8-Puzzle

Eduardo Vieira e Sousa
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
eduardoatrcomp@gmail.com

***Abstract*—Este relatório aborda a aplicação de diferentes algoritmos de busca sobre o jogo conhecido como 8-Puzzle. Cada um desses métodos teve o desempenho medido, assim como sua complexidade em termos de tempo e espaço. Além disso, duas heurísticas foram utilizadas nas buscas com informação, sendo elas: número de peças fora do lugar e a soma das distâncias de cada peça à sua posição correta. Os resultados foram comparados e estão apresentados em tabelas, agrupadas pelo tipo de busca. Por fim, uma abordagem mais geral do problema, o N-Puzzle, é analisada utilizando o algoritmo A*.**

I. Introdução

O 8-Puzzle é um jogo que consiste em posicionar as 8 peças de um quebra-cabeças da maneira correta. Isso é feito movendo as peças para o espaço vazio, trocando-as de posição, até que o objetivo seja alcançado. Para este trabalho, o jogo é composto por peças enumeradas de 1 a 8, sendo que o objetivo é posicioná-las em ordem crescente, começando pelo canto superior esquerdo do tabuleiro.

Esse problema pode ser resolvido utilizando algoritmos de busca, que consistem em explorar um espaço de jogadas possíveis, até que seja encontrada uma solução. Para a implementação desses métodos foi utilizada a linguagem Java, e todo o processo, desde a modelagem do problema até a apresentação dos resultados e suas análises, está demonstrado nas seções seguintes.

II. Modelagem

Todo o problema foi modelado seguindo as sugestões apresentadas no livro-texto [1]. Além disso, as estruturas e rotinas apresentadas a seguir já estão adaptadas para o caso mais geral do jogo, o N-Puzzle.

A. Nodo

Os nodos são estruturas básicas que compõe a árvore de busca, carregando as informações sobre os estados do problema, para que uma função sucessora possa ser aplicada gerando novos nodos, com diferentes estados, até que um estado objetivo, também conhecido como *"Goal"*, seja encontrado. Na abordagem utilizada, a classe **Puzzle** representa o nodo, contendo os seguintes atributos:

- **tamanho**: um inteiro N que indica o tamanho dos lados do tabuleiro utilizado, sendo que no total, um tabuleiro possui $N^2 - 1$ peças;
- **tabuleiro**: representa o estado do problema através de uma matriz $N \times N$, onde o elemento e_{ij} representa a peça localizada linha i, e na coluna j do tabuleiro de entrada. Deste modo, a matriz contém uma sequência de números de 0 a $N^2 - 1$, em qualquer ordem, onde o espaço vazio é representado pelo valor 0;
- **pai**: um ponteiro para o nodo a partir do qual o estado atual foi gerado;
- **acao**: uma *String* que representa a ação tomada no pai do estado atual, para que o mesmo seja obtido. Essas ações são executadas sobre o ponto de vista do espaço vazio, e seus valores possíveis são: Cima, Direita, Esquerda e Baixo. A principal função dessa variável é a geração da saída do programa.
- **custo**: um inteiro C que representa o custo real para se alcançar o estado atual, partindo do estado inicial que gerou a árvore de busca. Esse valor é calculado através da soma do custo do nodo pai com o custo da ação que gerou o estado atual;
- **estimativa**: um inteiro E que representa o custo estimado para se alcançar o *Goal*, partindo do estado atual. Esse valor é calculado por meio de uma heurística, que varia dependendo do problema;
- **id**: um inteiro I que funciona como um identificador único, gerado para cada tabuleiro, permitindo uma rápida comparação e recuperação do estado dentro das estruturas de dados utilizadas nos algoritmos de busca.

B. Função Sucessora

Dado um estado do problema, a função sucessora é responsável por gerar os estados seguintes, localizados no próximo nível da árvore de busca. Esse processo é conhecido como expansão, e os nodos gerados também são chamados de filhos. A função é implementada no método **expandirNodo**, da seguinte maneira.

