Documentação TP1 IIA
Victor Augusto de Lima Cruz
2020054854

1. Introdução

O trabalho consiste na implementação e comparação de 6 algoritmos de busca, com informação sem informação e local. Os algoritmos são: Busca em largura, Busca de Aprofundamento Iterativo, Busca de custo uniforme, Busca Gulosa, Busca A* e Hill Climbing.

O problema que será resolvido com as buscas é o 8-puzzle que consiste em um tabuleiro 3x3 com 8 peças numeradas e um espaço vazio. O objetivo é ordenar os números movendo apenas o espaço vazio com elementos de casas adjacentes. Encontrar essa solução ótima é NP-difícil e será feita com os algoritmos de busca citados acima.

2. Modelagem

2.1 Estado

Em sua maioria para modelar o estado é usado uma matriz 3x3 da mesma forma em que o tabuleiro é visto. O estado em todos os algoritmos está na classe “Node” que contém uma matriz correspondente ao estado, uma referência para o pai e um custo, caso o algoritmo utilize isso. Para facilitar a visualização e funcionamento de igualdade e comparações utilizados nos algoritmos, foram definidos os atributos:
__str__, representação da matriz como string;
__eq__, comparação da matriz do nó;
__lt__, comparação do custo do nó.

2.2 Função sucessora

A função sucessora a partir de um estado, gera todas as possibilidades de movimentação, movimentando o espaço vazio para: cima, direita, baixo e esquerda caso possível. Ela foi implementada na função “expand” que recebe um nó e retorna uma lista com os nós possíveis, já referenciando o nó de entrada como pai. A última parte é útil para refazer o caminho da solução.

2.3 Pilha

Para a busca em profundidade iterativa foi usada uma pilha para manter a fronteira. Na pilha a inserção, remoção e checagem de pilha vazia são O(1). Checar se um elemento específico está na pilha é O(n). Também é implementada uma função de adicionar vários elementos que é O(n) na quantidade de elementos adicionados.

2.4 Fila

Para a busca em largura foi usada uma fila para manter a fronteira. Na fila a inserção, remoção e checagem de fila vazia são O(1). Checar se um elemento específico está na pilha é O(n). Também é implementada uma função de adicionar vários elementos que é O(n) na quantidade de elementos adicionados.

2.5 Conjunto

Para todos os algoritmos que é necessário manter alguma noção de nós expandidos para verificação foi utilizado conjunto ou Set. Sua complexidade de inserção, e checagem se elemento pertence é O(1) o que é bem útil para o uso desejado.

2.6 Heap

Para alguns dos algoritmos que é necessário expandir com base no menor custo, UCS, greedy e A* foi usado heap. Ele é útil pois mantém a retirada do menor elemento O(log n) com custo de inserção também O(log n).

3. Algoritmos implementados

3.1 Busca em largura (BFS)

A busca em largura usa uma fila para a fronteira e um set para o conjunto de nós expandidos. Por ser FIFO a fila faz com que a ordem dos nós retirados seja a mesma da que foram colocados fazendo a expansão na ordem da busca em largura. Após o primeiro nó ser retirado da fila é feita a verificação se ele não foi expandido, se não ele é expandido e é verificado se ele não é a solução. Assim ele é expandido e seus filhos que não foram expandidos são adicionados à fila. Esse processo se repete até encontrar a solução.

O algoritmo é completo, pois não entra em loops. O algoritmo é ótimo nesse caso pois o custo é não decrescente, no caso, o custo de cada movimentação é 1.
Considerando d a profundidade da solução, a complexidade de tempo no pior caso é O(4^d) já que cada nó tem no máximo 4 filhos, apesar disso a maioria dos nós tem 3 ou 2 filhos. A complexidade de espaço também é O(4^d).

