1. Introdução

O Trabalho Prático 1 da disciplina Inteligência Artificial em 2019/1 consistiu em implementar os seguintes algoritmos de busca para resolver o *N-Puzzle*:

- *Breadth-first Search* (BFS)
- *Iterative deepening search* (IDS)
- *Uniform-cost search* (UCS)
- *A* search* (A*)
- *Greedy best-first search* (GBFS)
- *Hill Climbing*

2. Definição do problema

O *8-Puzzle* é um conhecido jogo que consiste em um tabuleiro de 3x3 espaços, sendo que cada um desses espaços, exceto um, possui uma peça que pode ser numerada de 1 a 8 e nenhuma numeração pode se repetir. O objetivo do jogo é, dada uma certa configuração inicial das peças, chegar no estado final representado pela tabela:

1	2	3
4	5	6
7	8	

Para chegar nesse estado só é permitido arrastar as peças pelo tabuleiro, de maneira que uma jogada consisti em trocar o espaço vazio com uma de suas peças adjacentes.
Neste trabalho, o problema foi generalizado para o *N-Puzzle*, que segue o princípio do *8-Puzzle*, porém para um tabuleiro com NxN posições.

3. Modelagem do problema

Para representar o *8-Puzzle* foi usado um vetor de 9 posições para representar o tabuleiro do jogo. O estado final do jogo:

1	2	3
4	5	6
7	8	

É representado pelo vetor :

1	2	3	4	5	6	7	8	0

Sendo que o espaço em branco no tabuleiro é representado pela número 0 no vetor. Para o *N-Puzzle*, o vetor possui NxN posições. Considerando o tabuleiro como uma matriz, para encontrar a coluna e a linha correspondentes à uma posição i do vetor, $0 \leq i \leq 8$, basta fazer:
Coluna = i / N
Linha = i % N

4. Implementação

Os algoritmos foram implementados na linguagem C++, utilizando uma classe chamada *Puzzle* para representar uma instância do jogo. A classe está definida no arquivo *puzzle.hpp* e sua implementação está no *puzzle.cpp*.
A classe *Puzzle* contém a configuração do tabuleiro armazenada em um vetor de NxN posições, o índice do espaço em branco no vetor, o índice anterior do espaço em branco, um ponteiro para o pai desse nó, um vetor de ponteiros para os filhos desse nó e o número de jogadas necessárias para chegar do estado inicial até este nó.
Como o uso da palavra nó sugere, essa classe representa um nó de uma árvore, na qual os filhos são todos os tabuleiros gerados por uma jogada no tabuleiro do nó pai.
A classe possui vários métodos, entre os quais estão as heurísticas usadas por alguns algoritmos, a geração dos nós filhos e a verificação de se o nó é o estado final.
Os algoritmos foram implementados separadamente, cada um em um arquivo cpp. Para os algoritmo que utilizam uma fila de prioridades, foi criada a classe *Heap* que implementa um *min-heap* de *Puzzle*.

5. Visão Geral dos algoritmos

5.1. **Modificações para melhorar a eficiência.** Para ser mais eficiente, foram usados *unordered_set* da biblioteca padrão do C++ para implementar esses conjuntos. Essas estruturas de dados funcionam como *hash-tables* sendo possível verificar em $O(1)$ se um elemento está ou não na estrutura. Outra modificação foi criar uma função que transforma uma configuração do tabuleiro em um número inteiro único e sem colisões com outras configurações. Assim, os *unordered_set* continham apenas inteiros, melhorando a eficiência.

5.2. ***Breadth-first Search* (BFS).** O BFS faz uma busca em largura pela árvore de estados. Partindo do estado inicial, os estados (nós) são expandidos, sendo colocados no conjunto de expandidos, e os seus filhos são criados e colocados na fronteira. Caso um filho gerado seja o estado final, então o algoritmo para e acha a solução ótima. A solução é ótima, pois o BFS verifica cada nível da árvore em ordem e em cada nível todos os nós possuem o mesmo custo, que é estritamente crescente à medida em que se avança nos níveis.
O BFS é exponencial no número de passos para a solução ótima. Sendo d o número de passos ótimo e b o *branch factor* da árvore de estados, então a complexidade temporal e espacial do BFS é $O(b^d)$. Entretanto, é característica do *N-Puzzle* gerar muitos estados repetidos, assim pode-se podar todos os estados repetidos e melhorar significativamente a eficiência do algoritmo. A poda é feita usando os conjuntos explorados e fronteira.