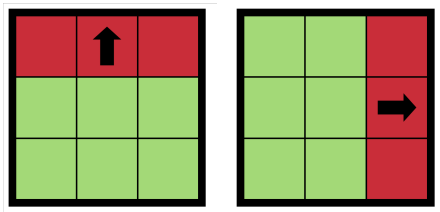
Para o N-Puzzle, as ações possíveis sobre um determinado estado consistem em "mover" o espaço vazio, representado pelo número 0, em todas as direções possíveis, dependendo de sua posição atual. Ao fazer isso, a peça na direção do movimento é trocada de lugar com ele, gerando um novo tabuleiro. Deste modo, a função sucessora deve encontrar o valor zero e verificar, por meio de sua posição no tabuleiro, se é possível movê-lo em cada uma das direções. Caso seja possível executar algum movimento, um nodo filho é gerado contendo um novo tabuleiro, permutando o lugar das duas peças envolvidas. Além disso, é necessário atribuir ao nodo filho a ação realizada e atualizar o seu custo, sendo que este

possui o custo do seu antecessor incrementado em 1 unidade, já que todos os movimentos possuem o mesmo custo.

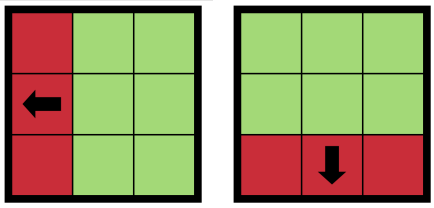
Como dito anteriormente, as restrições envolvidas nas possíveis jogadas vão variar de acordo com a posição do zero, sendo que os seguintes movimentos poderão ser executados quando ele se encontrar nas seguintes posições:

- Cima: quando não estiver na primeira linha;
- Direita: quando não estiver última coluna;
- Esquerda: quando não estiver na primeira coluna;
- Baixo: quando não estiver na última linha.

Na figura (1), a seta representa a direção do movimento, em verde estão representadas as posições onde ele é possível e em vermelho onde não é.



(a) Cima (b) Direita



(c) Esquerda (d) Baixo

Fig. 1. Restrições envolvidas em cada jogada

Todos os movimentos possíveis são testados pela função sucessora, os estados filhos são gerados e retornados em uma lista, para que possam ser explorados pelo algoritmo de busca que está sendo executado no momento.

A quantidade de filhos que cada nodo gera é conhecida como *branching factor*, e observando as restrições, é possível perceber que as 4 posições nos cantos do tabuleiro vão gerar apenas 2 filhos. Já as outras 4 posições intermediárias produzem 3 filhos, e a localizada no centro pode gerar 4, como mostra a figura (2). Deste modo, é possível fazer uma média ponderada, obtendo um *branching factor* médio, de acordo com a equação (1).

$$\frac{4 \cdot 2 + 4 \cdot 3 + 1 \cdot 4}{4 + 4 + 1} \cong 2,666 \quad (1)$$

No entanto, muitos dos estados gerados são repetidos, isto é, possuem tabuleiros idênticos aos de estados já explorados. Isso ocorre pois a cada jogada realizada, um movimento no sentido contrário pode ser executado, gerando novamente o estado anterior. Para impedir que a busca entre em *"loop"*, e também para evitar desperdício de tempo expandindo estados

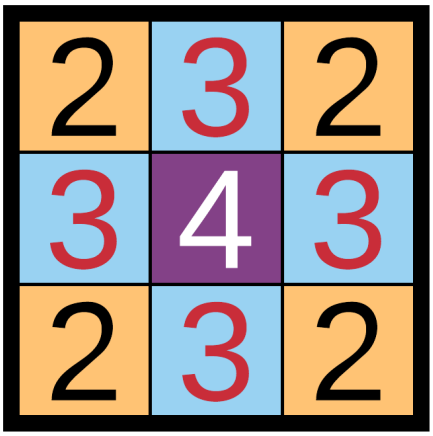


Fig. 2. Os números indicam a quantidade de filhos gerados a partir daquela posição

já explorados, é necessária a modelagem de duas estruturas de controle, que estão explicadas a seguir.

C. Fronteira

Representa o conjunto de nodos que já podem ser alcançados pela árvore de busca, mas que ainda não foram explorados. A estrutura de dados utilizada para representar a fronteira vai depender do algoritmo a ser implementado. Na abordagem proposta neste trabalho, a fronteira foi representada por 3 tipos de estruturas:

- Fila (FIFO): os elementos inseridos na fila saem em sua ordem de entrada;
- Pilha (LIFO): o último elemento a entrar é o primeiro a sair;
- Fila de prioridade (Heap): os elementos da fila de prioridade são inseridos de maneira ordenada, de acordo com o valor de algum atributo, do menor para o maior (MinHeap) e do maior para o menor (MaxHeap).

