Relatório TP1 - Algoritmos de Busca

Diogo Oliveira Neiss

Apresentação das Estruturas e Modelagem dos Componentes da Busca

A solução foi implementada com orientação a objetos em Java, tendo uma classe base representando o tabuleiro do jogo e classes dedicadas para manipular esse tabuleiro em diferentes níveis de abstração.

Optei por não utilizar uma árvore explícita, implementando o comportamento dos nós através de estados armazenando informações de nós anteriores e uma classe de histórico.

Abaixo, falarei um pouco sobre as pastas da aplicação e conceitos gerais.

Classes principais - pasta application

1. Grid: Responsável por transformar o vetor de entrada em uma representação inteligente, com verificações de existência de solução, corretude da entrada e a função primitiva de sucessão, sendo capaz de gerar as transições possíveis a partir dessa instância, respeitando fronteiras. É a classe manipulada pelas outras abstrações.

   Possui o método `move`, que recebe uma grid original e uma direção e retorna a grid resultante dessa operação

2. PuzzleState: Representa o estado do algoritmo de busca em um dado instante, como se fosse o nó da árvore. Contém o grid atual, os movimentos anteriores (pais da árvore), os movimentos possíveis, gerados pela função sucessora, e um custo associado, calculado pela profundidade da árvore.

   Permite ainda a operação de exploração, em que um estado pai, a grid resultante e a direção feita são informadas e um novo estado filho é criado, com o custo e histórico de movimentações atualizado, através de um construtor especializado.

3. PuzzleHistory: Responsável por acompanhar o progresso do algoritmo de busca, servindo como uma classe auxiliar para armazenar informações, como o

número de nós visitados, tempo de execução e o último PuzzleState visitado, que será recuperado futuramente como a solução. Possui a operação de `store`, que é chamada repetidamente no algoritmo de busca, de forma a registrar um novo estado explorado. Além disso, possui funções para mostrar o caminho da solução, reconstruindo a grid original a partir do histórico de movimentos no estado.

4. SearchAlgorithm: Classe principal, sobrescrita por todas as heurísticas de busca. Ao ser criada, instancia um histórico vazio e cria o estado inicial, também vazio, a partir da grid informada no construtor, e um método abstrato `solve`, que todas devem implementar. Além disso, possui métodos que geram as métricas de execução do algoritmo, facilitando a criação de relatórios

Aspectos importantes do algoritmo

Como apresentado anteriormente, utilizamos várias classes para representar o **estado** do algoritmo: a Grid representa a configuração física, PuzzleState do nó e PuzzleHistory de todo o algoritmo de busca, mas a termos gerais todos os comportamentos necessários para a execução do algoritmo de busca são acessíveis em PuzzleState.

A função sucessora é gerada na Grid, em que a partir de uma configuração de Grid são derivados os movimentos possíveis, que são então retornados. Esse mesmo método é emcapsulado na classe PuzzleState, simplificando a chamada.

Há ainda uma versão mais simplificada, `List<Grid> getNeighbors()`, que a partir de uma grid consegue criar as grids vizinhas, abstraindo a criação de objetos.

O **cálculo de métricas** é feito após a execução do método `solve`, em que tudo foi armazenado dentro do histórico e pode ser recuperado pelo método `runInfo`.

Implementação dos algoritmos (pasta heuristics)

Todas as heurísticas implementadas herdam da classe SearchAlgorithm, possibilitando polimorfismo na execução, desde que os métodos obrigatórios da classe abstrata estejam corretamente implementados. Ademais, há um enum contendo todos os tipos de algoritmos experimentados, então é lógico criarmos uma classe responsável por abstrair a instanciação do algoritmo de busca desejado.

Com isso, foi criada na raiz a classe AlgorithmFactory, responsável por criar um objeto do tipo solicitado mediante a grid inicial e o acrônimo do algoritmo, através de

simples pattern matching. Ao longo do desenvolvimento surgiram novas hipóteses sobre heurísticas, então novos acrônimos foram criados além da especificação. São eles

- **B** - Breadth-first search (Busca em Largura simples)
- **I** - Iterative deepening search (Busca com Aprofundamento Iterativo **SEM** repetição)
- **I1** - Iterative deepening search 1 (Variação 1 da Busca com Aprofundamento Iterativo, em que os nós visitados anteriormente não são armazenados. Em resumo, é o algoritmo clássico)
- **U** - Uniform-cost search (Busca de Custo Uniforme)
- **A** - A* search (Manhattan) (Busca A* com heurística de distância Manhattan)
- **A1** - A* search (Euclidean) (Busca A* com heurística de distância Euclidiana)
- **A2** - A* search (Correct Positions) (Busca A* com heurística de quantidade de posições corretas)
- **G** - Greedy best-first (Depth) (Busca Gulosa por Profundidade)
- **G1** - Greedy best-first (Manhattan) (Busca Gulosa com heurística de Manhattan)
- **H** - Hill Climbing (Manhattan) (Heurística de distância Manhattan)
- **HR** - Hill Climbing (Random Stochastic) (Seleção Estocástica Aleatória)
- **HS1** - Hill Climbing (Simulated Annealing, Manhattan) (Simulated Annealing utilizando heurística de Manhattan)
- **HS2** - Hill Climbing (Simulated Annealing, Euclidean) (Simulated Annealing utilizando heurística Euclidiana)
- **HS3** - Hill Climbing (Simulated Annealing, Correctness) (Simulated Annealing utilizando heurística de quantidade de posições corretas)
- **R** - RandomSearch (Busca Aleatória, utilizada apenas para comparação entre os algoritmos)

Existem diferentes enums de configuração representando as opções de cada algoritmo e as heurísticas de distância, cabendo a implementação do AlgorithmFactory fazer o matching correto do acrônimo do algoritmo e instancia-lo de acordo.

Demais pastas

- models armazena enums e records utilizados ao longo do algoritmo
- inputs armazena o código que faz o parse do csv de teste para classes java
- tracking tem a classe responsável por gerar a base de dados com as métricas de todos os algoritmos

Registro de resultados

Foi criado um parâmetro adicional, -a ou —all, que pode ser passado para o programa no início, de forma que ele execute todos as heurísticas listadas acima para o dataset de teste providenciado, armazenando ao final em um .csv todas as métricas desejadas (para o IDS clássico, apenas até o 19).

Ao final de cada execução armazenamos as seguintes variáveis

- **algorithmName**: Nome do algoritmo utilizado.
- **nodesVisited**: Número de nós visitados durante a busca.
- **movementsTaken**: Número de movimentos realizados pelo algoritmo.
- **optimalMovementsTaken**: Número ótimo de movimentos para esse caso de teste.
- **solutionFound**: Indica se uma solução, ótima ou não, foi encontrada (Verdadeiro ou Falso).
- **solutionGap**: Diferença entre os movimentos tomados e os movimentos ótimos, percentualmente.
- **timeElapsed**: Tempo de execução do algoritmo em milissegundos.

Com esse dataset de resultados, carregamos os dados em um notebook python para a geração de plots e análises do restante do relatório.

Principais Diferenças entre os Algoritmos

Falarei da diferença teórica e de implementação.

1. BFS: Realiza busca em largura, implementado com uma fila. A cada iteração, removemos a cabeça, verificamos a existência de solução e jogamos os filhos no final da fila, repetindo isso até a fila estar vazia ou uma solução ser encontrada.
2. Iterative deepening search: Utiliza uma pilha para adição e remoção e adapta a busca em profundidade (DFS) restringindo sua profundidade máxima até k, que é aumentado a cada iteração. Sua grande vantagem é não precisar de

armazenamento exponencial para o armazenamento de nós visitados, porém a depender do problema, como o caso do 8-puzzle, isso é um problema, levando a tempos de execução astronômicos, motivando então a criação de uma variante com verificação de repetição (que infelizmente nem sempre encontra o ótimo)

3. **U** (Uniform-cost search): Ao invés de uma fila ou pilha, utiliza uma fila de prioridades, incorporando uma função de custo na escolha do próximo nó, invés de ordem de entrada. Expande o nó com o menor custo cumulativo até o momento, que é interpretado com o nó de menor profundidade. É ótimo e completo, o que significa que ele sempre encontrará a solução mais curta se uma solução existir, embora possa expandir mais nós que o necessário.
4. **A** (A* search): Combina os pontos fortes da busca de custo uniforme e da busca gulosa. Ele utiliza uma função de avaliação que é a soma de dois componentes: o custo do caminho do estado inicial ao nó atual e uma heurística que estima o custo do caminho mais barato do nó atual ao estado objetivo. Assim como a busca uniforme, utiliza uma fila de prioridades. A busca A* é completa e ótima, desde que a heurística utilizada seja admissível.
5. **G** (Greedy best-first): É uma estratégia de busca informada que utiliza apenas a função heurística para decidir qual nó expandir a seguir. Ele seleciona o nó mais promissor com base na estimativa de custo do nó ao objetivo. Enquanto é mais eficiente em termos de tempo em muitos casos, não garante uma solução ótima.
6. **H** (Hill Climbing): Começa com uma solução arbitrária e itera para melhorá-la. Em cada iteração, o algoritmo seleciona o vizinho mais próximo com o valor mais alto e se move para ele. A busca termina quando não há melhorias possíveis. A principal desvantagem deste método é que ele pode ficar preso em máximos locais que no caso de 8puzzle podem não ser soluções viáveis do problema. Foram implementadas 3 classes:
    - Normal: Permite movimentos laterais até um limite k.
    - Estocástico: Monta uma lista de vizinhos que melhora a solução e escolhe aleatoriamente um deles
    - Simulated Annealing: Algoritmo de metropolis com decaimento de temperatura regulando exploration vs exploitation.
7. **R** (RandomSearch): Como o nome sugere, esta abordagem realiza buscas aleatórias no espaço de estados. Em cada iteração, ele escolhe um sucessor aleatoriamente e se move para ele. Embora possa ser útil em alguns cenários,

especialmente quando o espaço de busca é vasto e a paisagem é complexa, não oferece garantias de encontrar uma solução ou de otimização. No escopo desse trabalho, só foi implementada para gerar métricas de comparação.

Além disso, para evitar tempos proibitivos de execução, foi restringido para apenas visitar nós não visitados anteriormente, garantindo convergência, mesmo que em uma solução incorreta.

Especificação das Heurísticas Utilizadas

Foram utilizadas “quatro” heurísticas comprovadamente admissíveis (nunca superestimam o custo real para alcançar o objetivo).

