Trabalho Prático 1: 8-Puzzle

Introdução à Inteligência Artificial

Leonel Mota Sampaio Durão - 2019006876

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG - Brasil

leonelmota5@ufmg.br

1. Introdução

O trabalho em questão envolve a implementação de soluções para o problema 8-Puzzle utilizando diferentes métodos de busca. Foram implementados métodos de busca de várias classes diferentes: busca sem informação (*Breadth-First Search, Iterative Deepening Search e Uniform Cost Search*), busca com informação (*A* Search, Greedy Best-First Search*), neste caso foram utilizadas duas heurísticas distintas discutidas mais adiante, e busca local (*Hill Climbing*). O programa inteiro foi implementado utilizando a linguagem C++ e a sua *Standard Template Library.* Todos os métodos foram testados com as entradas distribuídas com o trabalho e os resultados foram comparados e serão analisados com auxílio de gráficos no decorrer do relatório.

2. Estruturas de Dados

O problema inteiro foi modelado através de classes. Como a busca de estados para a resolução do problema podem ser vistas como uma busca de uma árvore, foi criada uma classe *Node* que representa cada um dos nós dessa árvore. Essa estrutura encapsula o estado do problema, ou seja, a configuração dos elementos no tabuleiro, que é armazenado como um *vector<int>* de tamanho 9 que representa um tabuleiro de forma linear. Além disso, a classe *Node* armazena um ponteiro para o seu nó pai, aquele que o chamou na árvore de busca e um vetor de ponteiros para seus nós filhos, aqueles que ele pode chamar na árvore de busca. A classe também possui métodos que calculam se o estado que ele representa é um estado final, que calcula recursivamente os seus pais até o nó raiz e que calculam os seus filhos. A função sucessora, que calcula os filhos, cria novos nós que possuem os estados que são possíveis de serem atingidos a partir do nó atual com as ações. As ações são trocar o elemento de valor zero de lugar com o elemento à esquerda, direita, cima e baixo, contudo, não é possível fazer essa troca caso a direção leva para fora do tabuleiro.

Além disso, foi criado uma classe *Solver* que encapsula todos os métodos de busca. A função principal cria uma instância de Solver e passa como parâmetro a configuração inicial e o algoritmo desejado para resolver o problema.

3. Algoritmos

3.1 Breadth First Search

O algoritmo faz o caminhamento da árvore de busca camada por camada da árvore, priorizando os nós com menor profundidade. Ou seja, primeiramente são visitados os nós vizinhos do nó raíz, então todos os vizinhos nesses e assim sucessivamente. Para evitar possíveis ciclos no espaço de busca, é mantido um *set* de todos os nós já visitados, então antes de visitar algum nó o programa faz uma checagem na estrutura se ele já está lá. Para a ordenação dos nós para serem visitados, o algoritmo se utiliza de uma *priority queue* ordenada pelos nós com menor profundidade. A vantagem da estrutura de dados *set* é a eficiência na inserção e checagem se algum elemento já se encontra lá, mesmo para grandes quantidades de elementos, operações $O(\log n)$. Uma vantagem desse algoritmo é a sua completude e otimalidade, ele sempre encontra uma solução e essa solução é ótima. Sua desvantagem é seu maior tempo de computação na média em relação à métodos de busca com informação, já que ele sempre irá expandir todos os nós até a profundidade da solução.

3.2 Iterative Deepening Search

Esse método funciona realizando diversas *Depth First Search* limitadas à uma certa profundidade e aumentamos esse limite iterativamente até acharmos o resultado. Foi implementado de forma recursiva pela sua simplicidade de implementação mas também poderia ser facilmente implementada de forma similar à *BFS* porém utilizando uma *stack* como estrutura de dados. Esse método possui as vantagens de uma *BFS* por ser completo e ótimo e a vantagem de uma *DFS* de possuir complexidade linear à profundidade em espaço. Além disso, a complexidade assintótica de tempo não é maior que a de uma *BFS*, já que os nós mais profundos serão computados poucas vezes e são muito menos numerosos. Contudo, seu tempo de computação na prática é muito grande e nem completou a execução para alguns casos maiores de teste.