Além disso, como sugerido no livro-texto [1], o ideal é que essas estruturas sejam combinadas a uma tabela *hash*, para possibilitar uma recuperação rápida dos nodos desejados, ou para verificar a existência de nodos repetidos, reduzindo bastante o tempo de execução dos algoritmos.

Para que os nodos possam ser inseridos em uma tabela *hash*, é necessária uma chave que identifique cada estado de maneira única, levando em consideração apenas o tabuleiro, evitando nodos repetidos, e desconsiderando os outros atributos. Para isso, no momento da criação do nodo, um identificador é gerado baseando-se na ordem das peças do tabuleiro. Um **ID** de N^2 dígitos é calculado, onde o dígito mais à esquerda corresponde à primeira peça do tabuleiro, e o dígito mais à direita corresponde à última peça do tabuleiro, percorrendo o mesmo de cima para baixo, da esquerda para direita, como mostrado no exemplo da figura (3).

Desta maneira, é possível criar **ID**s únicos que identificam cada nodo por seu tabuleiro, podendo ser utilizados como

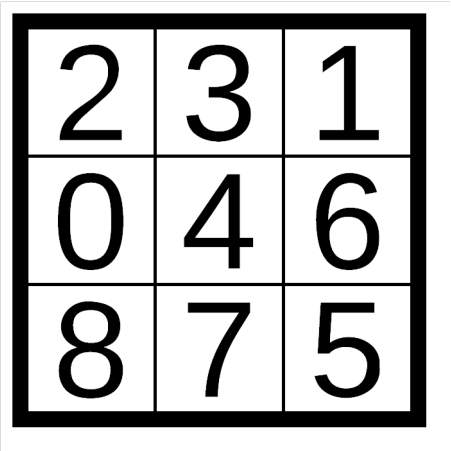


Fig. 3. O tabuleiro acima gera o ID: 231046875

chave na inserção destes nodos em um **HashMap**. Assim, é possível inserir apenas o identificador na estrutura de dados que representa a fronteira, utilizando ao invés dela o **HashMap** para realizar operações de busca e recuperação dos nodos, de maneira auxiliar, tornando a execução das buscas muito mais rápida e eficiente.

D. Explorados

São os nodos cujos filhos já foram gerados e inseridos na fronteira. Da mesma maneira que a fronteira, o livro-texto [1] recomenda a utilização de um *hash*, para tornar o processo de consulta mais eficiente. No entanto, neste caso não é necessário combinar a tabela *hash* com mais nenhuma outra estrutura, já que os nodos explorados devem apenas ser consultados, tornando possível a utilização de um **HashSet** apenas, utilizando como chave o **ID** gerado a partir do tabuleiro.

E. Solução

Os resultados gerados pelos algoritmos de buscas são retornados em um objeto do tipo **Solucao**, que consiste dos seguintes atributos:

- **resultado**: Um inteiro I que serve como flag para sinalizar o resultado da busca, sendo que os valores possíveis são: 1 para sucesso, 0 para *cutoff* e -1 para fracasso, que ocorre quando o algoritmo não encontra uma solução;
- **estadoInicial** e **estadoFinal**: Dois ponteiros que apontam para o estado inicial e o estado final do problema;
- **estados**: Uma lista contendo a sequência de estados da solução encontrada;
- **nodosGerados** e **nodosExplorados**: Dois inteiros que guardam a quantidade de nodos gerados e explorados para aquela solução.

F. Entrada e Saída

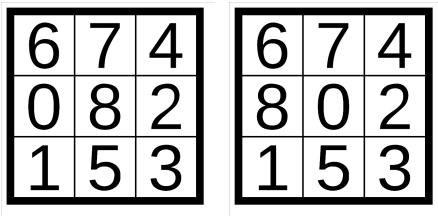
A entrada do programa é um arquivo de texto contendo em sua primeira linha um inteiro N e uma *String*. O valor N indica o tamanho do tabuleiro e a *String* indica o modo de execução do programa, que pode ser:

- **"SIMPLES"**: Indica uma execução simples do programa, que faz o cálculo apenas da quantidade de movimentos e do caminho para cada tabuleiro de entrada;
  "TESTE": Além de calcular o número de movimentos e o caminho, o modo de teste também gera outras estatísticas como o tempo de execução, o número de nodos gerados e explorados.