3.2 Busca de Aprofundamento Iterativo (IDS)

A busca em aprofundamento iterativo faz buscas em profundidade com profundidade aumentando em 1 a cada iteração. na busca em profundidade é usada uma Pilha onde os nós são colocados, retirados e expandidos em ordem LIFO. Como o último a ser colocado é o primeiro a sair é feita a busca na ordem desejada.

Para que o algoritmo seja completo, não entre em loops, é necessário tratá-los. No caso, cada nó na fronteira armazena seus uma lista com seus pais para fazer a verificação de loop. Isso tem um custo de espaço considerável, mas como a solução é ótima e a profundidade máxima não faria ter muitos nós na fronteira foi decidido assim.

O algoritmo é completo, devido ao tratamento de loops. ele é ótimo já que o custo é não decrescente, custo é sempre 1.

A complexidade de tempo é exponencial, mas a de espaço é linear nesse caso considerando que há um limite para o custo de manter a lista de pais nos nós da fronteira.

3.3 Busca de custo uniforme (UCS)

A busca de custo uniforme usa um heap para a fronteira, um set para os nós expandidos e um dicionário para armazenar o menor custo atual de cada nó. O heap armazena os nós em ordem de menor custo que é exatamente o que é necessário nessa busca, no caso o custo é a profundidade do nó na árvore, ou a profundidade do nó anterior mais 1, já que o custo de cada transição da árvore é 1.

Um nó é retirado do heap e expandido se já não tiver sido, é checado se ele é a solução, se não for, ele é expandido. Seus filhos são adicionados ao heap se já não foram expandidos e se não forem piores que nós iguais que já estão no heap, se forem adicionados seu custo no dicionário é alterado ou definido.

O algoritmo é completo, pois não entra em loops já que todos os custos são maiores ou iguais a um limite, no caso 1. Ele é ótimo já que segue o menor custo.

Considerando d a profundidade (que também é o custo) da solução, a complexidade de tempo no pior caso é O(4^(1+d)) já que cada nó tem no máximo 4 filhos, apesar disso a maioria dos nós tem 3 ou 2 filhos. A complexidade de espaço também é O(4^(1+d)).

3.4 Busca Gulosa

Como a busca gulosa ensinada na aula não é completa, foi feita uma pequena alteração que checa se os nós já foram expandidos antes de expandi-los ou de adicioná-los à fronteira.

A busca gulosa usa um heap para a fronteira e um set para os nós expandidos. O heap armazena os nós em ordem de menor custo que é exatamente o que é necessário nessa busca, no caso o custo é o que é estimado pela heurística.

Um nó é retirado do heap e expandido se já não tiver sido, é checado se ele é a solução, se não for, ele é expandido. Seus filhos são adicionados ao heap se já não foram expandidos.

O algoritmo implementado é completo e não é ótimo.

Considerando d a profundidade da solução, a complexidade de tempo e espaço no pior caso é O(4^d) já que cada nó tem no máximo 4 filhos, apesar disso a maioria dos nós tem 3 ou 2 filhos.

3.5 Busca A*

A busca A* tem a mesma implementação da de custo uniforme com exceção do custo ser a profundidade do nó somado ao valor da heurística para o mesmo.

O algoritmo é completo, pois não entra em loops já que todos os custos são maiores ou iguais a um limite, no caso 1. Ele é ótimo já que, como será visto, as heurísticas usadas são admissíveis. Além disso, dentro os ótimos, ele expande o menor número de nós.

Quanto à complexidade de tempo e espaço ele é exponencial no pior caso dependendo da heurística.

3.6 Hill Climbing

No Hill Climbing a partir do nó de partida é checado se ele é a solução, o melhor vizinho é escolhido utilizando a heurística de Distância de Manhattan:

- se ele for melhor, o processo se repete a partir dele;
- se ele for igual, o movimento é permitido mas é contado, se não houver melhora em k repetições ele para;
- se for pior, ele para e retorna o nó atual que é um ótimo local.

O algoritmo é completo e não é ótimo.