5.3. ***Iterative deepening search* (IDS).** O IDS faz repetidas buscas em profundidade na árvore de estados, sendo que à cada iteração o nível de descida na árvore é incrementado por 1. O algoritmo é completo e ótimo, porém não faz a poda de estados repetidos, pois o custo de se chegar em um estado pode ser diferente de acordo com as jogadas utilizadas. No BFS a poda é possível, pois ele navega por todos os estados de um nível para depois ir para o nível posterior, assim os estados repetidos terão custo maior do que o estado já explorado, no IDS isso não ocorre.

5.4. ***Uniform-cost search* (UCS).** O UCS utiliza uma fila de prioridades para explorar a árvore de estados de maneira mais eficiente. Os estados com menor custo são explorados primeiro e caso encontre o mesmo estado com um custo menor, então o custo dele é alterado na fila de prioridades. Entretanto, no *N-Puzzle* o UCS se degenera para um BFS com maior custo para guardar os estados da fronteira. Isso ocorre, pois cada jogada sempre aumenta o custo em uma unidade, assim o UCS explora cada nível da árvore inteiramente para só depois explorar o próximo nível, assim como no BFS.

5.5. ***A* search* (A*).** O A* foi implementado da maneira que o UCS, exceto pela função de calcular o custo de cada estado. No UCS, o custo é o número de jogadas para chegar nesse estado, já no A* o custo é o o número de jogadas para chegar nesse estado somado à uma previsão otimista do número de jogadas para se chegar ao estado final. À essa previsão é dado o nome de função heurística. A heurística usada no A* foi a distância *Manhattan* que é a soma do número de movimentos necessários para mover uma peça até seu lugar correto, ignorando as outras peças e não podendo mover na diagonal, para todas as peças. Uma maneira de ver a distância *Manhattan* para uma peça é considerar que ela está sozinha no tabuleiro (não tem outras peças, só espaços em branco) e a distância é a solução ótima para essa configuração respeitando as regras de movimento. Essa heurística é admissível.
Prova: Sendo h(s) a distância *Manhattan* para um estado s, s* o estado final, s0 o estado inicial e C* o número de passos da solução ótima para s0, sabe-se que h(s*) = 0. Assume-se que h(s0) ¿ C*. Nota-se que cada ação só pode mover uma peça, assim ao fazer uma ação apenas uma peça será modificada e consequentemente só a distância *Manhattan* dessa peça será reduzida em no máximo um movimento, assim h só será reduzida em no máximo 1. Dessa maneira, como pode-se chegar em s* com C* movimentos então : h(s*) $\geq$ h(s0) - C* $>$ 0, o que é uma contradição pois h(s*) = 0. Portanto, h(s0) $\leq$ C*. Fonte: *University of California, Irvine*

5.6. ***Greedy best-first search* (GBFS).** O GBFS também usa a mesma implementação do UCS, mas utiliza como função de custo apenas a heurística. Para ele foi usada a heurística que conta o número de peças fora do lugar. É fácil ver que essa heurística é admissível, já que só se pode mover uma peça por movimento e o número de peças fora do lugar significaria mover no mínimo uma peça por vez, sem prejudicar as outras peças.
O GBFS não é um algoritmo ótimo e nem é completo.

5.7. ***Hill Climbing*.** O *Hill Climbing* é um algoritmo de busca local, para este trabalho foi escolhida a versão mais simples *Steepest-ascent* modificada para permitir movimentos laterais e guardar o caminho até a solução.
A função maximizada foi o número de peças no lugar, dessa maneira o algoritmo sempre explora os estados com um número de peças maior ou igual (até certo limite) do que o estado atual. Entretanto, isso se mostrou extremamente inútil, visto que o algoritmo nunca volta atrás e acaba caindo em um máximo local que não é a solução com frequência. Assim, essa modelagem se mostrou péssima para o *Hill Climbing*. O algoritmo foi testado usando a distância *Manhattan* como custo, mas continuou caindo em mínimos locais.

