Trabalho Prático 1

8-Puzzle

Inteligência Artificial - 2019/01
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Deiziane Natani da Silva
2015121980

---

1. Introdução

Este trabalho compara o desempenho de métodos de busca sem informação, com informação e busca local na abordagem da solução de um *N-Puzzle* de tamanho 8, em um tabuleiro de tamanho 3x3. O quebra-cabeça é um caso simples, mas desafiador e ideal para demonstrar conceitos de busca em Inteligência Artificial. Ele é um jogo de tabuleiro de blocos deslizáveis que envolve uma dimensão definida como N linhas e M colunas - 3 linhas e 3 colunas para um *8-puzzle*. O objetivo do jogo é mover as peças a partir de um estado inicial até seu estado final, quando o tabuleiro está ordenado de forma crescente.

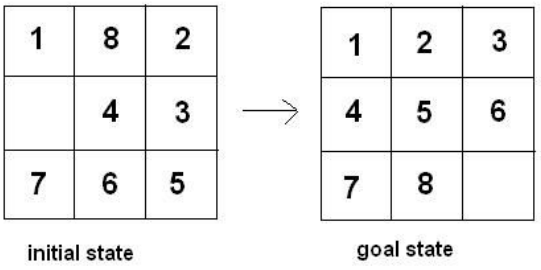


Fig. 1 - Tabuleiro *8-Puzzle*

As regras do jogo são simples, a peça vazia é a única que pode movimentar-se, sendo possível de dois a quatro movimentos: para cima, baixo, direita e esquerda. Esses movimentos geram novos estados até que o estado final seja encontrado - existem também casos em que a solução final é impossível de ser alcançada. A solução ótima para este problema pertence à classe NP-Completo. (RUSSELL; NORVIG, 2003)

Os algoritmos utilizados são *Breadth First Search* (Busca em Largura), *Iterative Deepening Search* (Busca de Aprofundamento Iterativo), *Uniform Cost Search* (Busca de Custo Uniforme), *A* Search*, *Greedy Best First Search* e *Hill Climbing Search*.

2. Modelagem

O programa foi modelado em 3 arquivos: *board.py*, *search.py* e *8-puzzle.py*. O primeiro possui uma classe *Board* é usado para implementar o tabuleiro 3x3, com todas as ações que podem ser aplicadas a este tabuleiro. No *search* são implementados todos os algoritmos de busca citados na introdução, sendo cada um uma função. O *8-puzzle* é onde está o programa principal, usado para executar os algoritmos e imprimir os resultados. Para o desenvolvimento do programa foi utilizado Python 2.

3. Algoritmos e Heurísticas

3.1. Busca sem Informação

As buscas sem informação utilizam somente a informação disponível na formulação do problema. Dessa forma, as estratégias de busca diferenciam-se apenas pela ordem em que expandem os nós da árvore de busca.

3.1.1. Breadth First Search

O *Breadth First Search* é um algoritmo do tipo sem informação, por isso não inicia o conhecimento total de todo o espaço de buscas. Em vez disso, ele constrói sua própria memória, lembrando todos os nós pelos quais passa, marcando-os como explorados. Além disso, a única maneira de o BFS saber quando parar é finalmente chegar a um nó que tenha o mesmo estado do Nó Meta (*Goal*). O BFS atravessa a árvore de pesquisa descobrindo a fronteira um nível por vez.

Para armazenar os nós da fronteira é utilizada uma fila FIFO (*First-In, First-Out*). Assim, novos nós (que são sempre mais profundos que seus pais) vão para a parte de trás da fila, e os nós antigos, que são mais rasos que os novos nós, são expandidos primeiro. O algoritmo é completo, sempre encontra uma solução mais rasa - que não necessariamente é a ótima, apenas se todas as ações têm custos iguais. Por armazenar os nós, ele tem um alto custo em memória, que é maior do que o tempo de execução.

3.1.2. Iterative Deepening Search

A Busca de Aprofundamento Iterativo é basicamente uma mistura da busca em profundidade e busca em largura. É similar à busca em largura, pois explora um nível de nós em cada iteração, porém mais eficiente em tempo e espaço, é completa quando o fator de ramificação (*branching factor*) é finito e quando o custo do caminho é uma função não decrescente da profundidade do nó.

