu_i - xp_i| + |yu_i - yp_i|, \quad (1)$$

onde xu_i é a coluna em que a peça de valor i se encontra no estado U, xp_i é a coluna em que a peça de valor i se encontra no estado P, yu_i é a linha em que a peça de valor i se encontra no estado U e yp_i é a linha em que a peça de valor i se encontra no estado P.

D. Busca em Largura (BFS)

Breadth-first search (BFS) ou Busca em Largura, é um algoritmo de busca cega em grafo que funciona da seguinte forma: Dado um grafo $G = (V, E)$ e um vértice inicial S, o algoritmo enfileira S em uma fila que define a ordem em que os vértices serão visitados. Enquanto a fila não estiver vazia, desenfileira o primeiro elemento U e enfileira os vértices adjacentes V se V ainda não tiver sido visitado. Utiliza-se cores para marcar se o vértice já foi visitado. Os vértices brancos não foram visitados, os vértices cinzas serão visitados e os pretos já foram visitados [2]. A Figura 3 mostra um exemplo da aplicação do algoritmo BFS em uma árvore.

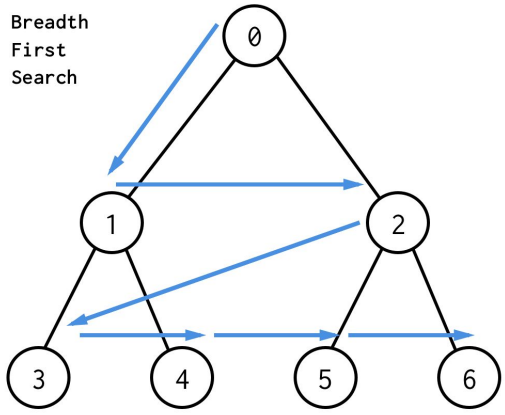

Figura 3: Ordem de exploração de uma busca feita pelo algoritmo BFS.

Ele retorna uma árvore como resposta onde está armazenado a distância de todos os vértices alcançáveis até S e os vértices predecessores de cada um. Possui complexidade de tempo e espaço $\mathcal{O}(b^d)$, sendo b o fator de ramificação e d o nível em que a solução está. No entanto, este método garante encontrar a solução ótima, com o menor número de passos para alcançá-la.

E. Busca em Profundidade (DFS)

Depth-first search (DFS) ou Busca em Profundidade, é um algoritmo de busca cega em grafo que recebe um vértice inicial S e procura a solução na parte mais profunda do espaço de soluções, visitando sempre o primeiro vértice encontrado de profundidade $N + 1$ até não ser possível gerar soluções mais profundas. Quando não é possível visitar um nó mais de profundidade maior, a busca faz um $backtracking$, ou seja, volta para explorar o resto dos vértices do espaço de solução que ainda não foram expandidos. Os vértices que serão visitados são inseridos em uma pilha, o que faz os últimos vértices a serem descobertos em uma lista de adjacência serem os primeiros a serem explorados. A Figura 4 mostra um exemplo da aplicação do DFS.

Assim como o BFS, retorna uma árvore como resposta onde está armazenado a distância de todos os vértices alcançáveis até S e os vértices predecessores de cada um. Possui complexidade de tempo e espaço $\mathcal{O}(b^d)$, sendo b o fator de ramificação e d o nível em que a solução está. Porém, não garante a solução ótima e corre o risco de entrar em $loopings$, visitando soluções que já foram encontradas de outras maneiras e repetindo muitos estados desnecessários.

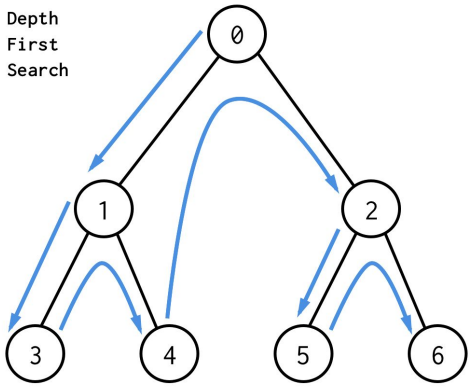

Figura 4: Ordem de exploração de uma busca feita pelo algoritmo DFS.