3.3 Uniform Cost Search

Outro método de busca sem informação, esse algoritmo é mais conhecido por Dijkstra. Nesse algoritmo, o próximo nó a ser visitado é, dentro os visinhos dos nós

já visitados, aquele que possui menor custo. Isso garante que o menor caminho para todos os nós partindo da raiz é encontrado, então a solução é sempre ótima. Para o problema o 8-Puzzle, todas as arestas possuem custo unitário, não existe movimentação mais custosa que outras. Então o custo de um nó é sua profundidade, fazendo o algoritmo funcionar da mesma forma que a *BFS*, possuindo as mesmas vantagens e desvantagens. De forma similar, foi utilizado como estrutura de dados uma *priority queue*.

3.4 A*

Esse método pertence à classe de busca com informação, em que usamos uma heurística para melhorar o tempo esperado de processamento. Funciona de forma muito similar ao Dijkstra, explorando primeiramente o nó com menor custo dentre os já visitados. A diferença entre os métodos é a forma de computar o custo, o Dijkstra se utiliza apenas o custo até chegar ao ponto, o A* utiliza este custo adicionado à uma estimativa do custo até chegar ao estado objetivo. Esse algoritmo é completo e ótimo e também possui uma complexidade de tempo exponencial no pior caso. Contudo, o número de nós visitados e o tempo de execução na prática são muito menores desde que uma boa heurística seja escolhida.

3.5 Greedy Best First Search

O *Greedy Best First Search* é outro algoritmo de busca com informação. Também funciona de forma similar ao Dijkstra porém com outra forma de calcular o custo. Esse método utiliza apenas a estimativa do custo para se chegar ao estado objetivo, não levando em conta o custo para se chegar ao nó atual. Mantendo uma estrutura para evitar estados repetidos, esse algoritmo é completo mas não é ótimo, podendo até mesmo ser muito distante do ótimo. A vantagem desse método é que seu tempo de execução é menor ainda que o A*, especialmente para casos maiores, apesar de também ser exponencial no pior caso.

3.6 Hill Climbing

Já o Hill Climbing é um algoritmo de busca local. Algoritmos de busca local funcionam “caminhando” para estados próximos do estado atual que esperam maximizar ou minimizar uma função de custo. Para a implementação do trabalho do algoritmo Hill Climbing, o próximo estado é o estado com menor custo vizinho ao estado atual, desde que o custo não seja maior. Ele pode se mover para no máximo *k* vizinhos com o mesmo custo que o estado atual, para que não fique preso em um loop infinito. Nas experimentações, foi averiguado que para o problema em questão o número máximo de movimentos laterais não interferiu no resultado final. Esse algoritmo não é completo e fica preso um grande número de vezes em um mínimo local. Uma das suas vantagens é que o número de nós expandidos tende a se

manter muito baixo e nos casos em que encontrou uma solução, essa solução foi ótima, mas isso não é garantido.

4. Heurísticas

Para a execução dos algoritmos de busca local e busca com informação, é necessário funções que aproximam o custo de se chegar de um dado estado para o estado final. Foi requisitada a implementação de duas funções desse tipo, ou heurísticas, e elas foram o número de elementos fora do lugar e a distância de Manhattan de cada elemento para seu lugar correto.

Para que a solução do método A* seja ótima, é necessário que essas heurísticas sejam admissíveis. Uma heurística é admissível se o custo dela é sempre menor ou igual ao custo real para se chegar ao estado final. É possível ver que o número de elementos fora do lugar é sempre menor ao custo real porque para cada elemento fora do lugar, você terá que movê-lo ao menos uma vez. A heurística da distância de Manhattan também é admissível porque para chegar à sua posição final um elemento deve ser movido de forma ortogonal, porém nem sempre o elemento será movido diretamente até seu objetivo.