6. Experimentos e resultados

Os algoritmos foram executados 20 vezes para cada caso teste. O código foi alterado para medir o tempo de cada chamada da função que implementa o algoritmo, assim desalocação de memória, entrada e saída não foram levados em consideração. O tempo foi medido em microssegundos. Os gráficos abaixo mostram os resultados obtidos com um intervalo de confiança de 95%.
Em relação ao tempo de execução percebe-se que o *Hill Climbing* foi o mais rápido, seguido pelo GBFS. Nas figuras [4] [3] pode se ver a enorme diferença entre os dois mais rápidos e o resto. O *Hill Climbing* possui um custo quase constante, enquanto o GBFS depende da sorte de ter escolhido um bom caminho para achar mais rapidamente e mostra que o tempo tende a aumentar com o número de passos ótimo. O terceiro mais rápido foi o A*, seguido pelo BFS como era esperado. O inesperado ocorreu em relação ao UCS, já que ele deveria se degenerar para o BFS, mas acabou sendo muito pior. Isso pode ser explicado pelo fato de ele usar um *heap* que possui operações para se manter e fazer busca mais caras do que uma tabela *hash* usada pelo BFS. Pode-se pensar que o ganho de usar tabelas *hash* está evidenciado na comparação entre o UCS e o BFS [1]. O IDS foi o mais custoso e o que mais claramente mostrou a complexidade exponencial do algoritmo, pode-se pensar que a diferença entre fazer a poda ou não está evidenciada na comparação entre o IDS e o BFS.

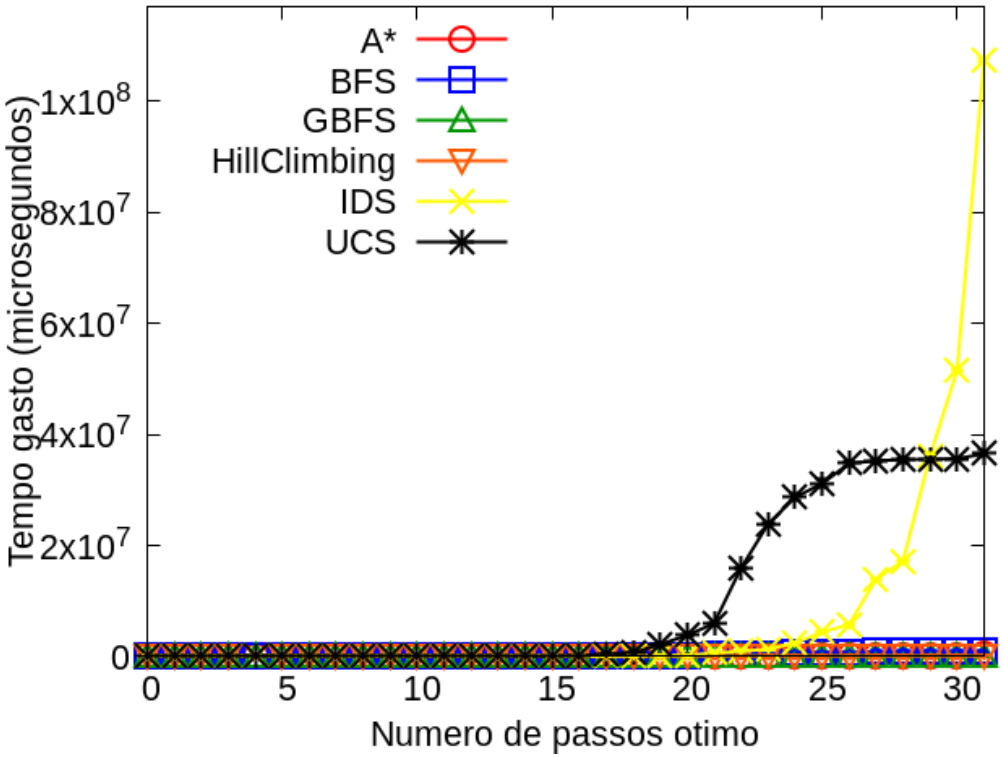

FIGURA 1. Tempo de execução de todos os algoritmos.