1. Distância manhattan: Número de movimentos laterais para atingir as posições corretas, ignorando a restrição da troca pelo quadrado em branco
2. Distância euclidiana: Mesma coisa da manhattan, porém considera movimentações diagonais no espaço.
3. Quantidade de peças na posição correta: Representa o número de reposicionamentos de peças necessários para atingir a solução correta, não importa o quão longe.
4. Quantidade de movimentos feitos até o momento: Representa a profundidade na árvore de busca

Como as três primeiras heurísticas representam versões relaxadas do problema n-puzzle e são garantidamente inferiores as movimentações reais, temos sua admissibilidade como heurística.

A heurística 4 não é uma heurística propriamente dita, apenas o custo da solução atual, então seu uso no algoritmo guloso o reduz a um algoritmo de busca sem informação.

Exemplos de Soluções Encontradas pelos Algoritmos

Os algoritmos analisados variaram em termos de eficácia, com alguns encontrando soluções mais rapidamente e outros visitando mais nós. Observamos que o A* foi o melhor, então mostrarei uma saída exemplo, além de informações de execução

```
Solution: 7 movements
Solution visited 8 nodes.
Steps taken in solution: DOWN LEFT UP UP RIGHT DOWN DOWN
```

```
1 5 2
4 8
7 6 3

1 5 2
4 8 3
7 6

1 5 2
4 8 3
7   6

1 5 2
4   3
7 8 6

1   2
4 5 3
7 8 6

1 2
4 5 3
7 8 6

1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8
```

Como podemos observar, o número de nós visitados foi muito próximo da solução ótima e o algoritmo ainda consegue mostrar todas as movimentações feitas para chegar no estado final.

Análise Quantitativa

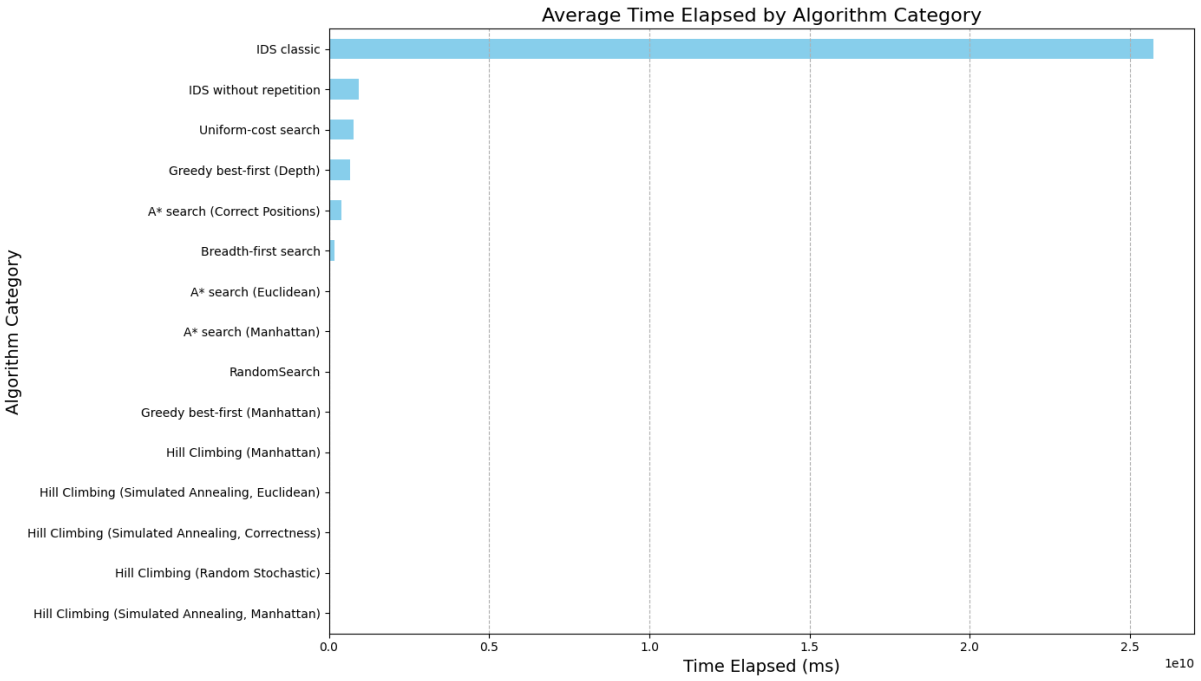
Executamos os 31 casos de teste para todos os algoritmos, exceto o IDS clássico com repetição, que só foi até o caso 20, já que estava demorando horas para rodar tudo.

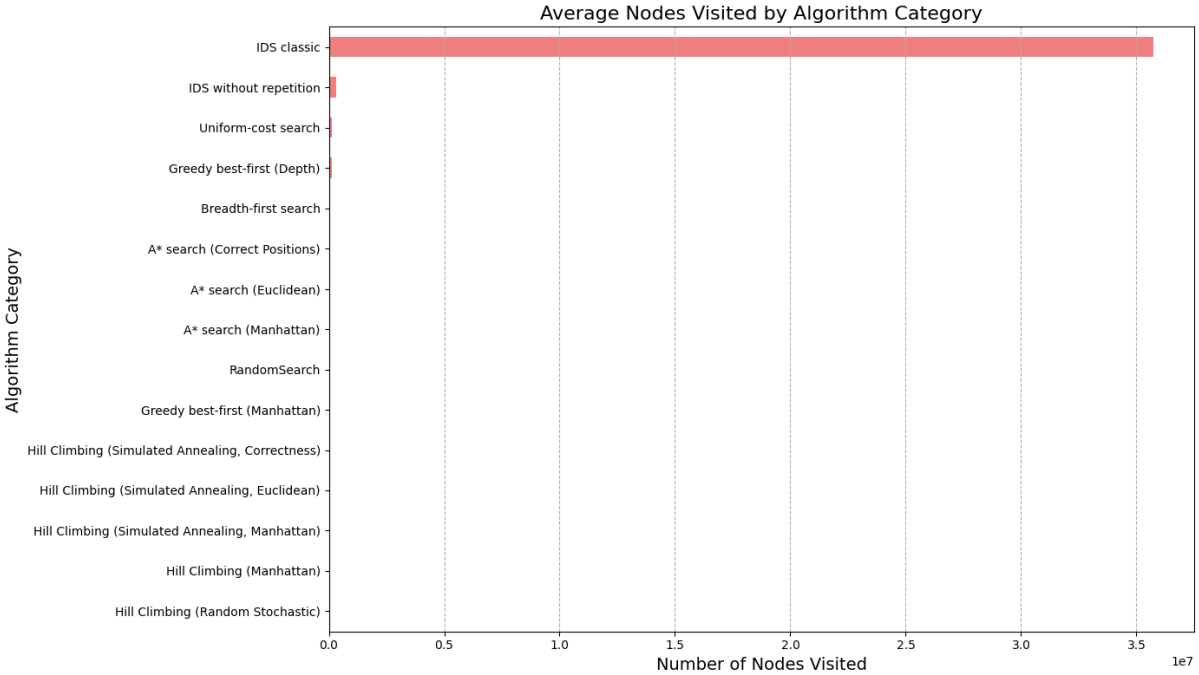
Os gráficos gerados no notebook oferecem uma análise quantitativa levando em consideração essas métricas principais.

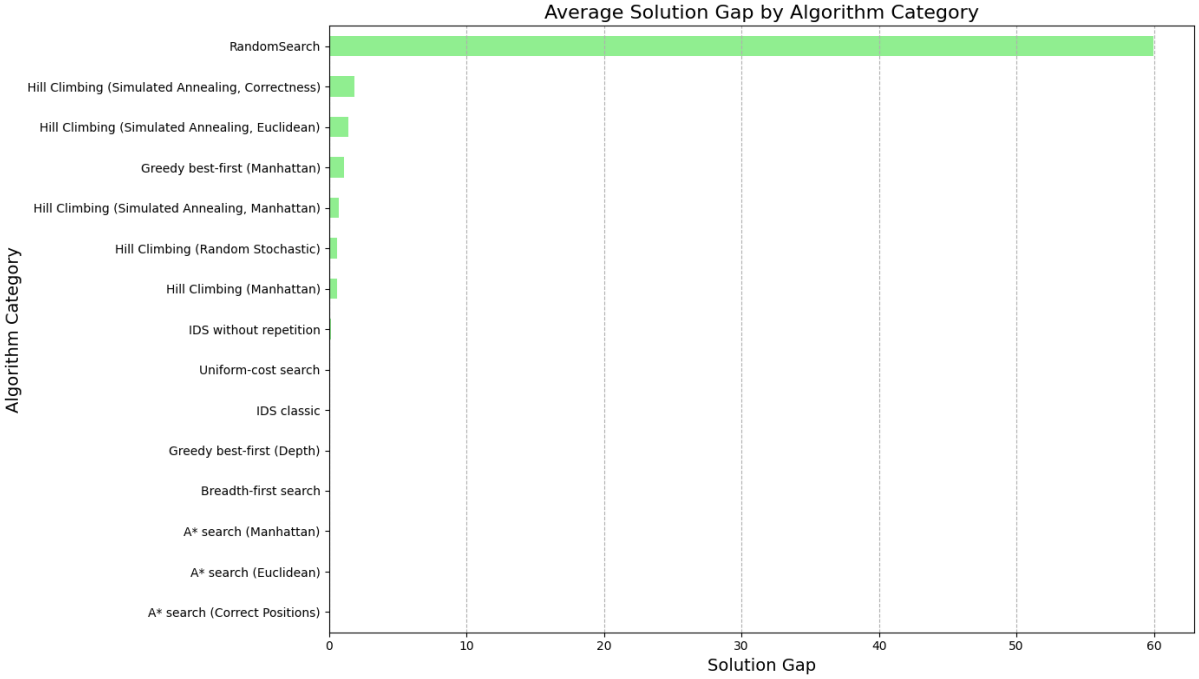
- **Nós Visitados**: Alguns algoritmos visitaram significativamente mais nós do que outros, o que pode indicar uma busca mais exaustiva, mas possivelmente menos eficiente.

- **Tempo de Execução**: O tempo de execução variou entre os algoritmos. Alguns foram mais rápidos, possivelmente devido a heurísticas mais eficientes ou a abordagens de busca mais direcionadas, porém isso deve ser olhado com cuidado, analisando métricas de corretude da solução, já que de nada adianta uma solução aproximativa que não resolve o problema proposto.
- **Solution Gap**: $\frac{|\hat{y}-y|}{y}$, com $\hat{y}$ sendo a solução encontrada pelo algoritmo e y o ótimo
- **Solution Found**: Indica se uma solução válida foi encontrada, deve ser usado como filtro para análise das métricas de tempo de execução e nós visitados.