5. Exemplo de Soluções

Para a seguinte entrada: $[1\,0\,2\,8\,5\,3\,4\,7\,6]$ a saída para os algoritmos BFS, IDS, UCS, A* foi:

```
9

1 2
8 5 3
4 7 6

1 5 2
8   3
4 7 6

1 5 2
  8 3
4 7 6

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

Para o Greedy Best First Search a solução encontrada foi:

73 1 2 8 5 3 4 7 6 1 2 8 5 3 4 7 6 1 2 3 8 5 4 7 6 1 2 3 8 5 6 4 7 1 2 3 8 5 6 4 7 1 2 3 8 6 4 5 7 1 2 3 8 6 4 5 7 1 2 3 8 6 7 4 5 1 2 3 8 6 7 4 5 1 2 3 8 7 4 6 5 1 2 3 8 7 4 6 5 1 2 3 4 8 7 6 5 1 2 3 4 8 7 6 5	1 2 3 4 7 6 8 5 1 2 3 4 7 6 8 5 1 2 3 6 4 7 8 5 1 2 3 6 4 7 8 5 1 2 3 6 7 8 4 5 1 2 3 6 7 8 4 5 1 2 3 6 7 5 8 4 1 2 3 6 7 5 8 4 1 2 3 6 7 5 8 4 1 2 3 7 5 6 8 4 1 2 3 7 5 6 8 4 1 2 3 7 5 4 6 8	1 2 3 7 5 4 6 8 1 2 3 7 5 4 6 8 1 2 3 5 4 7 6 8 1 2 3 5 4 7 6 8 1 2 3 5 6 4 7 8 1 2 3 5 6 4 7 8 1 2 3 5 6 7 8 4 1 2 3 5 6 7 8 4 1 2 3 7 5 6 8 4 1 2 