As próximas linhas do arquivo de entrada devem conter sequências de números que representam as peças do tabuleiro, separados por espaços, onde o primeiro número é a peça da primeira posição do canto superior esquerdo e o último número representa a última peça do canto inferior direito. Além disso, o espaço vazio deve ser representado pelo número zero, sendo que a sequência gerada para um tabuleiro é muito parecida com o **ID** demonstrado na figura (3), com a diferença que cada par de números é separado por um espaço. Cada tabuleiro de entrada deve estar em uma linha, separado dos demais, sendo que não existe limite para a quantidade de estados iniciais a serem calculados.

Todo o processo de leitura efetua a checagem da formatação do arquivo de entrada, testando os parâmetros da primeira linha e as matrizes. No entanto, é importante ficar atento à espaços vazios no fim dos parâmetros do arquivo ou entre as linhas, pois estes não são detectados visualmente mas impedem a leitura correta do arquivo.

Já saída do programa é um arquivo onde a *String* "Entrada(N)" inicia um bloco para as soluções da linha N do arquivo de entrada. O bloco de soluções dividido em sub-blocos, uma para cada algoritmo, onde em sua primeira linha está indicado o algoritmo executado, seguido do número de movimentos. Na linha seguinte é indicada a sequência de movimentos a serem executados para obter a solução encontrada pelo método, sendo que os valores possíveis são: **Cima**, **Direita**, **Esquerda** e **Baixo**. Essas ações devem ser realizadas do ponto de vista do espaço vazio, trocando-o de posição com a peça na direção indicada, como mostra a figura (4).



(a) Antes (b) Depois

Fig. 4. Movimento **Direita** executado sobre o tabuleiro

Além disso, se o programa for executado em modo de teste, eles irá gerar 3 linhas extras contendo o tempo, o número de estados gerados e explorados para cada algoritmo, sendo executa de uma vez todos eles sobre cada um dos tabuleiros fornecidos no arquivo de entrada. O Greedy Best-First Search é executado com a heurística do número de peças fora do

lugar, já o A* e o Hill Climbing são executados utilizando a soma das Distâncias de Manhattan para cada peça.

III. Algoritmos

Os algoritmos implementados nesse trabalho podem ser divididos em 3 tipos:

- Busca sem informação (*Blind Search*): usa somente as informações sobre o estado fornecidas pela definição do problema. Os algoritmos implementados foram o *Breadth-first Search*, *Iterative Deepening Search* e *Uniform-cost Search*;
- Busca com informação: utiliza informações específicas para o problema que estão além das fornecidas pela definição. Os algoritmos implementados foram o *Greedy Best-first Search* e o *A* Search*;
- Busca Local: guarda apenas o estado corrente, movendo-se através dos vizinhos buscando maximizar ou minimizar uma função objetivo. O algoritmo implementado foi o *Hill Climbing*, permitindo movimentos laterais.

Além disso, esses algoritmos podem ser analisados de acordo com os seguintes critérios:

- Completude: o algoritmo sempre acha a solução, caso ela exista;
- Otimalidade: o algoritmo encontra a melhor solução possível;
- Complexidade de Tempo: o tempo gasto para encontrar a solução;
- Complexidade de Espaço: a memória gasta para executar a busca.

A seguir, está explicada de maneira sucinta o funcionamento e as diferenças entre os algoritmos abordados.

A. Breadth-first Search (BFS)

Consiste em expandir os nodos na ordem em que são gerados, ou seja, *first-in first-out* (FIFO). Deste modo, todos os nodos de um certo nível são expandidos antes que os do próximo sejam visitados. Para a implementação deste algoritmo, a fronteira foi representada por uma fila, implementada pela classe **LinkedList** e combinada a um **HashMap**.

No caso do BFS, o teste para verificar se o estado final foi atingido pode ser aplicado na geração do nodo, ao invés de quando ele é selecionado para expansão. Isso pode ser feito pois o nodo que está sendo gerado é a versão do estado mais rasa na árvore, e como os custos de cada passo são iguais, isso garante que se tal nodo possui um estado final, ele será solução ótima. Essa otimização é conhecida como *Early Check Optimization*, e foi implementada para o BFS neste trabalho.