Ela, assim como a busca em profundidade, ela possui custo de memória reduzido.. Encontra o melhor limite de profundidade ℓ aumentando-o gradualmente até encontrar um objetivo que acontece quando alcançar *d*, profundidade do nó objetivo mais raso.

3.1.3. Uniform Cost Search

O BFS pode não encontrar a solução ótima se as ações têm custos diferentes. Nesse caso o Uniform Cost Search deve ser usado para garantir otimalidade. Ao contrário do BFS que usa apenas uma FIFO, o UCS usa uma fila de prioridade, onde os nós de menor custo g(n) são expandidos primeiro. Além disso, a UCS pode utilizar ainda mais memória que a BFS. Alterações em relação à busca em largura:

- O teste de objetivo é feito quando um nó é selecionado para expansão (e não quando é criado)
- Se o nó já está na fronteira, mesmo assim é necessário verificar se o caminho encontrado é mais barato que aquele guardado

Primeiro, observamos que, sempre que a busca de custo uniforme seleciona um nó n para expansão, o caminho ideal para esse nó foi encontrado. Então, como os custos da etapa não são negativos, os caminhos nunca ficam mais curtos à medida que os nós são adicionados. Estes dois fatos juntos implicam que a busca de custo uniforme expande os nós na ordem de seu custo ideal de caminho. Portanto, o primeiro nó de meta selecionado para expansão deve ser a solução ótima.

3.2. Busca com Informação

A Busca com Informação utiliza conhecimento do problema para guiar a busca. Esse conhecimento está além da própria definição do problema, como, por exemplo, o estado inicial, modelo de transição (função sucessora), custo de ação, etc. Dessa forma, podem encontrar soluções de forma mais eficiente do que as buscas sem informação.

3.2.1. Heurísticas

Cada problema exige uma função heurística diferente. Em cada uma, não se deve superestimar o custo real da solução. No problema foram utilizadas duas heurísticas: distância de Manhattan e *Misplaced Tiles*.

A primeira calcula o custo baseado na soma das distâncias das peças de suas posições finais. Como os ladrilhos não podem se mover ao longo das diagonais, a distância que calculada é a soma das distâncias horizontal e vertical. A heurística é admissível, porque tudo o que qualquer

movimento pode fazer é mover um ladrilho um passo mais perto do *goal*. A segunda calcula o custo baseado na quantidade de ladrilhos fora da posição correta. Ela também é admissível porque fica claro que qualquer ladrilho que esteja fora do lugar deve ser movido pelo menos uma vez.

3.2.2. A* Search

A ideia por trás do algoritmo é basicamente podar caminhos que são caros. Se baseia numa função de avaliação: f(n) = g(n) + h(n), onde g(n) = custo para chegar ao nó n e h(n) é custo estimado (através de heurística) para ir de n até o objetivo. A estratégia é completa e a busca é ótima se h(n) for heurística admissível.

3.2.3. Greedy Best First Search

O algoritmo de Melhor Escolha Guloso seleciona o nó a ser expandido utilizando uma função de avaliação denominada `f(n)` `onde` f(n) é uma função de custo: f(n) = h(n) - onde h(n) é o resultado de uma heurística - então o nó que apresentar menor f(n) é expandido primeiro. A implementação é idêntica ao da Busca de Custo Uniforme substituindo-se g(n) por f(n). O algoritmo é completo se os nós são finitos porém pode não encontrar a solução ótima.

3.3. Busca Local

Em muitos problemas de otimização o caminho ao objetivo é irrelevante só interessa o estado objetivo, ou seja, a solução. Nestes casos, podem ser usados algoritmos de busca local. Eles também usam pouca memória (normalmente, uma quantidade constante e podem encontrar soluções razoáveis/factíveis em espaços de estados grandes ou infinitos (e também contínuos).