Vamos começar analisando essas métricas para todos os algoritmos.





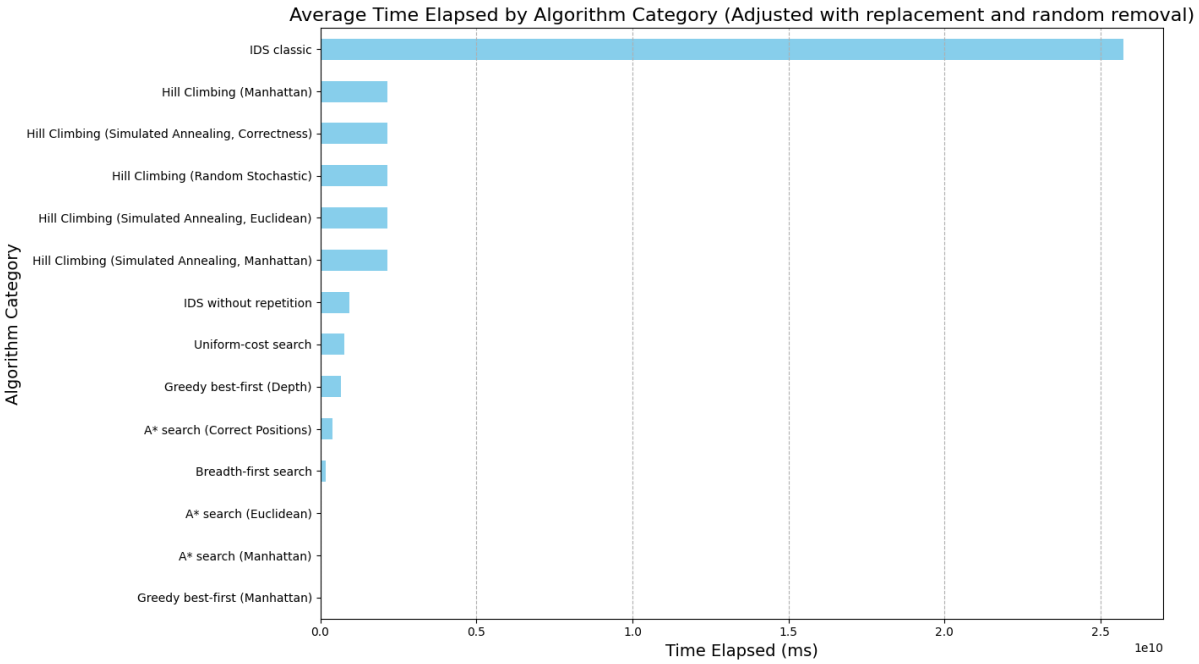


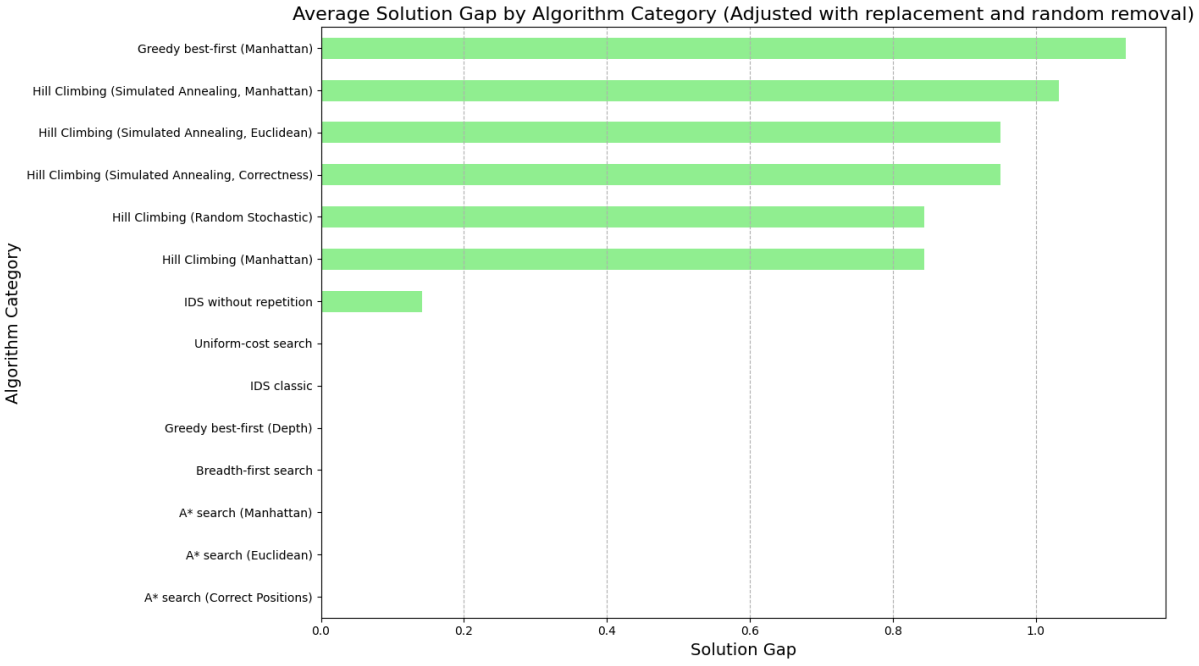
Era de se esperar que o algoritmo aleatório tivesse o maior gap de soluções, porém assusta um pouco o tempo de execução do IDS clássico, que é a variante que permite repetições de estados.

Vamos agora analisar o gap de soluções: Para isso, removi a busca aleatória e fiz uma substituição seletiva de métricas para os casos em que uma solução não foi

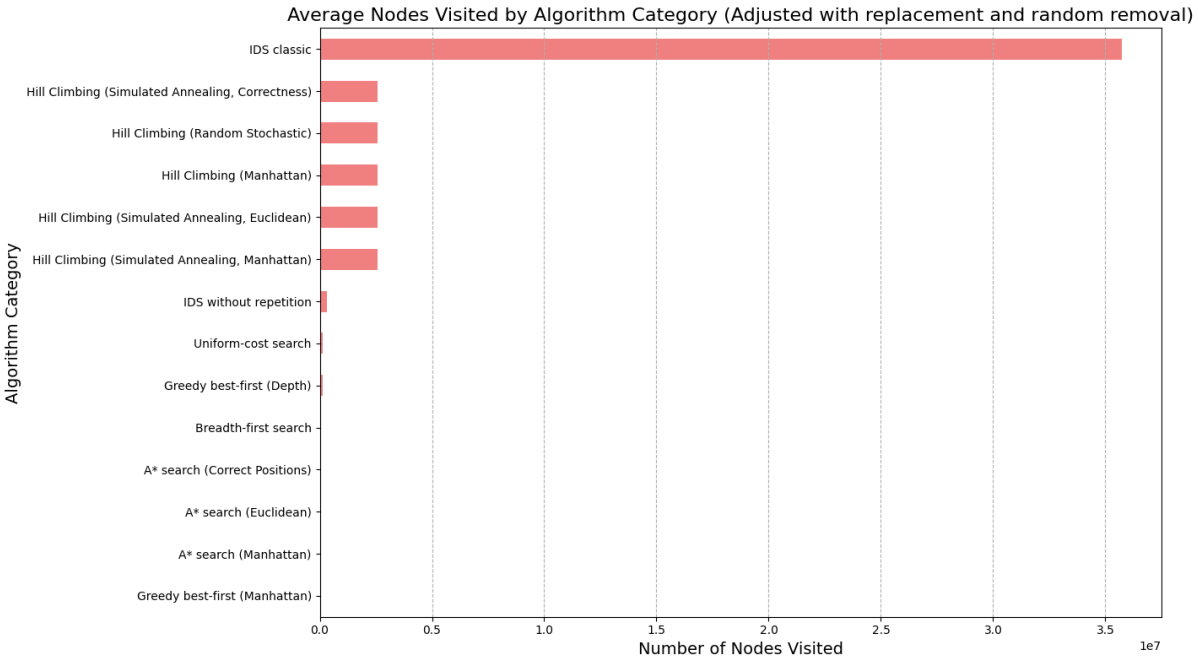
encontrada, substituindo pela média do pior algoritmo. Isso facilita a visualização de algoritmos não completos nos gráficos.

Isso foi necessário devido a algoritmos extremamente rápidos e com gaps relativamente baixos, porém com soluções incorretas.