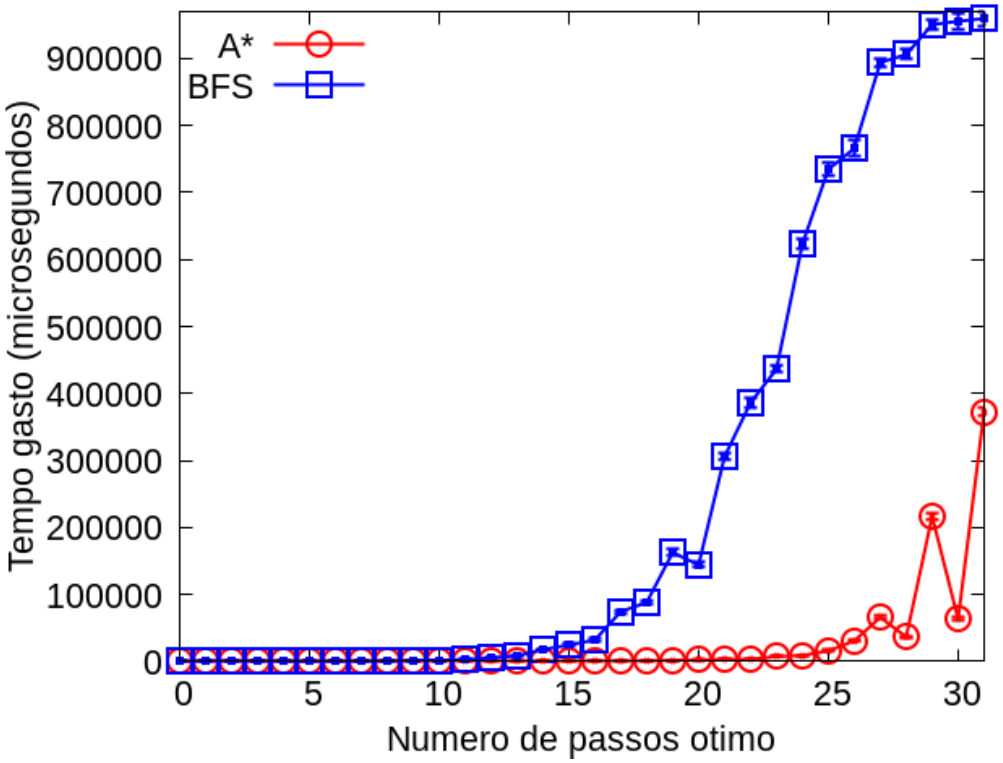

FIGURA 2. Tempo de execução dos melhores algoritmos.

Em relação à qualidade da solução encontrada o *Hill Climbing* é o pior, encontrando a solução apenas nos primeiros 8 casos de teste. Todos os outros encontram uma solução, sendo que apenas o GBFS não encontra a solução ótima. As soluções encontradas pelo guloso foram em geral muito maiores do que as ótimas, por exemplo no caso 26, a solução encontrada foi 218.