3 7 5 6 8 4 1 2 3 7 6 8 5 4 1 2 3 7 6 8 5 4	1 2 3 7 6 4 8 5 1 2 3 7 6 4 8 5 1 2 3 7 6 4 8 5 1 2 3 6 4 7 8 5 1 2 3 6 8 4 7 5 1 2 3 6 8 4 7 5 1 2 3 8 4 6 7 5 1 2 3 8 7 4 6 5 1 2 3 8 7 4 6 5 1 2 3 8 7 6 5 4 1 2 3 8 7 6 5 4 1 2 3 8 5 7 6 4	1 2 3 8 5 7 6 4 1 2 3 8 5 6 4 7 1 2 3 8 5 6 4 7 1 2 3 8 4 5 6 7 1 2 3 8 4 5 6 7 1 2 3 4 5 8 6 7 1 2 3 4 5 8 6 7 1 2 3 4 5 7 8 6 1 2 3 4 5 7 8 6 1 2 3 5 7 4 8 6	1 2 3 5 7 4 8 6 1 2 3 5 7 6 4 8 1 2 3 5 7 6 4 8 1 2 3 5 6 4 7 8 1 2 3 5 6 4 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8

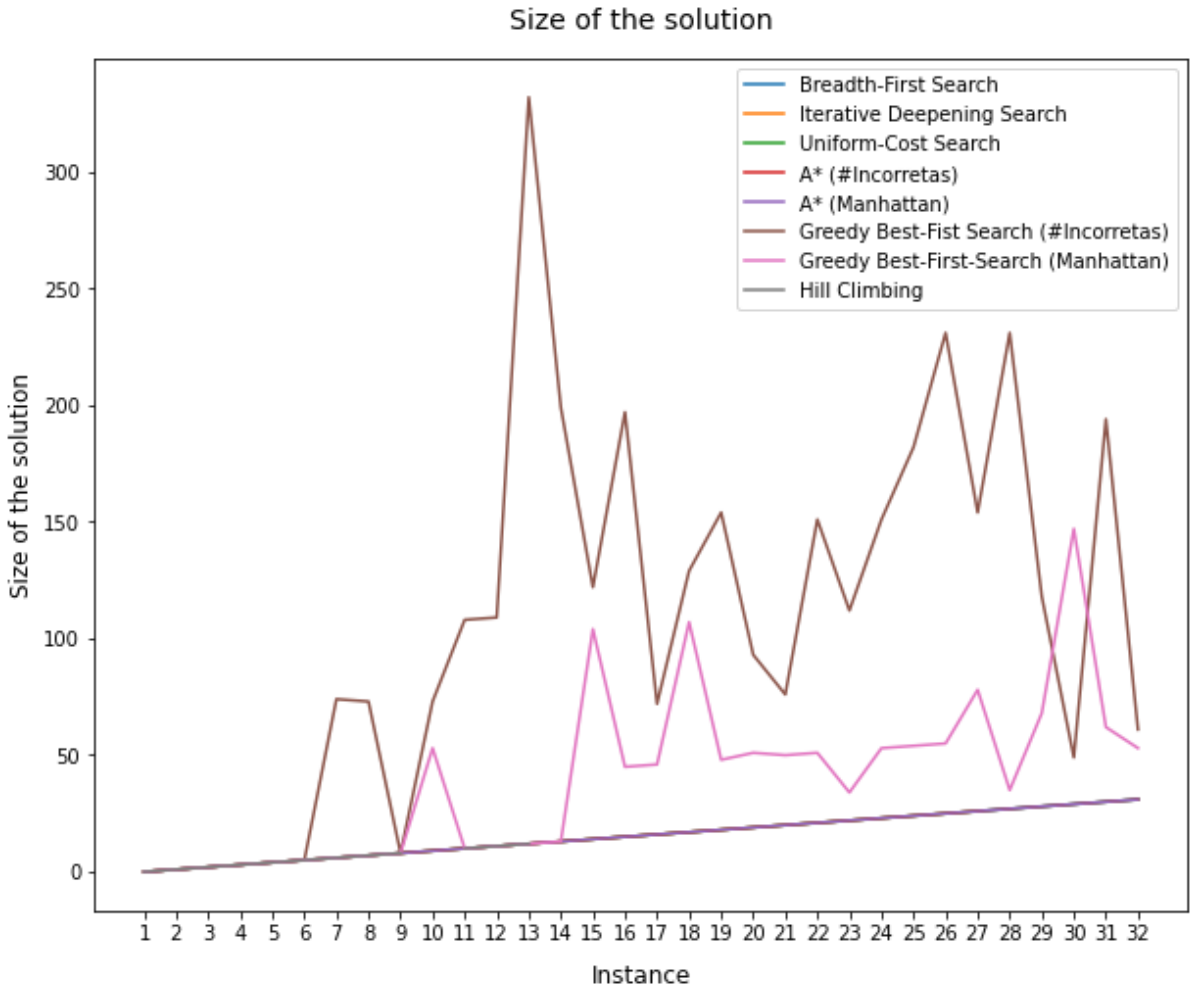
Já para o algoritmo Hill Climbing, uma solução não foi encontrada.

6. Análise

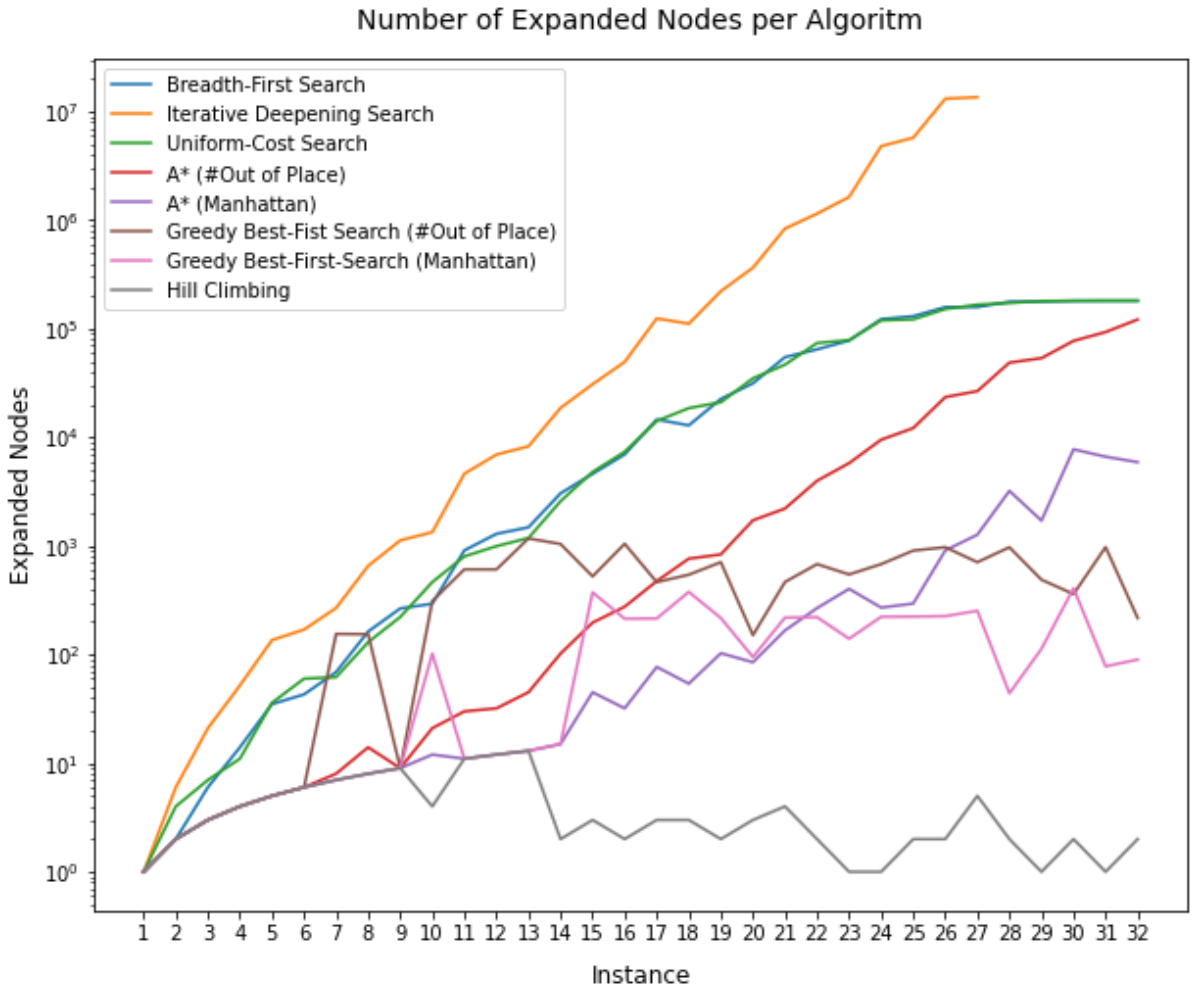
Para a análise dos resultados foram utilizados os exemplos 32 de entrada fornecidos na especificação com todos os métodos implementados, ordenados de acordo com sua