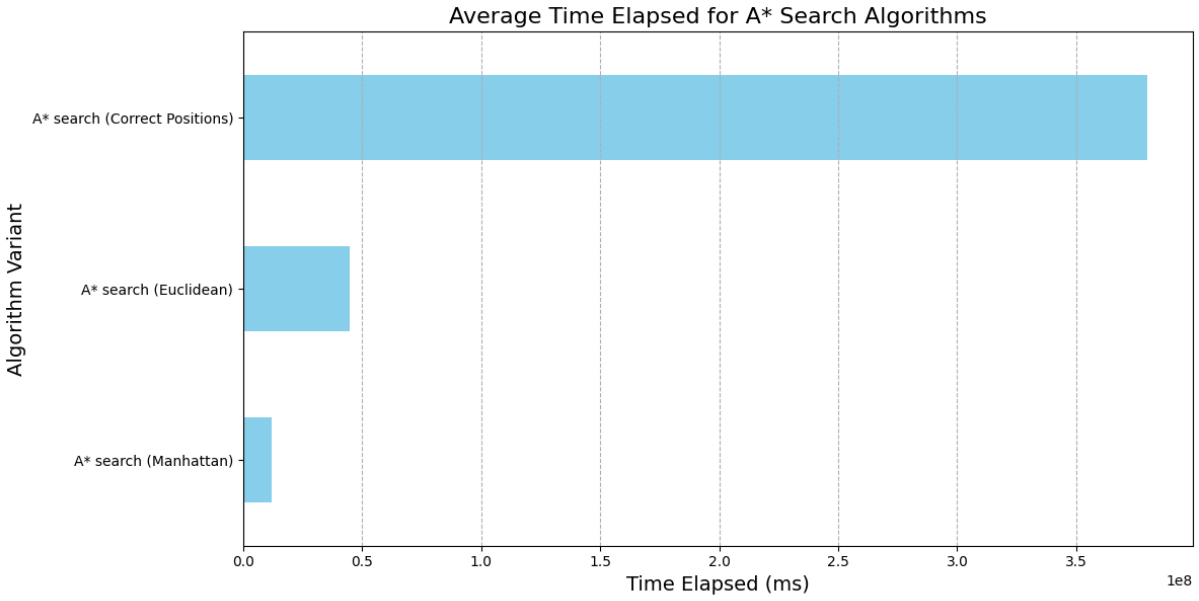


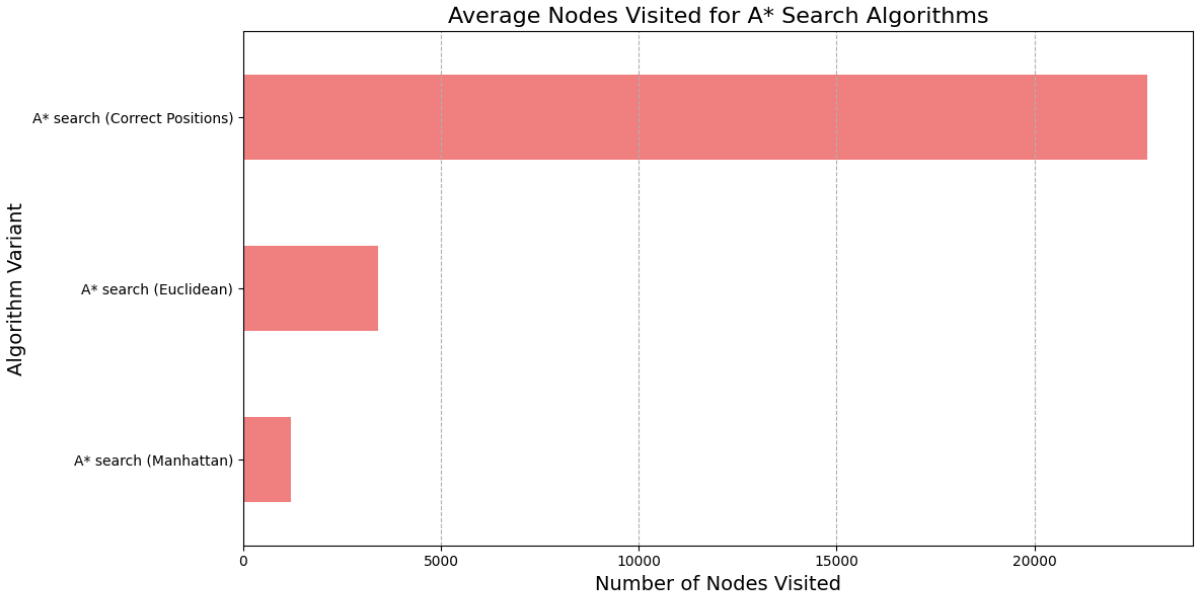
A hipótese experimental se manifesta, com todos os algoritmos ótimos mostrando gap de 0% e o iterative deepening search, que foi modificado para impedir revisitar nós, deixando de ser ótimo (mas ainda completo).

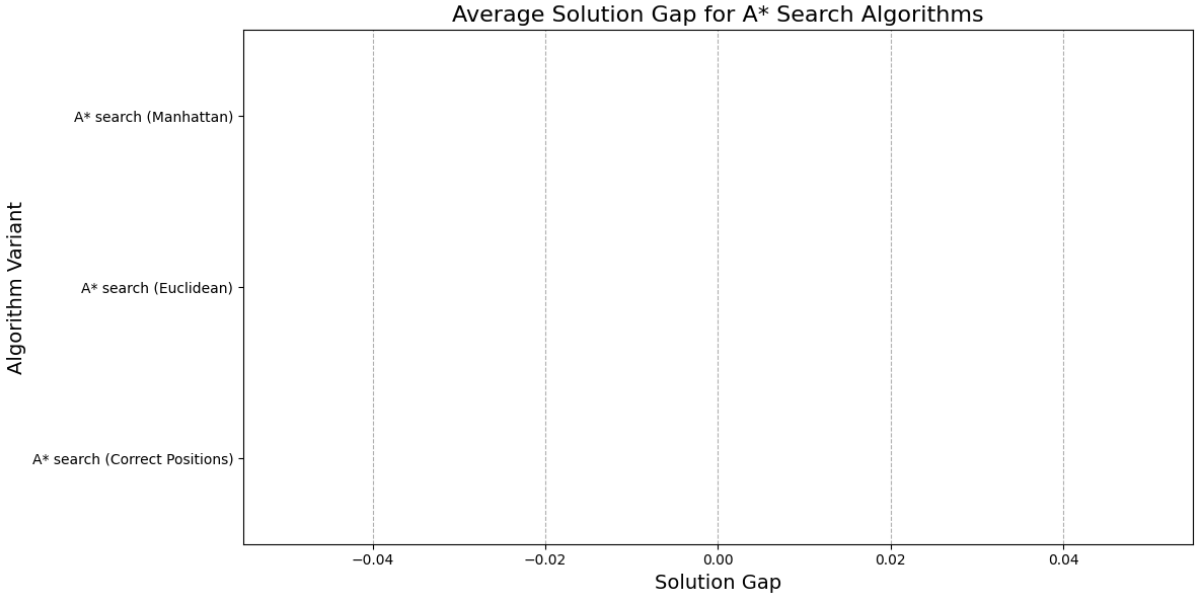


Mesmo com a remoção do aleatório, ainda temos que o IDS clássico visita milhões de vezes mais que os demais nos casos mais fáceis, imagine nos mais difíceis!

Vamos analisar agora as heurísticas no algoritmo A*

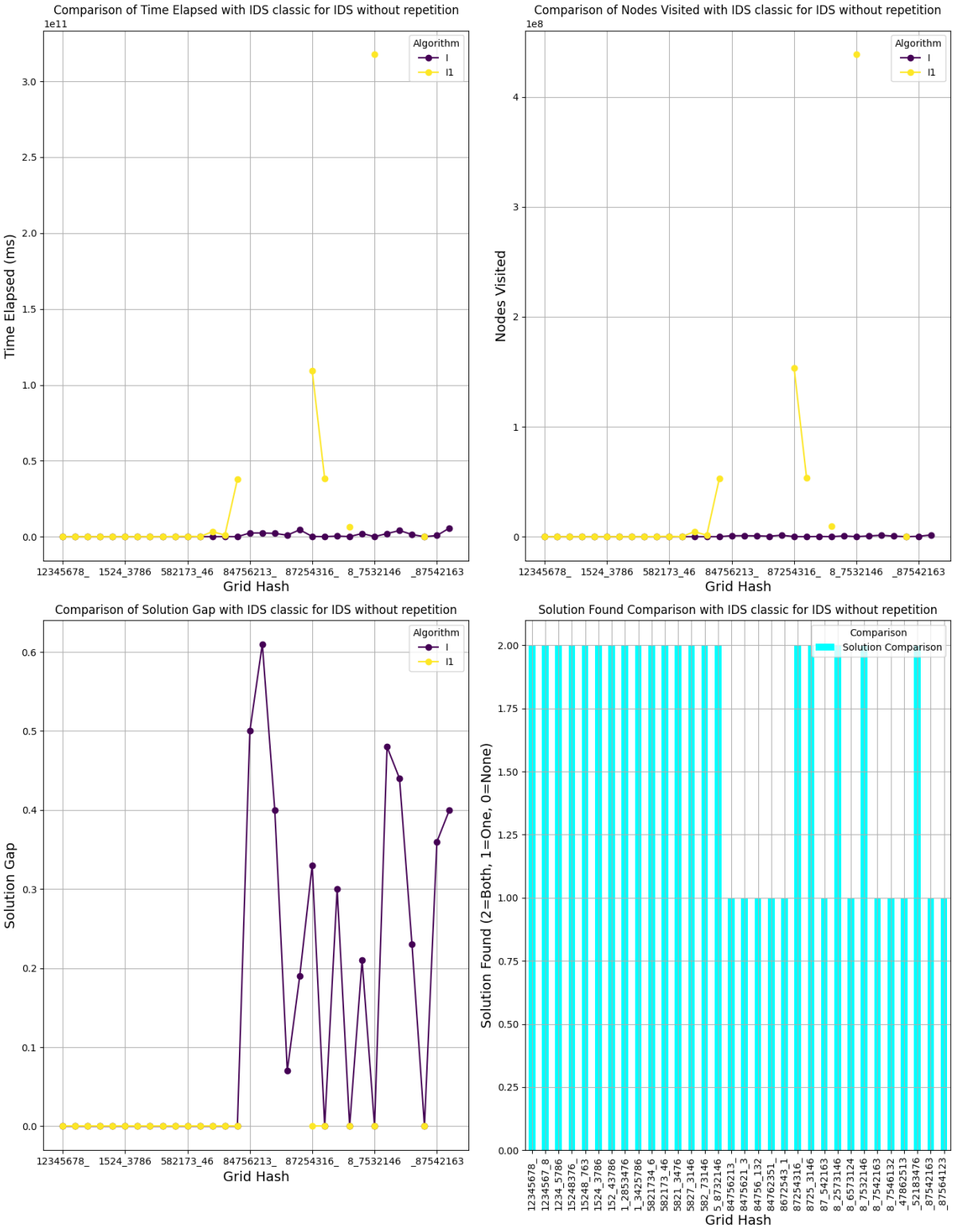

Average Time Elapsed for A* Search Algorithms
A* search (Correct Positions)
A* search (Euclidean)
A* search (Manhattan)
Algorithm Variant
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
1e8
Time Elapsed (ms)


Average Nodes Visited for A* Search Algorithms
A* search (Correct Positions)
A* search (Euclidean)
A* search (Manhattan)
Algorithm Variant
0
5000
10000
15000
20000
Number of Nodes Visited

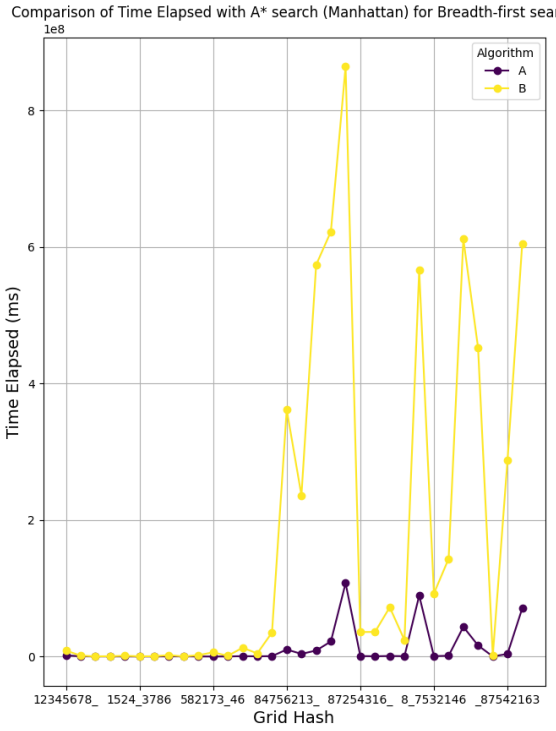

Average Solution Gap for A* Search Algorithms
A* search (Manhattan)
A* search (Euclidean)
A* search (Correct Positions)
Algorithm Variant
−0.04
−0.02
0.00
0.02
0.04
Solution Gap