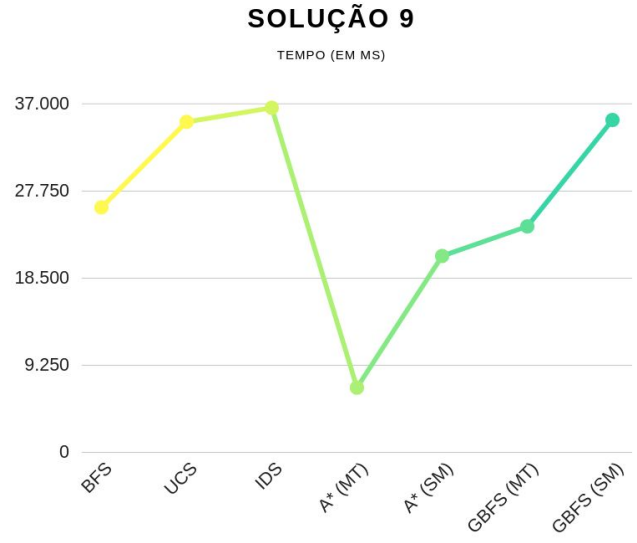
3.3.1. Hill Climbing Search

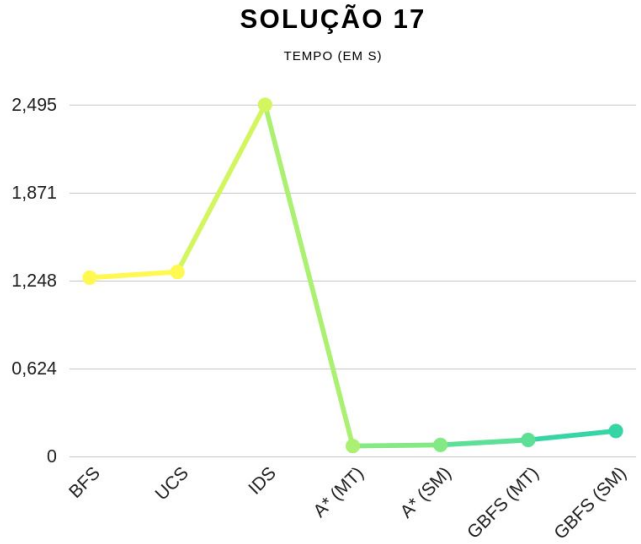
Como a *Greedy Best First Search* só baseia suas decisões na Função Heurística h(n), *Hill Climbing* funciona da mesma maneira, mas desconsidera totalmente a memória dos nós explorados. Portanto, ele percorre a Árvore de Pesquisa selecionando o sucessor com o valor de heurística mais barato, sem reter a memória dos estados explorados.

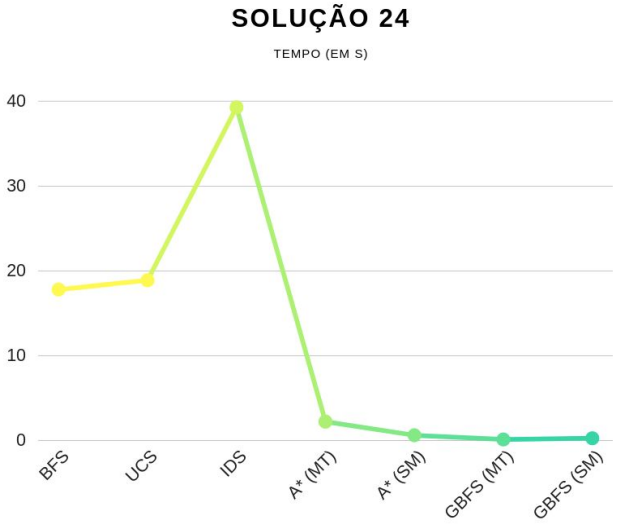
Isso garantirá que a técnica funcione com uso mínimo de memória, com o mínimo de computação possível, mas ainda mantendo a vantagem de um método de busca com informação. A desvantagem do *Hill Climbing* é que, devido à ausência de memória, há a possibilidade de repetir os mesmos estados e ficar preso em algum estado de máximos locais, platôs ou *shoulders*. Um platô é uma área plana da paisagem do espaço de estado. Ele pode ser um máximo local plano, a partir do qual não existe uma saída para cima, ou um *shoulder*, a partir do qual é possível progredir, porém são necessárias adaptações.