B. Iterative Deepening Search (ITS)

A busca em aprofundamento iterativo consiste na aplicação da busca em profundidade limitada (DLS), incrementando o limite de profundidade após cada busca. Por sua vez, a busca de profundidade limitada consiste na aplicação da busca em profundidade (DFS), limitando a profundidade máxima de busca, o que na prática faz com que o DFS explore todos os nodos até o nível limite.

A vantagem dessa estratégia está em um menor gasto de memória para se obter resultados que estejam localizados em profundidades favoráveis na árvore de busca. A fronteira do DLS foi construída utilizando uma pilha (LIFO), implementada por meio da classe **Stack**, deste modo explorando sempre o último nodo adicionado, e tratando nodos no limite de profundidade como se estes fossem nodos folha.

Já o IDS em si possui apenas um objeto do tipo DLS para que a busca limitada possa ser realizada, passando por parâmetro o **Puzzle** e um inteiro l que representa o limite de profundidade. Desse modo o limite é incrementado em um laço enquanto a solução não for encontrada.

C. Uniform-cost Search (UCS)

A busca de custo uniforme é muito parecida com o BFS, com a diferença de que ao invés de selecionar os nodos da fronteira utilizando uma fila comum (FIFO), ela utiliza uma fila de prioridade, ordenada pelo custo do nodo, do menor para o maior (MinHeap). Deste modo, o algoritmo será ótimo mesmo para passos com diferentes custos, o que não acontece no BFS. Para esse algoritmo, a fronteira foi implementada utilizando uma **PriorityQueue** combinada a um **HashMap**, para melhorar o desempenho na consulta e recuperação dos nodos.

D. Greedy Best-First Search (GBFS)

A busca gulosa expande sempre o nodo mais próximo do *Goal*. Essa distância é calculada e inserida na variável **estimativa**, que representa a função de avaliação. O cálculo é feito utilizando uma função heurística, que varia de acordo com o problema. A implementação permite escolher entre 2 funções heurísticas diferentes: número de peças fora do lugar e a soma das distâncias de cada peça à sua posição correta, utilizando a Distância de Manhattan. Sua fronteira foi implementada utilizando uma **PriorityQueue**, combinada a um **HashMap** e ordenada de acordo com a variável **estimativa**.

E. A* Search (AS)

Funciona de maneira quase idêntica à busca de custo uniforme, com a diferença de que a função de avaliação é calculada somando-se o custo real com o custo estimado, que calculado através de uma heurística e inserido na variável **estimativa**. Deste modo, as estruturas utilizadas são as mesmas, e assim como no GBFS, duas heurísticas estão disponíveis.

F. Hill Climbing (HC)

Esse algoritmo simplesmente se move para os estados vizinhos escolhendo sempre o que maximiza ou minimiza sua função objetivo, de acordo com o problema. Quando não é mais possível otimizar a solução, o algoritmo para. Na implementação deste trabalho, o algoritmo sofreu uma modificação para que ele ainda execute uma certa quantidade de movimentos quando se encontrar preso em uma área de *shoulder*, dentro da paisagem de estados. Esse limite é passado por parâmetro para a função de busca, juntamente com a função objetivo escolhida entre as duas opções disponíveis.

Por não guardar informações sobre os estados já percorridos, esse algoritmo não precisa de muitas estruturas de controles, fazendo uso de apenas alguns ponteiros.

IV. Heurísticas

A performance dos algoritmos de busca que utilizam heurísticas vai depender da qualidade das mesmas. Para que esses algoritmos sejam ótimos, uma das condições necessárias é que a heurística utilizada seja admissível. Isso significa que a heurística nunca vai superestimar o custo para se atingir o estado final a partir de um estado dado. Ou seja, o custo estimado pela heurística é sempre menor ou igual ao custo real para se atingir o *Goal* a partir daquele estado. Deste modo, ambas as heurísticas implementadas são admissíveis.

- Número de peças fora do lugar: é admissível pois cada peça deverá ser movimentada pelo menos uma vez;
- Distância de Manhattan: é admissível pois cada ação movimenta cada peça apenas um passo mais próximo do *Goal*.

V. Testes

Nesta seção estão demonstradas as saídas do programa executado em modo **SIMPLES** para o último caso de testes, que possui profundidade 31.