F. Subida de Encosta

O algoritmo Subida de Encosta, ou *Hill Climbing*, é um algoritmo de busca informada, onde leva em consideração um valor heurístico para explorar os vértice. O algoritmo percorre o espaço de soluções de forma decrescente, do vértice que possui o maior valor heurístico para o vértice que possui um valor menor. Quando encontra um vértice que não possui nenhum vértice adjacente com um valor heurístico menor, o algoritmo termina e retorna o vértice atual. Além disso, não

verifica se todos os vértices adjacentes possuem uma heurística menor, apenas seleciona o primeiro vértice que possuir um valor menor [3]. Esse método não garante que a solução será encontrada pois ela pode acabar caindo em máximos locais, onde não é possível mais encontrar vértices com um valor heurístico menor porém o vértice atual ainda não é a solução.

G. A*

O algoritmo A* (A-estrela) é um algoritmo de busca informada do tipo *best-first*, utilizado em grafos com vértices ponderados. Dado um vértice inicial S, ele busca chegar até o vértice objetivo G de forma que o custo total seja mínimo. Para isso, em cada iteração do algoritmo, ele expande o vértice que possui a menor função de custo, dada por:

$$f(n) = g(n) + h(n), \tag{2}$$

onde n é o vértice que será expandido, g(n) é a distância do vértice para o estado inicial e h(n) é o valor obtido pela heurística utilizada [3].

Uma das maneiras de implementar esse algoritmo é utilizando um *heap* de mínimo, onde cada vértice descoberto e inserido no *heap* e apenas o vértice raiz é expandido, sendo o vértice de menor custo a ser removido. O método garante encontrar a solução ótima dependendo da heurística utilizada.

III. METODOLOGIA

Neste trabalho foram implementadas 11 classes no total, sendo que as classes doublyList, Stack, Queue e Node já foram descritas em [4], recebendo pequenas modificações. A Figura 5 mostra o diagrama de classes UML das classes implementadas nesse trabalho.

Para representar cada estado do problema abordado, foi implementada a classe *Puzzle*, armazenando as principais características do quebra-cabeça, tendo como atributos:

- *pieces*: posições das peças do puzzle.
- *heuristic*: valor calculado pela função heurística utilizada.
- *dist*: distância do estado atual para o estado inicial.
- *totalCost*: custo total do estado.
- *emptyIndR*: linha do espaço vazio.
- *emptyIndC*: coluna do espaço vazio.
- *solvable*: indica se o puzzle é resolvível.

A classe *Node<Puzzle>* possui a classe *Puzzle* como composição e a classe *doublylist<Puzzle>* possui a classe *Node<Puzzle>* como composição e compõe a classe *puzzleSolver*, que é a classe base para as demais classes implementadas para resolver o problema. A classe *puzzleSolver* possui os atributos:

- *start*: estado inicial do puzzle.
- *result*: resultado final do puzzle.
- *seen*: lista de estados que já foram visitados.

Além disso, esta classe possui dois métodos abstratos, sendo eles o método *Solve()* e o método *movePieces()*.

A primeira classe derivada de *puzzleSolver* é a classe *solverBFS*. Ela resolve o problema *8-Puzzle* utilizando a abordagem de busca em largura (Seção II-D), inserindo os nós visitados em uma fila que compõe a classe. O Algoritmo 1 mostra um pseudo-código do método *Solve()* da classe solverBFS.

Algorithm 1: solverBFS.Solve()

```
if puzzle sem solução then
    return FALSE
end
Q.enqueue(start)
seen.insert(start)
while Q.Empty() do
    u = Q.Dequeue()
    if u == solução then
        result = u return TRUE
    end
    insereMovimentosPossíveis(u, Q)
end
return FALSE
```

A segunda classe derivada de *puzzleSolver* é a classe *solverDFS*. É utilizada a abordagem de busca em profundidade (Seção II-E) para resolver o *8-puzzle*, inserindo os nós visitados em uma pilha que compõe a classe. O Algoritmo 2 mostra um pseudo-código do método *Solve()* da classe solverDFS.

Algorithm 2: solverDFS.Solve()

```
if puzzle sem solução then
    return FALSE
end
S.Push(start)
seen.insert(start)
while S.Empty() do
    u = S.Pop()
    if u == solução then
        result = u return TRUE
    end
    insereMovimentosPossíveis(u, S)
end
return FALSE
```

A classe *solverHillClimb* herda os atributos e métodos da classe *puzzleSolver*. Ela resolve o quebra-cabeça utilizando a abordagem de busca informada *Hill Climb* (Seção II-F), não armazenando os nós que serão visitados em nenhuma estrutura de dados, apenas explorando o primeiro vértice que mostrar ter uma heurística melhor que do vértice atual, sendo utilizada a heurística distância de Manhattan (Seção II-C). O algoritmo 3 mostra um pseudo-código do método *Solve()* da classe *solverHillClimb*.