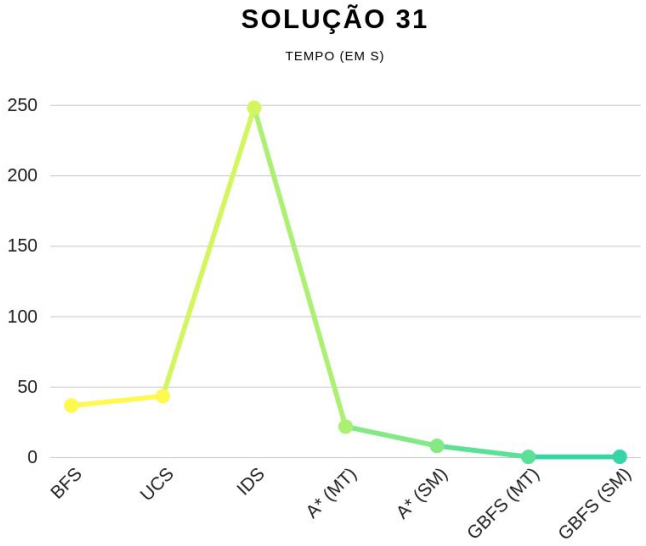
Para tentar resolver o problema dos *shoulders* o algoritmo implementado permite movimentos que não melhoram a solução - ou seja, tem custos iguais - até um limite *k*.

4. Análise dos Algoritmos e Soluções









Comparação de Iterações, Memória e Caminho

Tabela 1. Solução 09

	Movimentos	Iterações	Memória
BFS	9	393	254
UCS	9	459	288
IDS	9	882	5
A* (MT)	9	30	22
A* (SM)	9	22	20
GBFS (MT)	29	192	128
GBFS (SM)	31	67	46

Tabela 2. Solução 31

	Movimentos	Iterações	Memória
BFS	31	519.169	50
UCS	31	519.143	76
IDS	31	4.338.424	17
A* (MT)	31	173.880	42.684
A* (SM)	31	17.413	9.862
GBFS (MT)	161	1.222	755
GBFS (SM)	151	522	361

Análise do Hill Climbing

Tabela 3. Solução 04

	Movimentos	Iterações	Memória	Goal
Hill Climbing (MT)	4	5	4	✔
Hill Climbing (SM)	2	3	2	❌

Tabela 4. Solução 09

	Movimentos	Iterações	Memória	Goal
Hill Climbing (MT)	3	4	3	❌
Hill Climbing (SM)	3	4	3	❌

Tabela 5. Solução 17

	Movimentos	Iterações	Memória	Goal
Hill Climbing (MT)	10	11	10	❌
Hill Climbing (SM)	12	13	16	❌

Tabela 6. Solução 24

	Movimentos	Iterações	Memória	Goal
Hill Climbing (MT)	0	1	0	❌
Hill Climbing (SM)	10	11	10	❌

5. Conclusão

A partir dos resultados, parece que a técnica Hill Climbing consegue encontrar algumas soluções, mas ignora mais nós do que as outras buscas - o algoritmo só aceita soluções que melhorem a função heurística. Dessa forma, há uma grande chance de falha ao usar as heurísticas escolhidas (Misplaced Tiles e Sum of Manhattan) em casos de soluções mais longas ou problemas mais complexos. Há situações em que a solução é alcançada com uma heurística e com a outra não.

Com relação aos outros algoritmos, o A* é geralmente o melhor em tempo de execução, porém, isso depende também das heurísticas utilizadas. Os algoritmos de busca sem informação são mais lentos na maioria dos casos (principalmente o Iterative Deepening Search).

O trabalho foi de grande importância no entendimentos do conhecimento adquiridos até essa parte da disciplina. Pudemos ver o funcionamento do que foi aprendido e enfrentar os desafios encontrados em implementá-los. Foi possível

fixar melhor o conteúdo e compreendermos seus pormenores, que são de extrema importância.

6. Referências

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003.