A complexidade de tempo é O(k) considerando que a solução nunca está a mais de 31 passos a complexidade de espaço é O(1) já que só precisa armazenar o nó atual e os vizinhos

4. Heurísticas utilizadas

As heurísticas utilizadas consistem em relaxamento das regras de movimentação das peças que no 8-puzzle são uma peça pode ser movida de A para B se:

1) A e B são adjacentes;
2) B está vazia

4.1 Fora de posição (h1)

Essa heurística conta o número de elementos que estão fora de sua posição final, ela é obtida relaxando ambas as regras de movimentação.

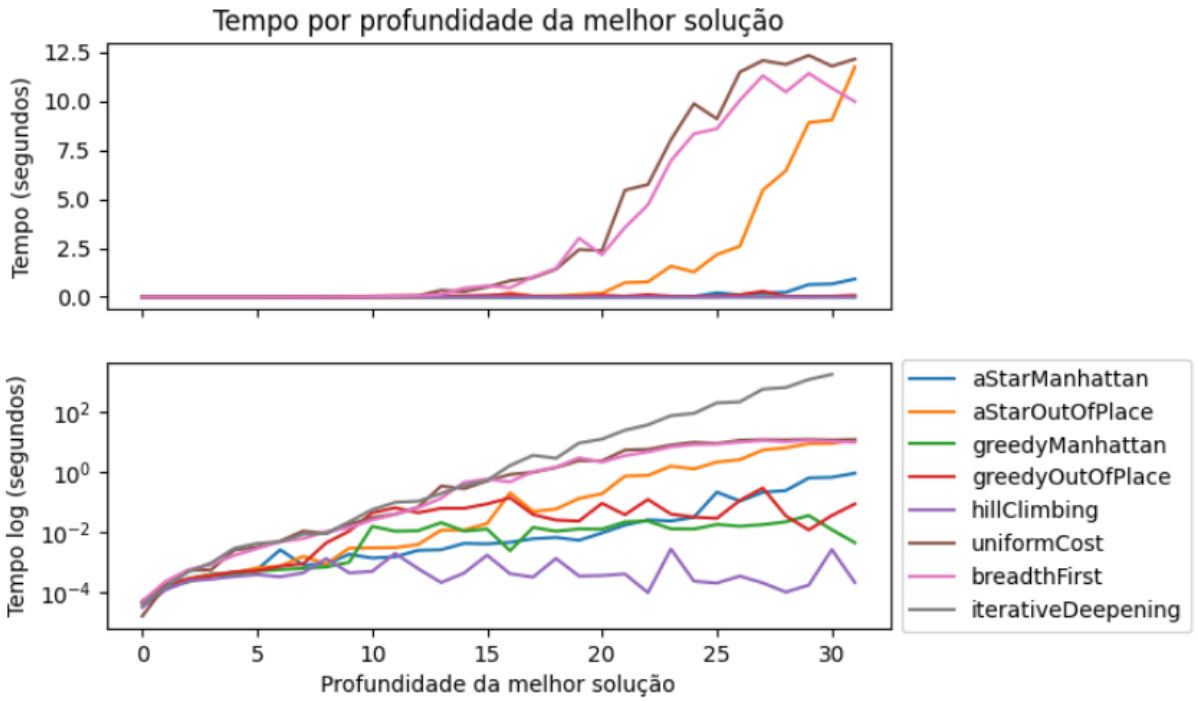
Ela é admissível porque cada movimento move apenas uma peça, sendo necessário pelo menos um movimento para colocar uma peça em sua posição final. Dessa forma, se n peças estão fora de posição o custo até a solução ótima não pode ser menor que n.