distância ótima até a solução. Os algoritmos são comparados de acordo com tempo de execução, número de nós expandidos e qualidade da resposta, além do número de respostas encontradas por cada algoritmo. A análise faz uso de gráficos para melhor visualizar os resultados.

Algorithm	Number of Solutions Found
A* Manhattan	32
A* Out of Place	32
BFS	32
Greedy Manhattan	32
Greedy Out of Place	32
UCS	32
IDS	27
HC	12

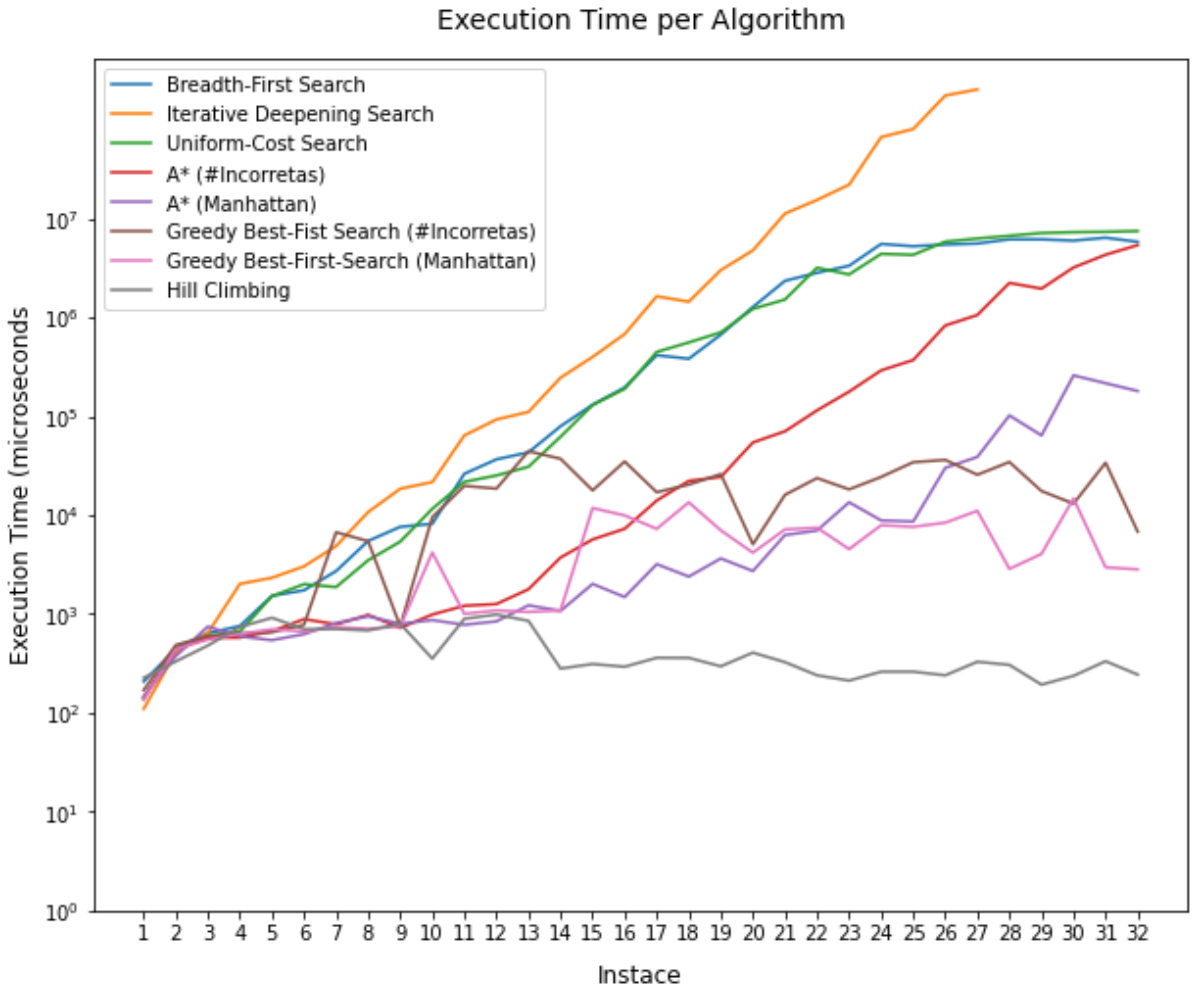
Podemos ver que o Hill Climbing não é um método muito bom para o problema em questão, possivelmente por possuir muitos mínimos locais. Já o IDS não consegue terminar a execução em instâncias mais difíceis.



Nessa figura podemos ver que todos os algoritmos encontram respostas ótimas, com exceção do Greedy Best First Search, com as duas heurísticas distintas.



Para facilitar a interpretação dos resultados, o gráfico está em escala logarítmica, que é ideal para crescimentos exponenciais. Com exceção do Hill Climbing, todos crescem bastante com a distância até a solução da configuração inicial, mesmo para um problema bastante pequeno. Contudo, os algoritmos de busca com informação se expandem algumas ordens de grandeza a menos que os de busca sem informação para instâncias mais difíceis, menos para o A* com heurística de número de incorretas. É possível ver que em todos os casos a heurística de Manhattan foi melhor que a de número de incorretas, mostrando a importância da escolha de uma boa heurística. O maior crescimento do IDS é esperado, e em alguns casos ele nem consegue terminar a execução.



O gráfico de tempo é bastante parecido com o de nós expandidos, já que quanto maior o números de nós a visitar, mais tempo é gasto executando.