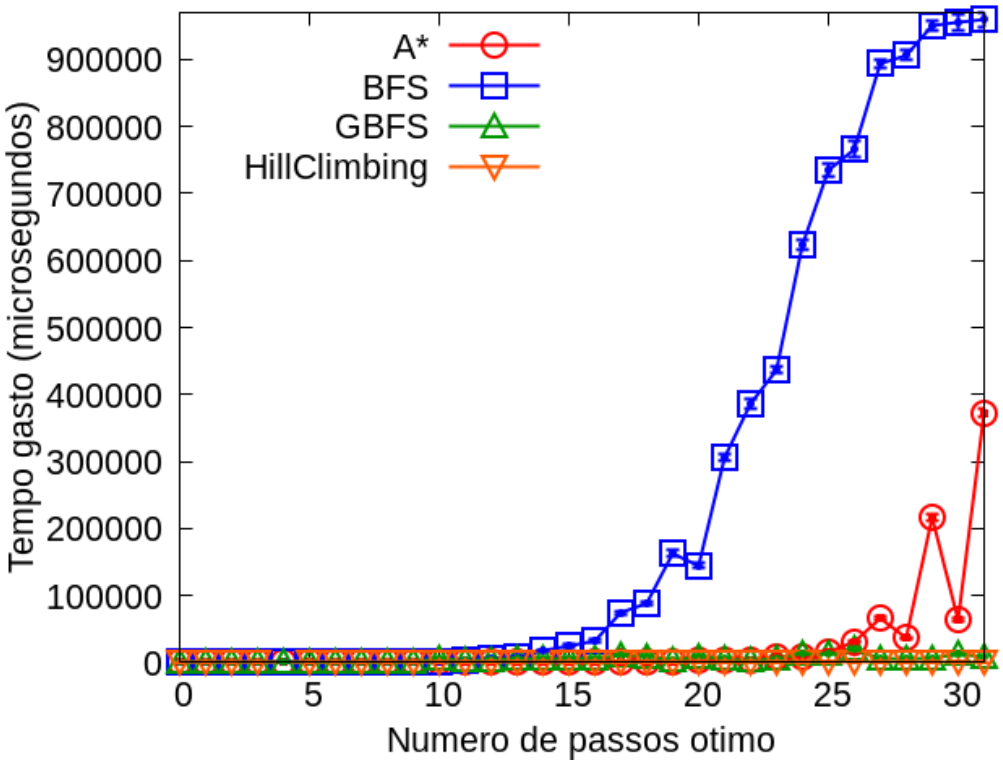

FIGURA 3. Tempo de execução dos algoritmos rápidos.

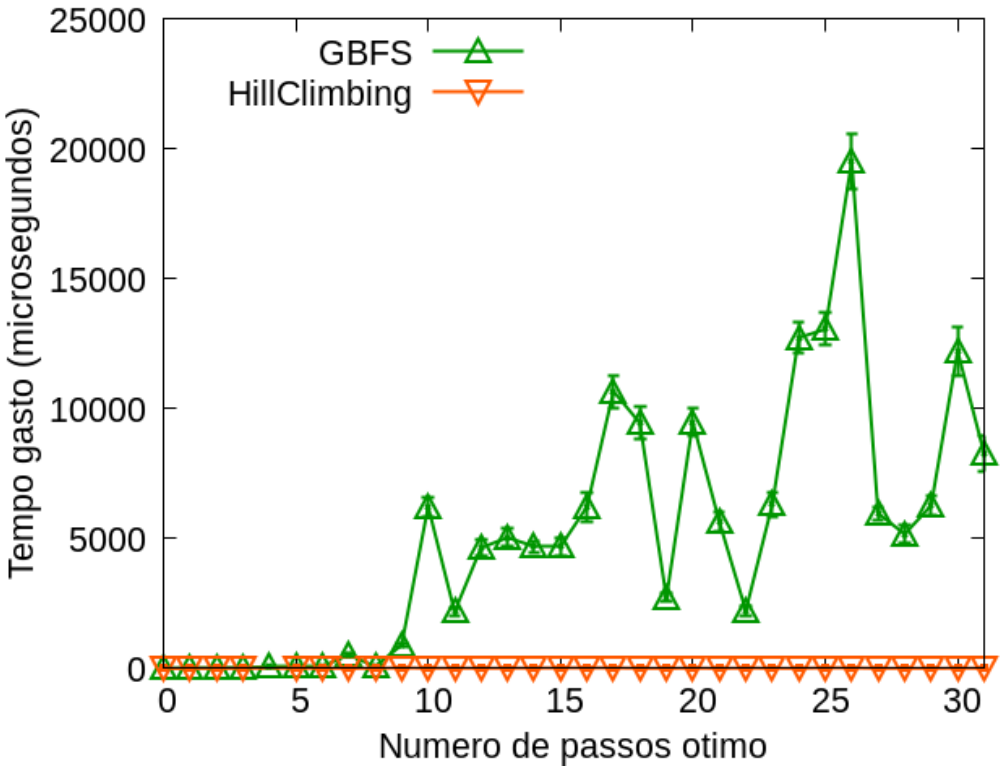

FIGURA 4. Comparação entre os tempos do Hill e do GBFS.