4.2 Distância de Manhattan (h2)

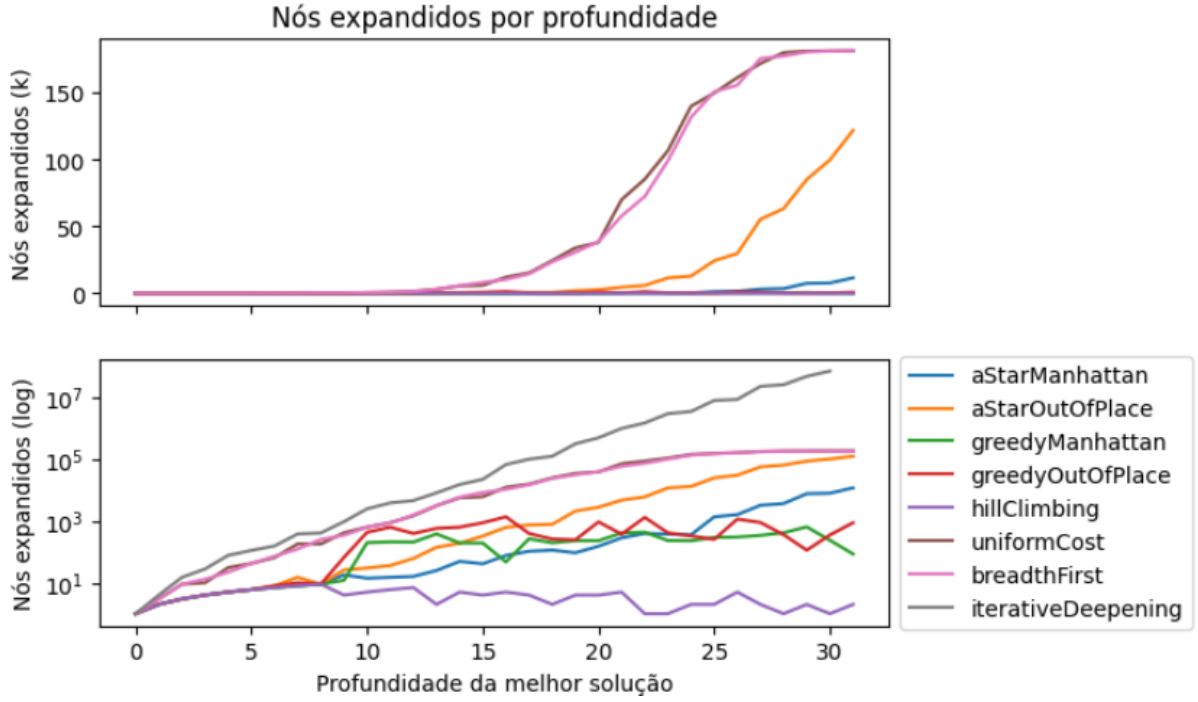
Essa heurística soma o número de movimentos necessários para colocar cada peça em sua posição final, movendo em apenas casas adjacentes, ela é obtida relaxando a regra de B está vazia.

Ela é admissível porque cada movimento move apenas uma peça em uma posição. Sendo n a quantidade de movimentos, sem considerar a regra de casas vazias, necessária para colocar a peça em sua posição final, uma peça precisa ser movida pelo menos n vezes para estar em sua posição correta. Dessa forma, se n é a soma das distâncias entre peças e suas posições finais o custo até a solução ótima não pode ser menor que n.

5. Análise dos resultados



Como é possível ver nos gráficos acima, o pior em tempo é o IDS (ele está apenas no gráfico em escala log pois demora consideravelmente mais). Isso se deve provavelmente à expansão de nós repetidos em relação aos outros, foi visto que o efeito disso não é tão relevante pois sempre há mais folhas que pais. Mas nesse caso parece ter afetado negativamente o desempenho.