Para a classe *solverAStar*, foi necessário implementar uma classe *Heap* (Seção II-A) para manter uma fila de prioridade nos vértices visitados. O *Heap* implementado é um *Heap* de mínimo e possui como atributos:

- *root*: raiz do *heap*.
- *last*: último nó do *heap*.
- *size*: número de elementos inseridos.

O método *Heapify()* é mostrado no pseudo-código 4.

A última classe derivada de *puzzleSolver* é a classe *solverAStar* que utiliza a abordagem A* (Seção II-G) para

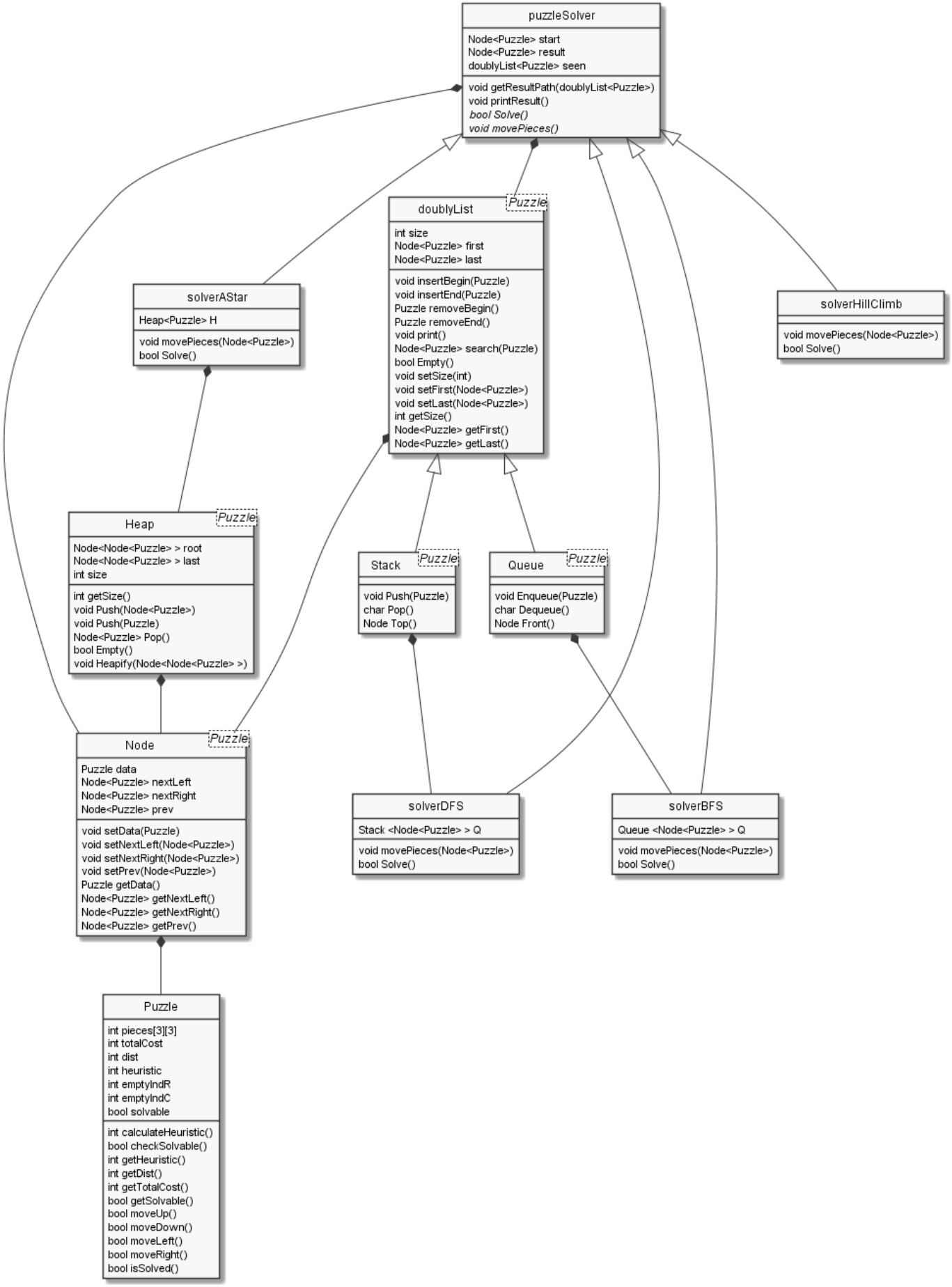

Figura 5: Diagrama UML das classes implementadas neste trabalho.