Entrada(1)

BFS: 31 movimentos
Solução: Cima Cima Direita Baixo Esquerda Esquerda Baixo Direita Direita Cima Esquerda Esquerda Cima Direita Direita Baixo Esquerda Baixo Esquerda Cima Cima Direita Baixo Baixo Esquerda Cima Cima Direita Direita Baixo Baixo

UCS: 31 movimentos
Solução: Direita Cima Cima Esquerda Baixo Esquerda Baixo Direita Direita Cima Cima Esquerda Esquerda Baixo Direita Cima Esquerda Baixo Baixo Direita Direita Cima Cima Esquerda Esquerda Baixo Baixo Direita Cima Direita Baixo

IDS: 31 movimentos
Solução: Esquerda Cima Direita Baixo Direita Cima Esquerda Cima Direita Baixo Esquerda Esquerda Baixo Direita Direita Cima Esquerda Cima Esquerda Baixo Baixo Direita Cima Cima Esquerda Baixo Baixo Direita Cima Direita Baixo

GBFS(h1): 73 movimentos
Solução: Direita Cima Cima Esquerda Esquerda Baixo Baixo Direita Direita Cima Cima Esquerda Baixo Baixo Esquerda Cima Direita Cima Esquerda Baixo Baixo Direita Cima Esquerda Cima Direita Baixo Baixo Direita Cima Cima Esquerda Esquerda Baixo Baixo Direita Cima Direita Baixo Esquerda Cima Esquerda Baixo Direita Cima Esquerda Cima Direita Baixo Baixo Esquerda Cima Direita Cima Esquerda Baixo Baixo Direita Direita Cima Esquerda Baixo Esquerda Cima Direita Direita Baixo Esquerda Cima Esquerda Baixo Direita Direita

AS(h2): 31 movimentos
Solução: Cima Esquerda Baixo Direita Direita Cima Cima Esquerda Baixo Esquerda Cima Direita Baixo Esquerda Baixo Direita Direita Cima Esquerda Esquerda Baixo Direita Direita Cima Cima Esquerda Esquerda Baixo Baixo Direita Direita

HC(h2): 1 movimentos
Solução: Direita

VI. Resultados e Análises

Os dados expostos a seguir foram gerados a partir dos exemplos fornecidos no arquivo "npuzzle.pdf", que contém 32 estados iniciais ordenados de 0 a 31, correspondendo ao número de movimentos da solução ótima. Todos os métodos de busca foram executados sobre todas as entradas fornecidas, gerando resultados para os 32 níveis de profundidade, que foram utilizados para uma análise quantitativa do comportamento de cada algoritmo.

Antes de começar a análise, é importante ressaltar que a utilização de tabelas *hash* para consulta e recuperação dos nodos reduz também drasticamente o tempo dos algoritmos, que no caso é o tempo de execução e não o tempo medido em quantidade de nodos gerados, do qual o livro-texto [1] se utiliza para fazer uma análise da complexidade de cada método de busca.

Deste modo, os resultados estão apresentados em tabelas, que por sua vez estão divididas por tipo de algoritmo. O tempo de execução foi calculado em milissegundos, e a profundidade da solução ótima está representada em cada tabela pela letra **d**. Além disso, para uma melhor exibição, estão apresentados apenas os resultados das profundidades de 4 em 4, o que não prejudica a noção do crescimento da função para cada algoritmo.

A. Busca com Informação

A tabela (I) apresenta os resultados obtidos para os algoritmos de busca sem informação: Breadth-first Search, Uniform-cost Search e Iterative Deepening Search. Os dados medidos foram o tempo de execução, o número de nodos gerados e o número de nodos explorados, sendo que esses valores estão em função da profundidade da solução **d**.