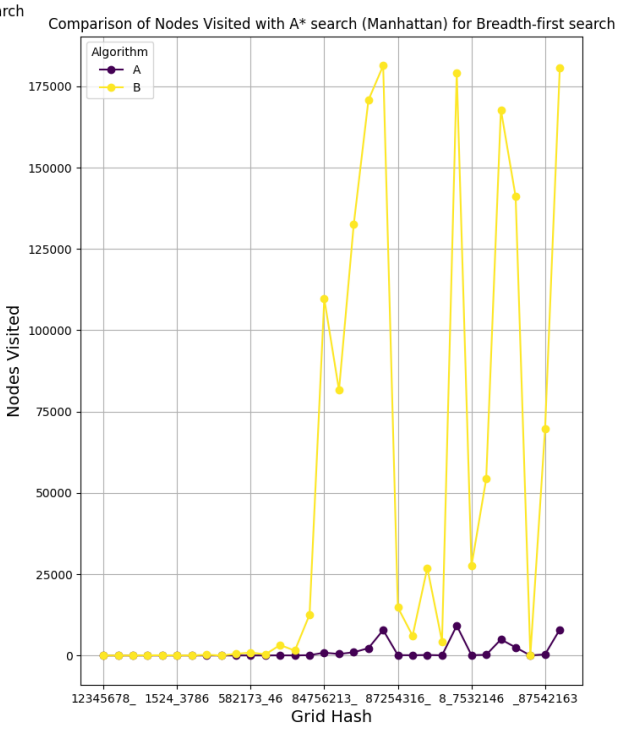
A heurística relaxada que mais se aproxima do caso real é a manhattan, então é esperada essa manifestação experimental em melhor desempenho.

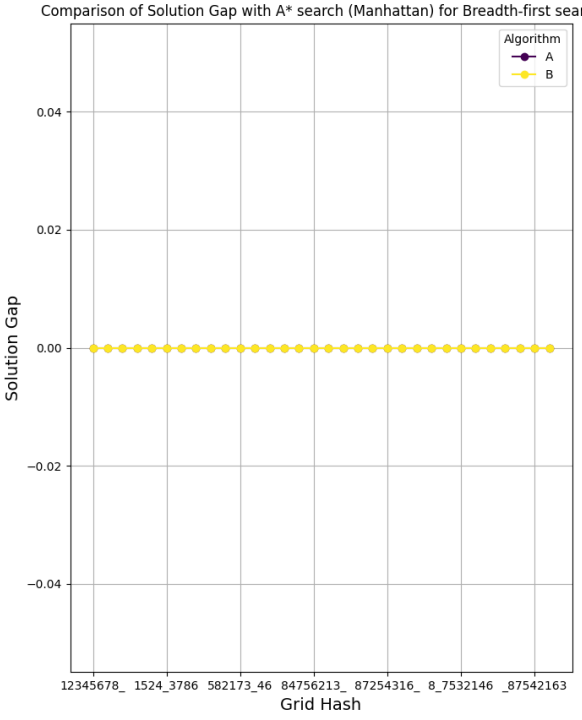
Vamos ver agora a comparação de vários algoritmos com um algoritmo base. Inicialmente, iremos demonstrar o quão melhor o IDS com repetição é frente ao com repetição

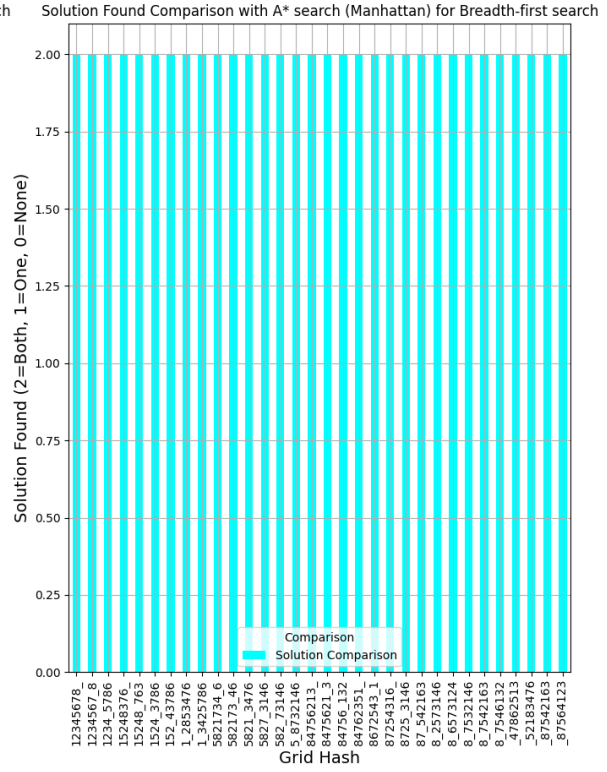


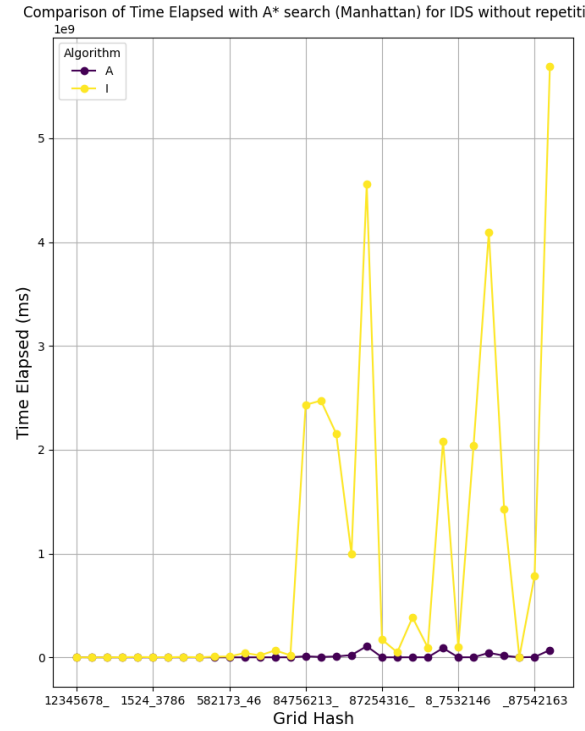
Vamos agora ver como o A* é o melhor algoritmo para o problema do 8-puzzle, tanto em termos de completude, otimalidade, tempo de execução e nós expandidos

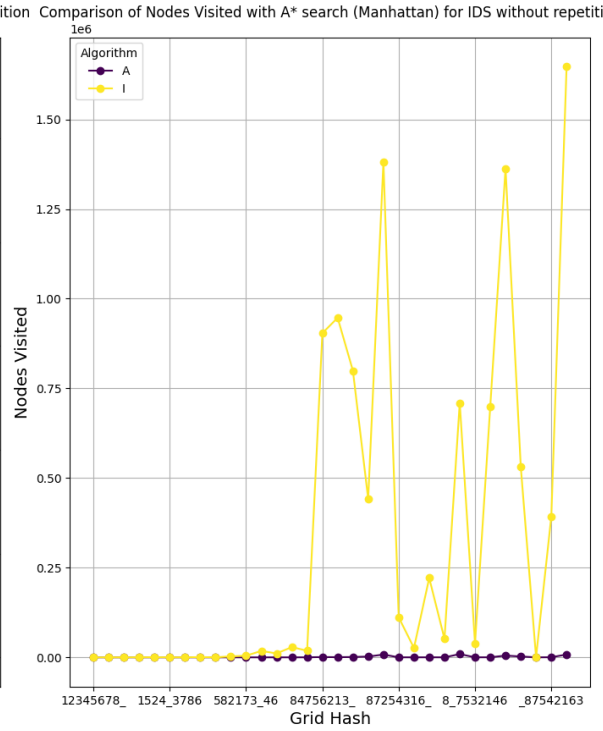

Comparison of Time Elapsed with A* search (Manhattan) for Breadth-first search
1e8
Time Elapsed (ms)
Grid Hash
Algorithm
A
B
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163

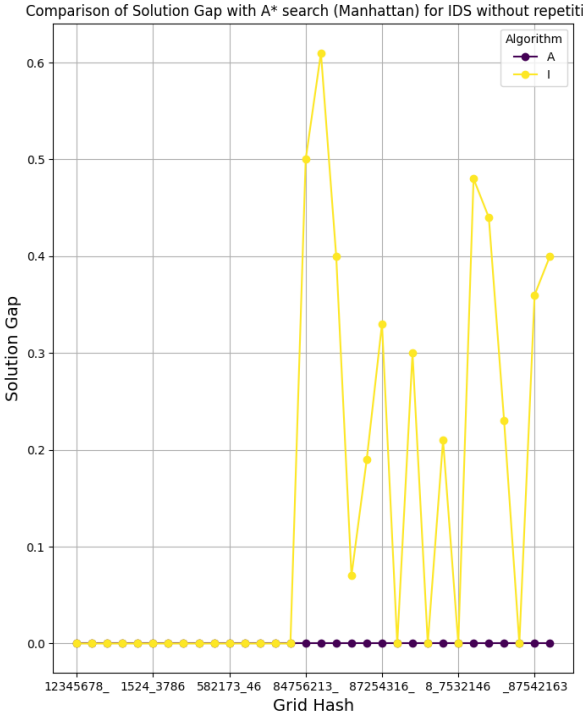

Comparison of Nodes Visited with A* search (Manhattan) for Breadth-first search
Nodes Visited
Grid Hash
Algorithm
A
B
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163