Algorithm 3: solverHillClimb.Solve()

```
if puzzle sem solução then
    return FALSE
end
u = start seen.insert(start)
while u != NULO do
    if u == solução then
        result = u return TRUE
    end
    u = moveVértice()
end
return FALSE
```

Algorithm 4: Heap.Heapify(atual)

```
if filhoEsquerdo ¡ filhoDireito then
    menor = filhoEsquerdo
else
    menor = filhoDireito
end
if menor ¡ atual then
    troca(menor, atual)
    Heapify(menor)
end
```

resolver o problema do *8-Puzzle*. Os vértices descobertos são inseridos em um *heap* binário de mínimo para manter os vértices de menor custo como prioridade para serem visitados. O custo é calculado somando heurística de distância de Manhattan (Seção II-C) com a distância do vértice atual para o vértice inicial. O Algoritmo 5 mostra o pseudo-código do método *Solve()* da classe *solverAStar*.

Algorithm 5: solverAStar.Solve()

```
if puzzle sem solução then
    return FALSE
end
H.Push(start)
seen.insert(start)
while H.Empty() do
    u = H.Pop()
    if u == solução then
        result = u return TRUE
    end
    insereMovimentosPossíveis(u, H)
end
return FALSE
```

IV. EXPERIMENTOS E RESULTADOS

Para avaliar os algoritmos implementados, foi feito um experimento para verificar se foram implementados corretamente e conseguem resolver o problema *8-Puzzle*. Além disso, foi medido o tempo que cada um dos algoritmos leva para resolver, o número de movimentos necessários para resolver e o número de vértices explorados. O estado inicial utilizado para todos os algoritmos é mostrado na Tabela I.

4	1	6
3	2	8
7	0	5

Tabela I: Caso de teste passado em aula.

A Figura 6 apresenta os resultados do método BFS recebendo o estado inicial definido em aula. Pode-se observar que ele conseguiu resolver o *Puzzle* em 1.045 segundos, visitando 3880 nós e realizando 13 movimentos no total para chegar até a solução. Os resultados obtidos pelo método DFS são mostrados na Figura 7, tendo encontrado a solução em 927.931 segundos, explorando 87126 vértices no total e tendo que realizar 47809 movimentos para chegar até a solução, não sendo a solução ótima para o problema. Esses resultados podem ser melhorados caso seja estabelecido um limite de profundidade a ser explorada pelo método, fazendo com que o algoritmo realize o *backtracking* antes, evitando caminhos muito longos que não levam ao resultado final.

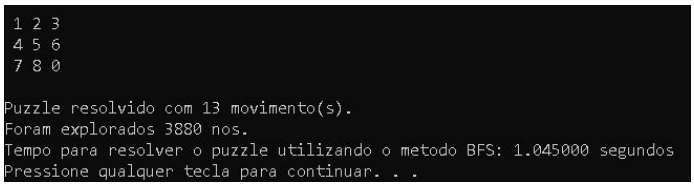

Figura 6: Resultado obtido pelo método BFS.

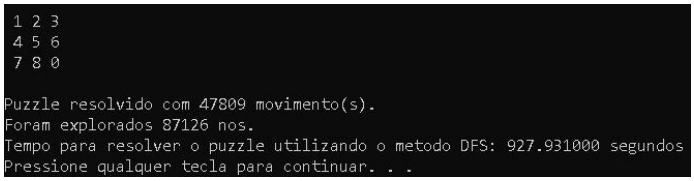

Figura 7: Resultado obtido pelo método DFS.

Não foi possível encontrar a solução utilizando o método Subida de Encosta, porque o método acabou encontrando um máximo local e não conseguiu descobrir mais nenhum nó que tivesse um valor heurístico menor que do nó atual. É possível contornar esse problema implementando outras variações do algoritmo Subida de Encosta, que evitam máximos locais. A Figura 8 mostra os resultados desse método.

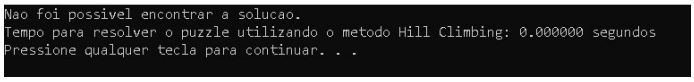

Figura 8: Resultado obtido pelo método *Hill Climbing*.

A Figura 9 mostra os resultados obtidos pelo algoritmo de busca A*. O método conseguiu resolver o *Puzzle* em 0.046 segundos, explorando apenas 37 nós e realizando apenas 13 movimentos para chegar no estado objetivo, obtendo a solução ótima assim como o método BFS.