Com um custo constante de 1 para cada passo, o BFS e o UCS possuem um comportamento idêntico, porém o BFS obteve uma pequena vantagem sobre o UCS em todos os casos, como pode ser constatado nas contagens de nodos gerados e explorados. Essa diferença é oriunda de uma pequena otimização feita no BFS, que faz a checagem de *Goal* em cada nodo no momento em que ele é gerado, ao invés de fazê-lo em sua expansão. Essa otimização também é conhecida como *Early Check Optimization*, e foi implementada apenas no BFS, para que seus efeitos pudessem ser mensurados utilizando os dados do UCS como comparativo. Já o tempo do ICS foi consideravelmente maior que o de ambos, crescendo

TABLE I
COMPARAÇÃO DO TEMPO DE EXECUÇÃO (MILISSEGUNDOS) EM FUNÇÃO DA PROFUNDIDADE DA SOLUÇÃO

	Tempo (ms)		
d	BFS	UCS	IDS
1	0	0	0
4	0	0	1
8	1	1	2
12	3	3	7
16	14	21	42
20	32	56	212
24	206	247	1163
28	262	285	4874
31	203	228	11704

substancialmente a cada nível, já que os níveis anteriores precisam ser reexplorados novamente.

TABLE II
COMPARAÇÃO DO NÚMERO DE NODOS GERADOS EM FUNÇÃO DA PROFUNDIDADE DA SOLUÇÃO

	Gerados		
d	BFS	UCS	IDS
1	3	9	3
4	35	100	108
8	406	660	618
12	2611	4002	6831
16	18286	28518	86639
20	73231	105020	532424
24	292764	371360	2840756
28	457292	480874	11654933
31	483677	483838	28217164

Como é possível constatar nas tabelas (II) e (III), o BFS manteve uma pequena vantagem tanto no número de nodos gerados como nos explorados devido à otimização implementada, o que justifica o menor tempo de execução apresentado em comparação ao UCS. Além disso, é interessante notar que ambos exploraram quase todo o espaço de estados do 8-Puzzle, que é de 181440 estados. Já o IDS apresentou valores muito maiores em ambos os casos, já que esses valores são somados a cada profundidade buscada pelo algoritmo do algoritmo.

TABLE III
COMPARAÇÃO DO NÚMERO DE NODOS EXPLORADOS EM FUNÇÃO DA PROFUNDIDADE DA SOLUÇÃO

	Explorados		
d	BFS	UCS	IDS
1	1	3	1
4	14	35	40
8	144	248	226
12	949	1486	2260
16	6783	10450	23669
20	27466	38110	112805
24	108540	139606	423546
28	171567	180366	1049817
31	181385	181439	1650734

B. Busca com Informação

As tabelas a seguir apresentam os resultados obtidos para os algoritmos de busca com informação: Greedy Best-First Search e A* Search. Ambos os métodos foram testados utilizando duas heurísticas diferentes: número de peças fora do lugar **(h1)** e soma das Distâncias de Manhattan de cada peça à sua posição correta **(h2)**. Isso foi feito para que não apenas os métodos pudessem ser comparados, mas também as heurísticas.

TABLE IV
COMPARAÇÃO ENTRE AS DUAS HEURÍSTICAS EM TERMOS DO NÚMERO DE NODOS GERADOS, EXPLORADOS E MOVIMENTOS REALIZADOS

	GBFS					
	Movimentos		**Gerados**		**Explorados**	
d	h1	h2	h1	h2	h1	h2
1	1	1	3	3	1	1
4	4	4	12	12	4	4
8	8	8	22	22	8	8
12	30	40	1499	1191	549	429
16	30	46	1363	1202	502	436
20	46	50	1944	1107	719	402
24	46	54	3680	1173	1367	425
28	56	28	2827	119	1042	43
31	73	47	904	384	334	145

Ambas as heurísticas levaram à solução ótima até a profundidade 9, e a partir daí a Distância de Manhattan se mostrou um pouco mais eficiente nos níveis mais profundos. Essa vantagem também se manteve no número de nodos gerados e explorados, como é possível ver na tabela (IV). Com a utilização das tabelas *hash* para as estruturas de controle, o tempo de execução do GBFS foi constante para todos os níveis.

TABLE V
COMPARAÇÃO ENTRE AS DUAS HEURÍSTICAS EM TERMOS DO NÚMERO DE NODOS GERADOS, EXPLORADOS E TEMPO DE EXECUÇÃO

	A*					
	Tempo (ms)		**Gerados**		**Explorados**	
d	h1	h2	h1	h2	h1	h2
1	0	0	3	3	1	1
4	0	0	12	12	4	4
8	0	0	22	22	8	8
12	1	1	168	41	60	15
16	2	1	1665	240	614	88
20	8	1	7611	480	2805	179
24	25	3	45532	2237	16824	842
28	113	16	190925	15521	71009	5864
31	206	32	385076	55925	143848	21197

Como pode ser visto na tabela (V), a segunda heurística leva vantagem em todos os aspectos no A*, fazendo com que o algoritmo encontre a solução ótima em uma pequena quantidade de tempo e espaço, quando comparado aos outros tipos de busca.