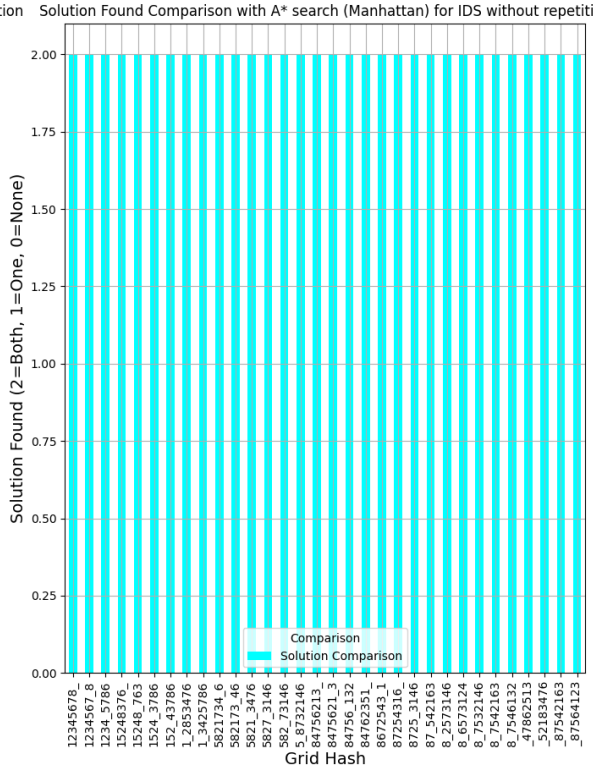

Comparison of Solution Gap with A* search (Manhattan) for Breadth-first search
Solution Gap
Grid Hash
Algorithm
A
B
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163


Solution Found Comparison with A* search (Manhattan) for Breadth-first search
Solution Found (2=Both, 1=One, 0=None)
Grid Hash
Comparison
Solution Comparison