A Tabela II mostra os resultados obtidos por todos os algoritmos implementados nesse trabalho. A linha destacada em verde mostra o método que obteve os melhores resultados.

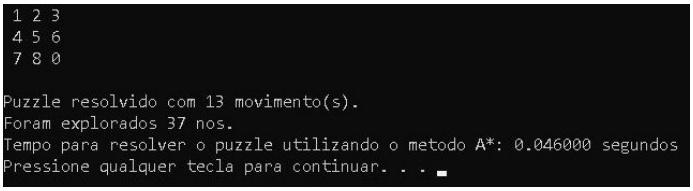

Figura 9: Resultado obtido pelo método A*.

Método	Resolvido	Tempo	Nós explorados	Movimentos
BFS	Sim	1.045	3880	13
DFS	Sim	927.931	87126	47809
Subida de Encosta	Não	-	-	-
A*	Sim	0.046	37	13

Tabela II: Tabela de resultados obtidos.

V. TRABALHOS RELACIONADOS

No trabalho de [5] foi desenvolvida uma nova abordagem chamada de *iBFS* capaz de executar i instâncias do algoritmo BFS, a partir de i vértices iniciais, de forma eficiente em GPUs. Desenvolveu-se um *kernel* de GPU para aproveitar melhor os nós de fronteira compartilhados entre cada instância e utilizou-se regras *Group by* para maximizar ainda mais o compartilhamento de fronteira dentro de cada grupo. Os resultados mostraram que o método executa 30 vezes mais rápido do que a execução do BFS sequencialmente.

Os autores [6] propuseram um modelo seguro de busca ranqueada de palavras-chaves sobre dados encriptados em nuvem, que ao mesmo tempo permite operações de remoção e inserção de documentos. Para isso, foi construído uma estrutura de dados indexada baseada em árvore e proposto um algoritmo chamado *Greedy Depth-first Search* para realizar a busca ranqueada de forma eficiente. O modelo proposto obteve complexidade de tempo sub-linear e conseguiu realizar inserções e remoções de forma flexível.

Em [7] foi proposto um algoritmo de busca local meta-heurístico, chamado de β-*Hill Climbing*, para resolver o problema do jogo Sudoku. O algoritmo é uma extensão do algoritmo Subida de Encosta com a adição de um novo operador chamado de operador-β que permite que a busca escape de máximos locais adicionando novas capacidades de exploração ao Subida de Encosta tradicional. Na avaliação, foi utilizado um caso de teste específico e o algoritmo conseguiu resolve-lo com 19 iterações e em 2 segundos.

VI. CONCLUSÃO

Neste trabalho, foram implementados 4 algoritmos de busca para resolver o problema do jogo *8-Puzzle*, sendo eles 2 algoritmos de busca cega e 2 algoritmos de busca informada, utilizando a heurística distância de Manhattan. Para a implementação, foi definida uma hierarquia de classes utilizando estrutura de dados para cada algoritmo.

Os algoritmos foram avaliados utilizando um estado final passado em aula. Foi medido o tempo em segundos para resolver o problema, o número de estados visitados e o número de movimentos feitos para chegar até a solução. Os resultados mostraram que o método mais rápido foi o A*, conseguindo resolver em 0.046 segundos e visitando apenas 37 nós para encontrar a solução ótima. O algoritmo Subida de Encosta não foi capaz de resolver o *puzzle* definido pois acabou caindo em um máximo local. Além disso, o algoritmo DFS levou muito tempo para encontrar a solução e não encontrou a solução ótima.

As possíveis melhorias para este trabalho é a implementação de variações do algoritmo Subida de Encosta para evitar máximos locais e a limitação de profundidade da exploração do método DFS. Além disso, podem ser feitos experimentos com casos mais difíceis e realizar uma avaliação profunda da performance dos algoritmos.

REFERÊNCIAS

[1] Aaron Archer. The 15 puzzle: how it drove the world crazy. *The Mathematical Intelligencer*, 29(2):83–85, Mar 2007.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, third edition. 2009.

[3] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[4] Murillo Freitas Bouzon. Implementação de filas e pilhas dinâmicas utilizando lista ligada simples e dupla. *PEL216 - Programação Científica*, pages 1–4, 2019.

[5] Hang Liu, H. Howie Huang, and Yaoping Hu. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD Conference*, 2016.

[6] M. Mohanrao. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. 2016.

[7] Mohammed Azmi Al-Betar, Mohammed A. Awadallah, Asaju La'aro Bolaji, and Basem O. Alijla. -hill climbing algorithm for sudoku game. *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, pages 84–88, 2017.