As soluções encontradas pelos algoritmos ás vezes diferem umas das outras. Por exemplo, no caso 31, o A* encontrou a seguinte sequência:
1 2 3 4 5 6 7 8 0
1 2 3 4 5 6 7 0 8

1 2 3 4 5 6 0 7 8
1 2 3 0 5 6 4 7 8
1 2 3 5 0 6 4 7 8
1 2 3 5 6 0 4 7 8
1 2 3 5 6 8 4 7 0
1 2 3 5 6 8 4 0 7
1 2 3 5 6 8 0 4 7
1 2 3 0 6 8 5 4 7
0 2 3 1 6 8 5 4 7
2 0 3 1 6 8 5 4 7
2 6 3 1 0 8 5 4 7
2 6 3 1 8 0 5 4 7
2 6 0 1 8 3 5 4 7
2 0 6 1 8 3 5 4 7
2 8 6 1 0 3 5 4 7
2 8 6 1 3 0 5 4 7
2 8 6 1 3 7 5 4 0
2 8 6 1 3 7 5 0 4
2 8 6 1 3 7 0 5 4
2 8 6 0 3 7 1 5 4
2 8 6 3 0 7 1 5 4
2 8 6 3 5 7 1 0 4
2 8 6 3 5 7 0 1 4
2 8 6 0 5 7 3 1 4
0 8 6 2 5 7 3 1 4
8 0 6 2 5 7 3 1 4
8 6 0 2 5 7 3 1 4
8 6 7 2 5 0 3 1 4
8 6 7 2 5 4 3 1 0
8 6 7 2 5 4 3 0 1

Enquanto o IDS encontrou:
1 2 3 4 5 6 7 8 0
1 2 3 4 5 6 7 0 8
1 2 3 4 5 6 0 7 8
1 2 3 0 5 6 4 7 8
0 2 3 1 5 6 4 7 8
2 0 3 1 5 6 4 7 8
2 3 0 1 5 6 4 7 8
2 3 6 1 5 0 4 7 8
2 3 6 1 5 8 4 7 0
2 3 6 1 5 8 4 0 7
2 3 6 1 5 8 0 4 7
2 3 6 0 5 8 1 4 7
2 3 6 5 0 8 1 4 7
2 0 6 5 3 8 1 4 7
2 6 0 5 3 8 1 4 7
2 6 8 5 3 0 1 4 7
2 6 8 5 3 7 1 4 0
2 6 8 5 3 7 1 0 4
2 6 8 5 3 7 0 1 4
2 6 8 0 3 7 5 1 4

2 6 8 3 0 7 5 1 4
2 0 8 3 6 7 5 1 4
2 8 0 3 6 7 5 1 4
2 8 7 3 6 0 5 1 4
2 8 7 3 6 4 5 1 0
2 8 7 3 6 4 5 0 1
2 8 7 3 6 4 0 5 1
2 8 7 0 6 4 3 5 1
0 8 7 2 6 4 3 5 1
8 0 7 2 6 4 3 5 1
8 6 7 2 0 4 3 5 1
8 6 7 2 5 4 3 0 1

7. Conclusão

O A* foi o melhor algoritmo para resolver o *8-Puzzle*, seguido pelo BFS. O *Hill Climbing* ,da maneira como foi modelado, não foi uma boa escolha para resolver o problema. o GBFS resolve com muita rapidez, mas a solução encontrada, normalmente é muito pior que a ótima. O UCS não é uma boa escolha, visto que o degenera para o BFS, que é mais eficiente nesse caso. O IDS foi o pior algoritmo que acha a solução ótima.