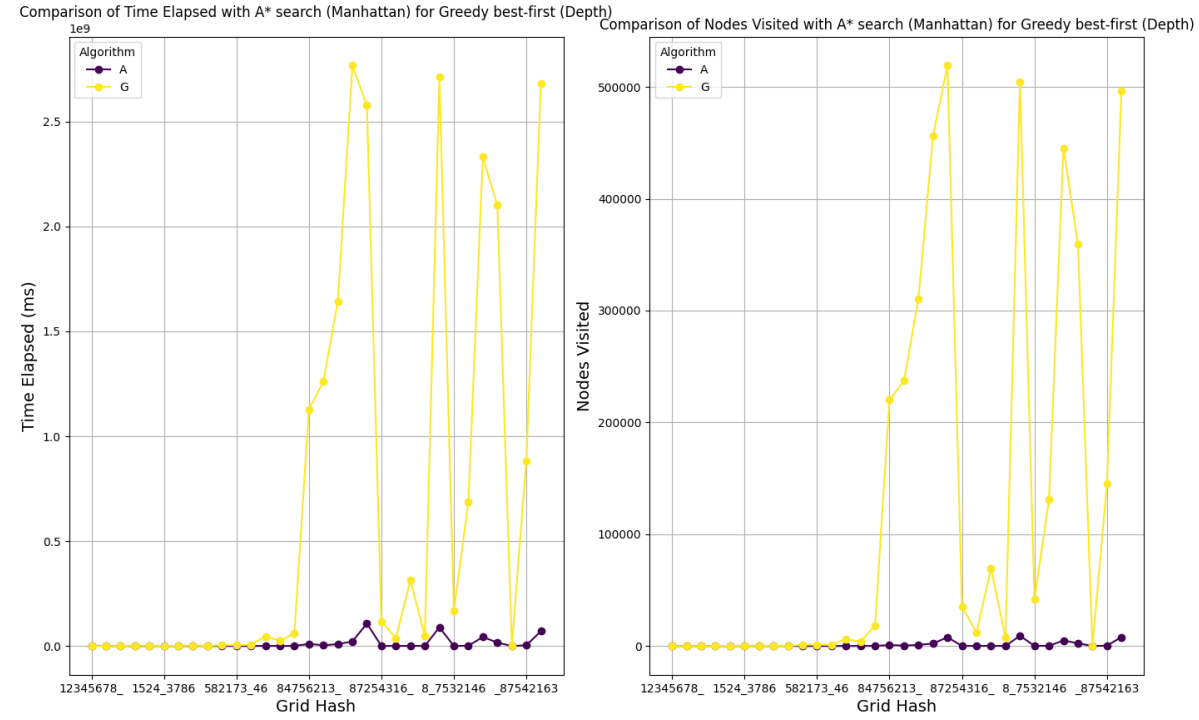


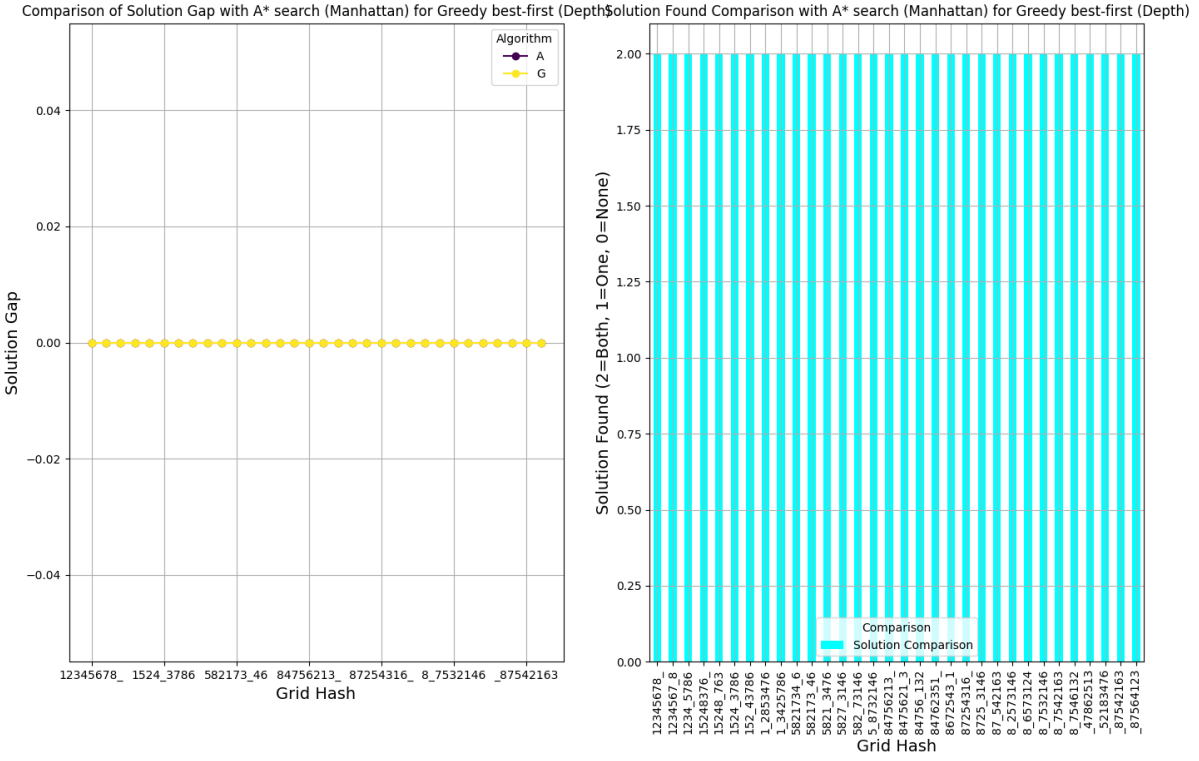




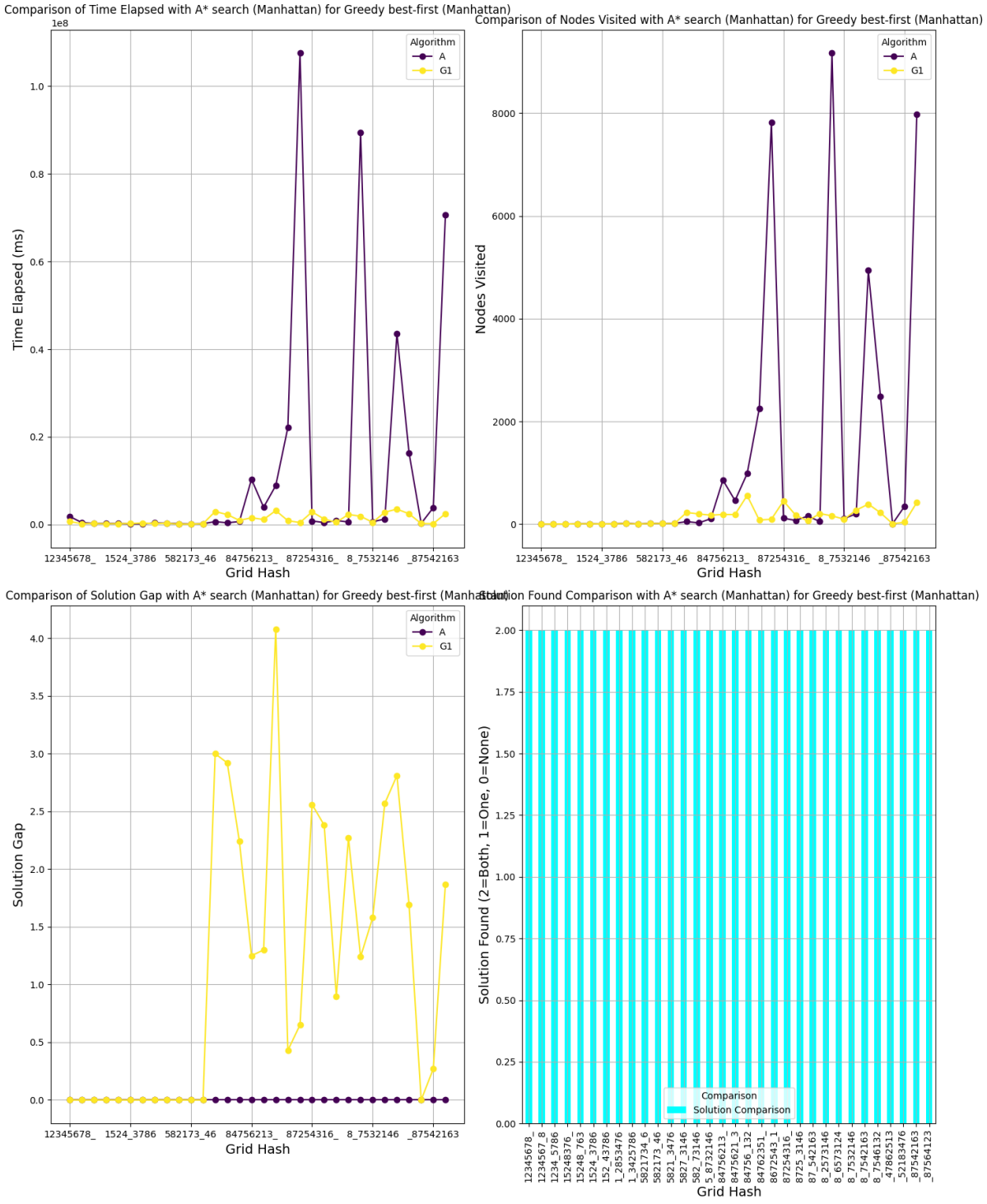


Interessante ver o tempo de execução contra a alternativa gulosa



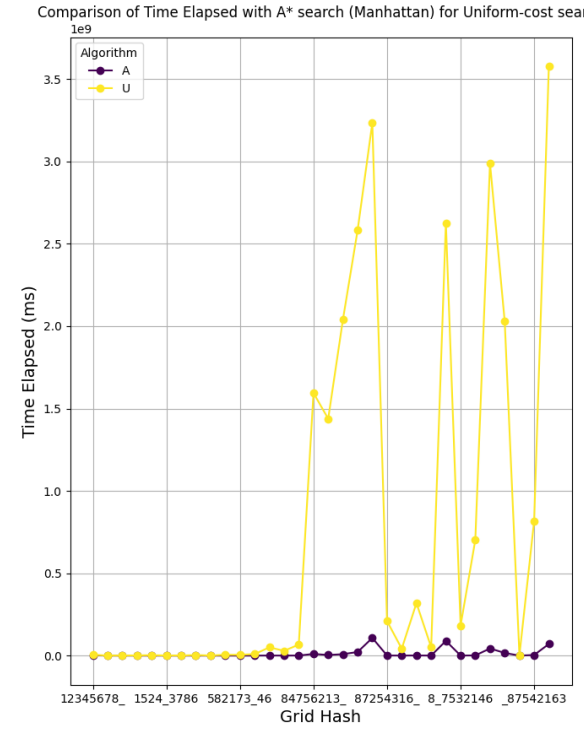


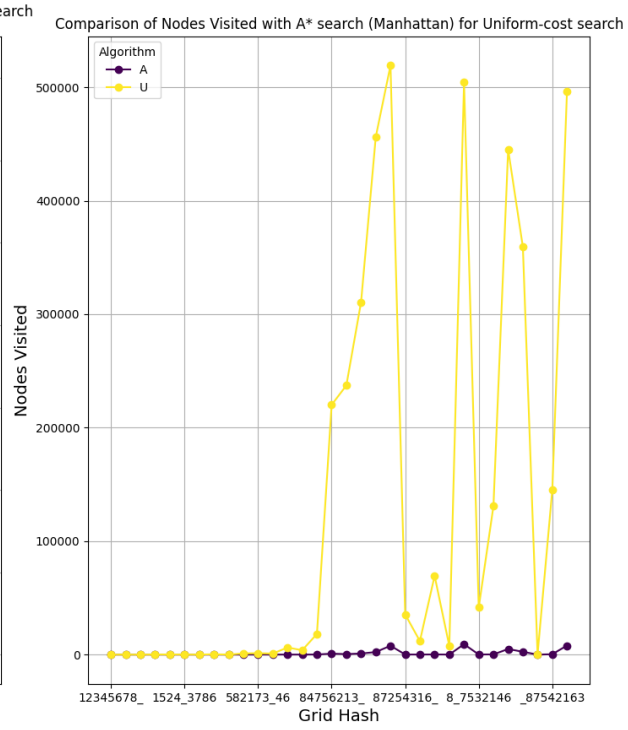
Nesse caso, a heurística de profundidade garante otimalidade, porém o algoritmo demora a convergir

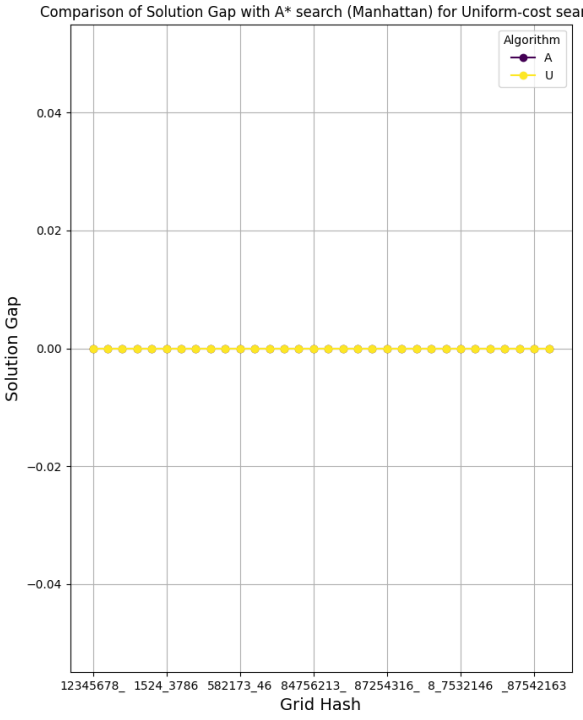


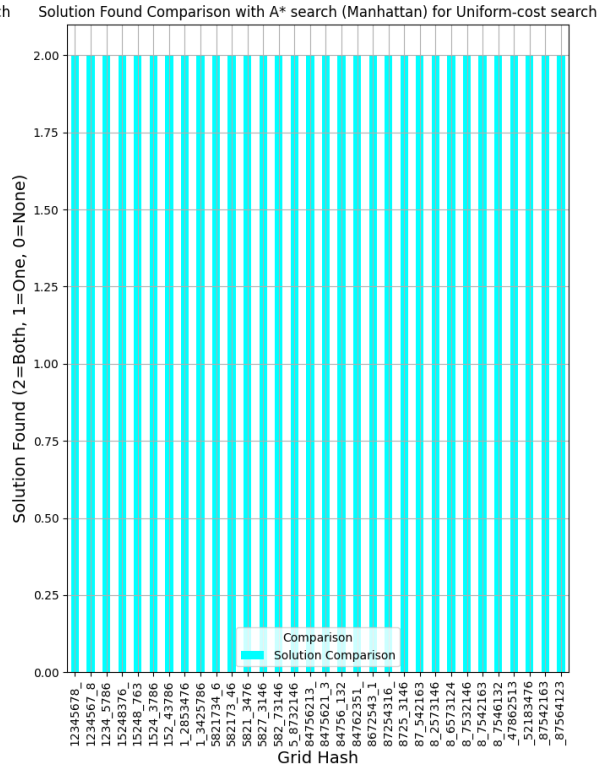
Aqui o ponto principal é a otimalidade: a busca gulosa não encontra o ótimo, apesar de ser completa

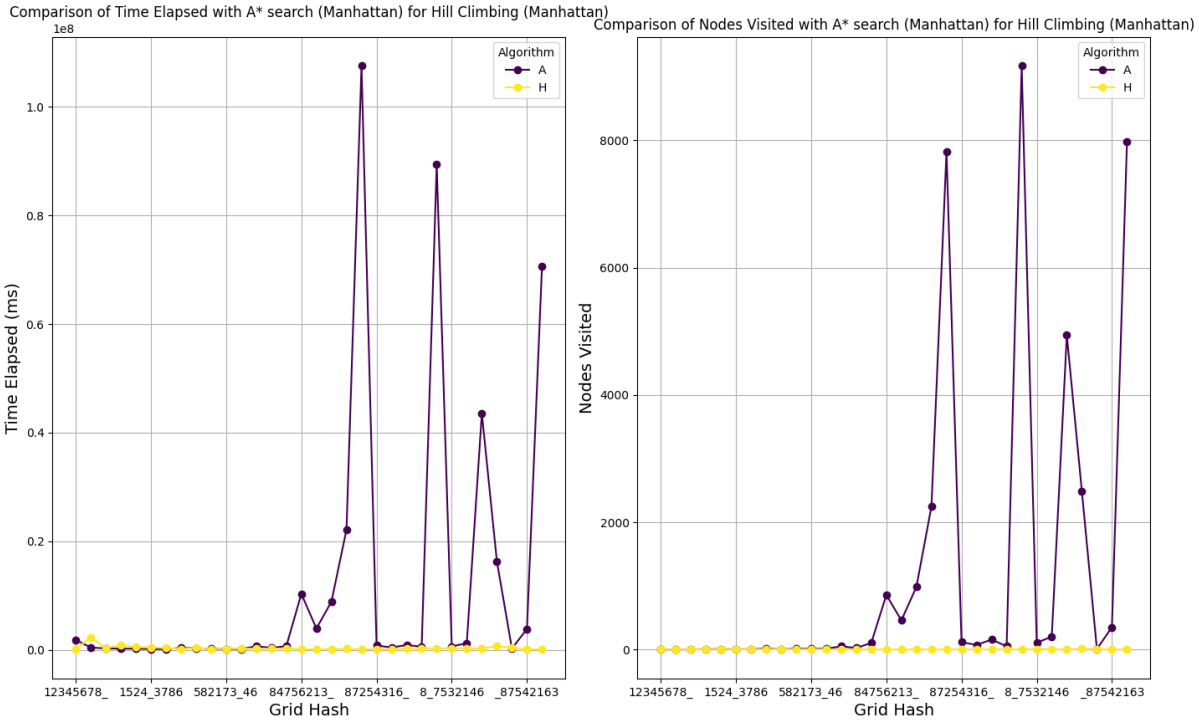
Comparando o A* com a busca uniforme, algoritmos que “parecem”