C. Busca Local

As tabelas a seguir apresentam os resultados obtidos para o algoritmos de busca local Hill Climbing, que foi testado utilizando duas heurísticas diferentes como função objetivo: número de peças fora do lugar **(h1)** e soma das Distâncias de Manhattan de cada peça à sua posição correta **(h2)**. Além disso, foi definido um limite de 50 movimentos laterais para cada busca, sendo que a função objetivo deve ser minimizada.

TABLE VI
COMPARAÇÃO ENTRE AS DUAS HEURÍSTICAS EM TERMOS DE MOVIMENTOS E NÚMERO DE NODOS GERADOS

	HC			
	Movimentos		**Gerados**	
d	h1	h2	h1	h2
1	1	1	5	5
4	4	4	14	14
8	8	8	24	24
12	20	12	62	36
16	54	1	137	7
20	56	3	144	10
24	42	1	109	5
28	54	0	141	2
31	51	1	128	5

Os valores acima do limite de movimentos indicam que o algoritmo ficou preso em um *shoulder*. Já os resultados menores do que a profundidade da solução ótima indicam que o HC encontrou um mínimo local. Deste modo, é possível verificar claramente a superioridade da Distância de Manhattan para esse problema, como mostram os resultados das tabelas (VI) e (VII).

TABLE VII
COMPARAÇÃO ENTRE AS DUAS HEURÍSTICAS EM TERMOS DAS SOLUÇÕES ENCONTRADAS

Soluções	**h1**	**h2**
Ótimas	7	12
Mínimos Locais	5	20
Shoulders	20	0

A segunda heurística fez com quem o HC sempre convergisse, ao contrário da primeira que ficou presa em shoulders a maioria das vezes. Além disso, a quantidade de soluções ótimas proporcionadas por **h2** foi consideravelmente maior.

VII. CONSIDERAÇÕES FINAIS

Alguns detalhes na implementação podem alterar as quantidades de nodos gerados e explorados, como por exemplo a ordem na qual os movimentos são testados e inseridos na fronteira. Essas diferenças ficam ainda mais perceptíveis no GBFS por exemplo, e podem definir qual nodo será explorado quando os primeiros da lista de prioridade tiverem estimativas iguais. Isso pode acarretar na geração de um nodo com uma estimativa menor e a exploração de um ramo completamente diferente da árvore, quando comparado com outras implementações, o que evidencia a necessidade de heurísticas eficientes que ajudem na distinção dos estados e ações da melhor maneira possível.

VIII. EXTRA: N-PUZZLE

A versão genérica do problema, o N-Puzzle, é um desafio extremamente difícil. Para se ter noção disso, basta calcular o número de configurações acessíveis para cada versão do jogo, isso pode ser feito de acordo com a equação (2).

$$\frac{(n+1)!}{2} \tag{2}$$

O número de configurações acessíveis pode ser obtido dividindo o número de configurações possíveis por 2, isto é, as configurações que podem ser obtidas realizando todos os movimentos sobre um estado final ordenado, até que seja mais possível gerar um novo estado. Isso significa que apenas metade das configurações possíveis possuem solução, porém isso não ajuda muito no problema, pois a função fatorial cresce tão rapidamente, que se torna difícil até calcular esse número, como demonstrado a seguir:

- 3-Puzzle: 12 configurações
- 8-Puzzle: 181.440 configurações
- 15-Puzzle: $653.837.184.000$ configurações
- 24-Puzzle: aproximadamente $7 \cdot 76 \times 10^{24}$ configurações

Além disso, a única maneira de se encontrar a solução ótima é fazendo a busca exaustiva sobre o espaço de estados, sendo que o 3-Puzzle possui a sua profundidade máxima em 6, já a solução mais profunda do 8-Puzzle possui 31 movimentos e por fim, o 15-Puzzle, que possui 17 soluções ótimas com a profundidade máxima de 80 jogadas [2].

REFERENCES

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

[2] E. Korf, Richard. (2008). Linear-Time Disk-Based Implicit Graph Search. J. ACM. 55. 10.1145/1455248.1455250.