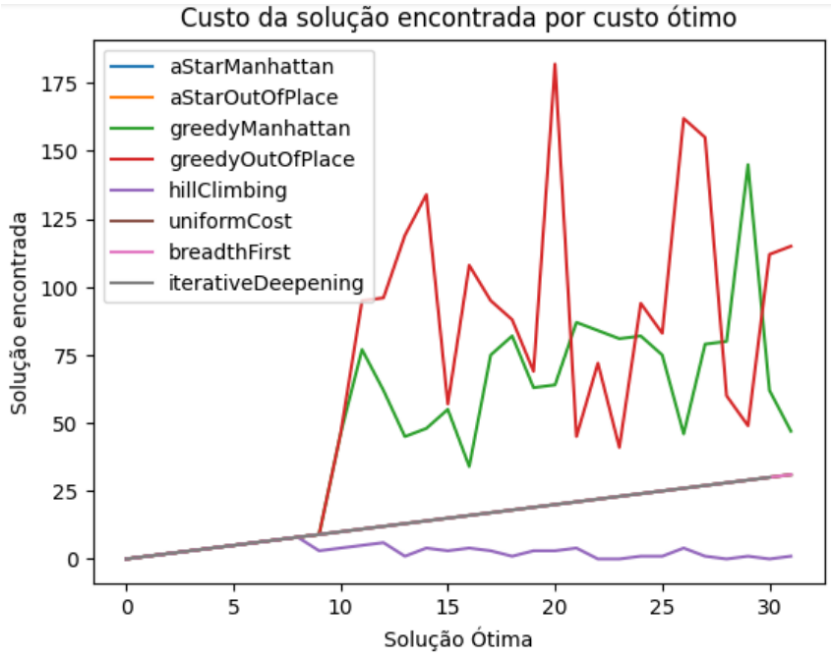
Como esperado, assim como no tempo, o IDS também expande uma quantidade de nós significativamente maior que os outros algoritmos utilizados.

Observando os quatro gráficos acima é possível ver que o uniform cost e o breadth first são quase idênticos em tempo e quantidade de nós expandidos. Como o custo é sempre 1, ambos os algoritmos expandem primeiro nós na profundidade 1 depois na 2, e assim em diante. As diferenças podem ser explicadas pela ordem que os nós de cada profundidade é expandida.

Outra análise interessante é que o A* com a heurística Out of Place (h1) desempenha pior que o com a heurística da distância de Manhattan (h2). Isso se deve à heurística h1 ser dominada por h2. Como para cada elemento fora de posição a heurística h2 soma pelo menos 1 na distância h2 é maior ou igual a h1. Por isso, há um guia melhor para a busca.

Um comportamento visto em aula interessante de ser observado é que o A* é o algoritmo ótimo que menos expande nós e é eficientemente ótimo.

O comportamento de tempo observado no A* com a heurística h1 provavelmente se deve à ela não ser muito boa.



Quanto aos algoritmos de busca gulosa, com ambas as heurísticas, e hill climbing. Eles são consideravelmente mais rápidos que os outros, mas muitas vezes deixam de encontrar soluções ótimas para os problemas. Como é possível ver no gráfico acima, a busca gulosa com h1 e com h2 não encontrou mais nenhuma solução ótima quando ela teve profundidade maior que 10. O hill-climbing por sua vez sequer encontrou uma solução ótima quando ela tinha custo maior que 10 e ficou preso em um ótimo local.

Os algoritmos ótimos podem ser vistos em uma linha reta onde a solução encontrada é a de custo ótimo.

6. Exemplos de soluções encontradas

```
>python TP1 H 8 0 7 5 4 6 1 3 2 PRINT
1

8   7
5 4 6
1 3 2

8 7
5 4 6
1 3 2
```

Nesse caso o Hill Climbing não encontra a solução e para em um ótimo local.

```
>python TP1 A 8 0 7 5 4 6 1 3 2
27
```

Nesse caso o A* encontra a solução ótima e a imprime

7. Conclusões

Conforme visto na discussão dos resultados, foi possível ver na prática como os algoritmos estudados em sala funcionam. Também foi possível ver as vantagens e desvantagens dos mesmos na prática.

Um conhecimento que foi bastante exercitado foi o de estruturas de dados, pequenas mudanças na implementação como mudar de list para set mudam drasticamente o tempo de execução dos algoritmos.