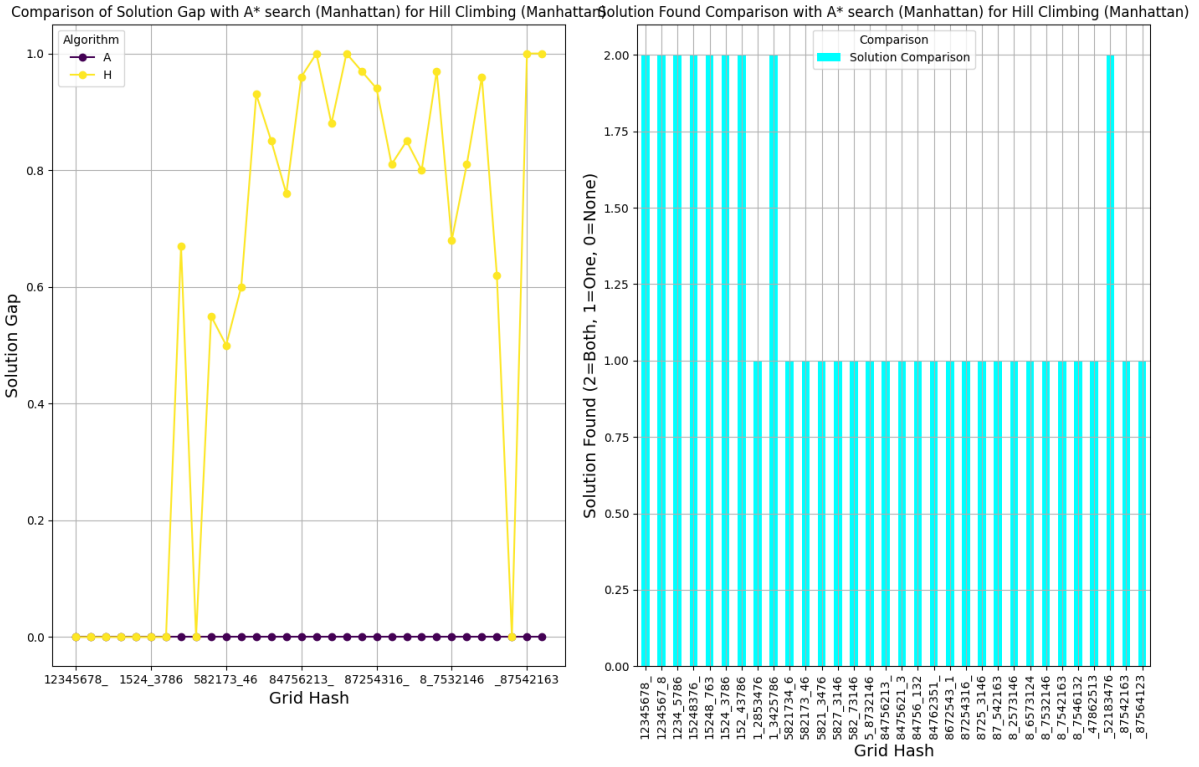

Comparison of Time Elapsed with A* search (Manhattan) for Uniform-cost search
1e9
Algorithm
A
U
Time Elapsed (ms)
Grid Hash
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163


Comparison of Nodes Visited with A* search (Manhattan) for Uniform-cost search
Algorithm
A
U
Nodes Visited
Grid Hash
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163

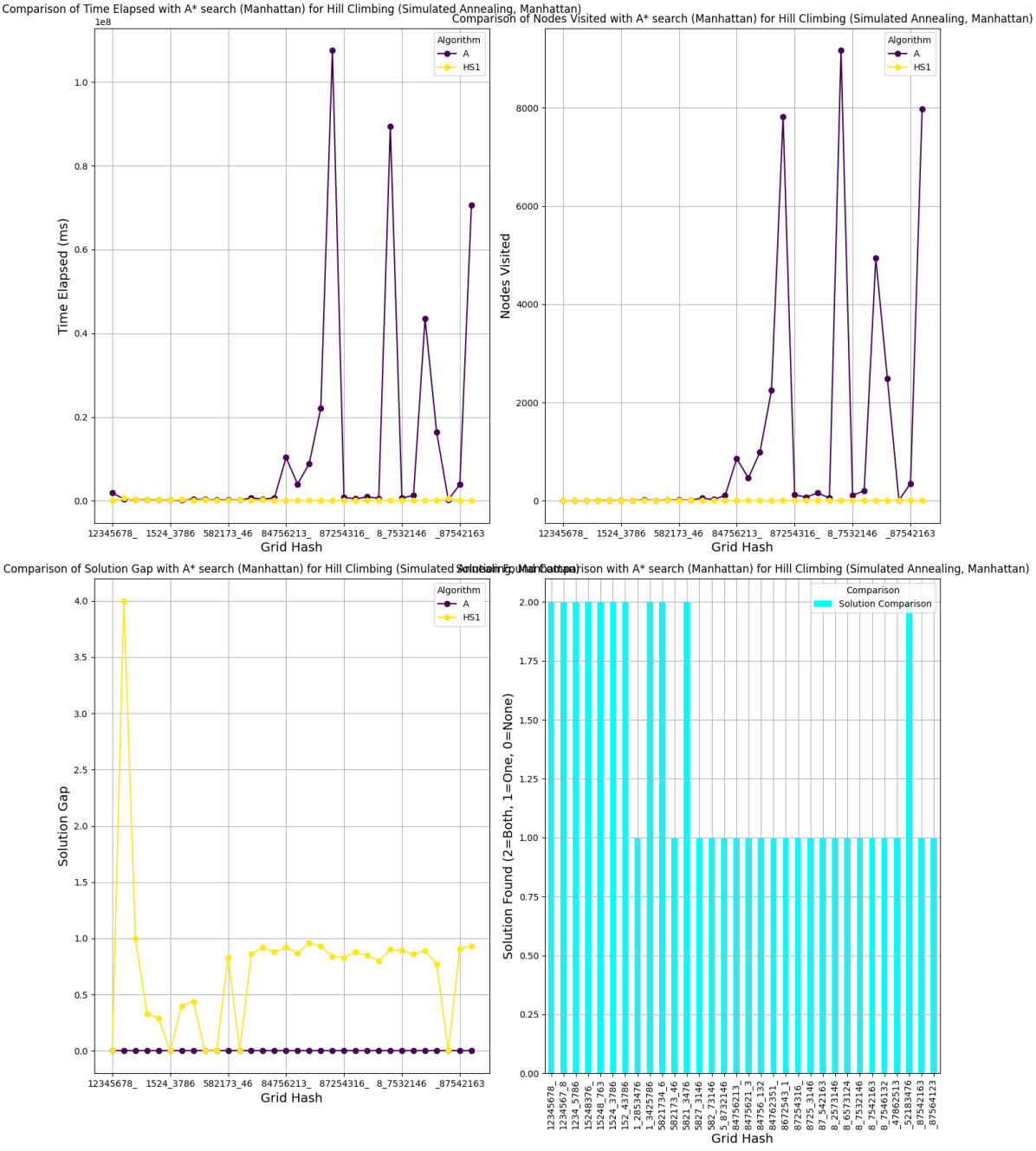

Comparison of Solution Gap with A* search (Manhattan) for Uniform-cost search
Algorithm
A
U
Solution Gap
Grid Hash
12345678_
1524_3786
582173_46
84756213_
87254316_
8_7532146
_87542163


Solution Found Comparison with A* search (Manhattan) for Uniform-cost search
Solution Found (2=Both, 1=One, 0=None)
Comparison
Solution Comparison
Grid Hash





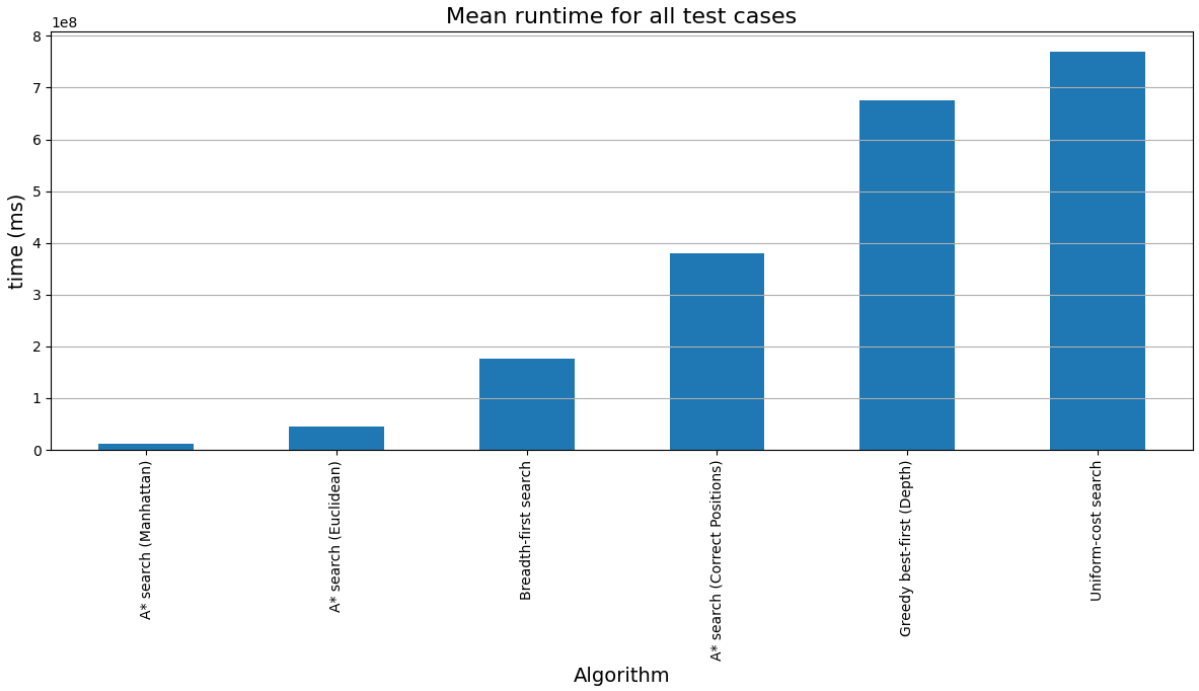
Mesmo extremamente rápido, o hill climbing quase nunca encontra a solução!



Por fim, vamos ver os algoritmos de Hill Climbing

Discussão dos Resultados Obtidos

Foi observado que em tempo de execução o A* é o melhor algoritmo em todos os casos, superando todos os outros em diversas métricas.



Os algoritmos de hill climbing se mostraram ineficientes para solução, se quiséssemos uma solução boa com eles seria necessário fazer dezenas de reinícios aleatórios. Dentre eles, a variante estocástica e a com simulated annealing performaram muito bem, de forma que a exploração dessas mesmas métricas para variantes com outros valores de temperatura e decaimento, além de mais reinícios aleatórios, sejam interessantes em trabalhos futuros.

Resumindo, concluímos que

- O algoritmo IDS clássico, que não armazena nós visitados, é **impraticável** para o 8-puzzle, sendo necessário adaptá-lo para reduzir soluções repetidas
- A heurística manhattan é a melhor experimentalmente, o que se ajusta com a teoria, já que é a heurística menos relaxada, garantidamente maior que as outras duas
- A* tem o melhor desempenho dos algoritmos com informação
- Busca uniforme tem o melhor desempenho entre algoritmos sem informação considerando tempo e memória, com o IDS sendo lento demais e o BFS consumindo memória excessiva para casos difíceis

- Algoritmos de hill climbing requerem muitos reinícios aleatórios ou formas de armazenar soluções prévias para convergirem no ótimo. É interessante o paralelo com algoritmos genéticos, em que isso é promovido por elitismo, em que gerações boas anteriores se preservam nas futuras, evitando uma subida cega de morro.