TP1: 8 - Puzzle

Introdução a Inteligência Artificial

Aluna: Izabela Garcia Tavares Pinheiro Pérez

Matrícula: 2022076332

Informações Gerais

No desenvolvimento deste trabalho, o ambiente de desenvolvimento utilizado é o Linux Ubuntu 22.04.3 LTS, junto com a versão 3.10.12 do Python,

Introdução

O 8-Puzzle é um quebra-cabeça deslizante com o objetivo de reorganizar as peças de um tabuleiro 3x3 para que elas estejam dispostas em ordem crescente, da esquerda para a direita e de cima para baixo, com um espaço vazio representado pelo número 0. Para resolver esse quebra-cabeça, podemos aplicar diversos algoritmos de busca que exploram diferentes estratégias para encontrar a solução.

Neste documento, apresentaremos uma documentação para o trabalho de IA (Inteligência Artificial) que implementa a resolução do 8-Puzzle utilizando diferentes algorit mos de busca. Abordaremos a estrutura do código, as estruturas de dados usadas, as heurísticas adotadas e uma análise dos resultados obtidos.

Estrutura do Código

O código implementa a resolução do 8-Puzzle e utiliza as seguintes estruturas e componentes:

Estrutura do Tabuleiro

O tabuleiro do 8-Puzzle é representado por uma matriz 3x3, onde cada elemento representa uma célula do tabuleiro. O objetivo é chegar a um estado onde as células estejam ordenadas de acordo com a disposição final desejada.

```
GOAL = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]]
```

`GOAL` representa o estado final desejado do tabuleiro.

Para representar essa estrutura foi criada uma classe “Cell” que desempenha um papel crucial no contexto do trabalho, pois contém informações detalhadas sobre uma posição no tabuleiro do quebra-cabeça 8-Puzzle e todas as ramificações (movimentos) possíveis a partir do estado atual. Ela é usada para representar os estados do tabuleiro em diferentes momentos durante a resolução do quebra-cabeça.

Essa classe possui os seguintes atributos e métodos principais:

- `state`: Armazena o estado atual do tabuleiro.
- `parent`: Referência para o estado anterior do tabuleiro, representando a célula pai.
- `empty`: A posição do espaço vazio no tabuleiro.

- `branches`: Lista das possíveis ramificações (movimentos) a partir do estado atual.
- `depth`: A profundidade atual do estado no espaço de busca.
- `f`: Valor da função f(n), que é usado em algoritmos de busca informada, como A*.

Os principais métodos da classe incluem:

- `__setBranches()`**:** Calcula e define as possíveis ramificações (movimentos) a partir do estado atual, levando em consideração a posição do espaço vazio.
- `getBranches()`: Retorna a lista de ramificações possíveis a partir do estado atual, criando-as se ainda não estiverem definidas.
- `isCycle()`: Verifica se há um ciclo nas ramificações (usado no algoritmo IDS - Busca em Profundidade Iterativa).
- Sobrecarga de operadores como `__eq__`, `__hash__`, e `__lt__` para comparar objetos Cell.

Heurísticas

Foram implementadas duas heurísticas para auxiliar na busca por soluções eficientes:

1. **Heurística 1**: Conta o número de células fora do lugar em relação ao estado final.
2. **Heurística 2**: Calcula a soma das distâncias Manhattan de todas as células em relação à posição final desejada.

As heurísticas são admissíveis, o que significa que elas nunca superestimam o custo real para atingir o objetivo. Isso garante que os algoritmos de busca informada, como A*, funcionem corretamente.

Algoritmos de Busca

O código implementa os seguintes algoritmos de busca:

- Busca em Largura (BFS)
- Busca em Profundidade Iterativa (Iterative DFS)
- Busca de custo uniforme (Dijkstra)
- Busca Gulosa (Greedy BFS)
- A* (AStar)
- Subida de Encosta (Hill Climbing)

Dos quais as 3 primeiras são buscas sem informação, A* e Busca Gulosa são buscas com informação (heurística) e Hill Climbing é uma busca local.

Principais Diferenças Entre os Algoritmos

A principal diferença entre os algoritmos está na estratégia de busca adotada. Aqui estão as principais diferenças:

- **BFS (Busca em Largura)**: Explora todos os nós em uma camada antes de passar para a próxima camada. Garante encontrar a solução mais curta, mas pode ser lento em casos complexos.
- **Iterative DFS (Busca em Profundidade Iterativa)**: Começa com uma profundidade máxima e aumenta gradualmente. É completo e eficiente em termos de memória.
- **Dijkstra**: Encontra o caminho mais curto em um grafo com pesos não negativos. Garante a solução mais curta, mas pode ser lento em grafos grandes.
- **Busca Gulosa (Greedy BFS)**: Escolhe o nó com a menor heurística (custo estimado para a solução) e continua a partir dele. Pode não encontrar a solução ótima, mas é eficiente em termos de memória.

- **Subida de Encosta (Hill Climbing)**: Escolhe o vizinho com a melhor heurística e continua a partir dele. Pode ficar preso em ótimos locais locais.
- **A (AStar):** Combina o custo real (custo f) do início até o nó atual (custo g) e a heurística (custo h) para escolher o próximo nó. É completo e eficiente.

Heurísticas Utilizadas

As heurísticas adotadas no código são as seguintes:

1. **Heurística 1 (Contagem de Peças Fora do Lugar)**: Esta heurística conta o número de células fora do lugar em relação ao estado final desejado. Ela é admissível, pois nunca superestima o custo real para atingir o objetivo. Isso ocorre porque, em cada movimento, pelo menos uma célula está na posição correta ou se move para a posição correta, reduzindo o número de peças fora do lugar.
2. **Heurística 2 (Soma das Distâncias Manhattan)**: Esta heurística calcula a soma das distâncias Manhattan entre cada célula e sua posição final desejada. Ela também é admissível, pois não superestima o custo real. A distância Manhattan é a distância entre duas células em uma grade, calculada como a soma das diferenças absolutas entre suas coordenadas x e y.

Resultados Obtidos

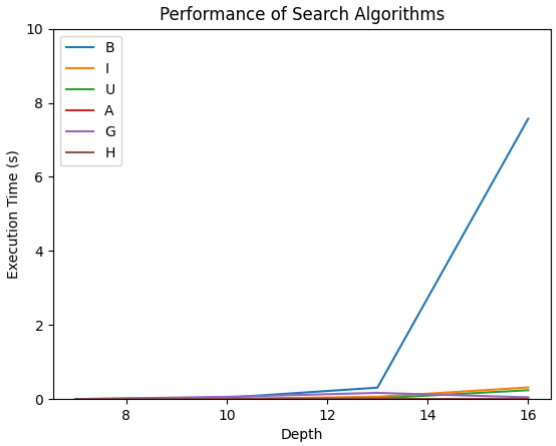
A análise dos resultados depende do algoritmo de busca escolhido. Os resultados podem variar dependendo da configuração inicial do tabuleiro. Alguns pontos a serem observados são:

- **BFS**: Garante encontrar a solução mais curta, mas pode ser lento para problemas complexos devido à expansão de muitos nós.
- **Iterative DFS**: Eficiente em termos de memória, mas pode levar mais tempo para encontrar a solução em comparação com o BFS.
- **Dijkstra**: Garante a solução mais curta, mas pode ser lento em grafos grandes devido ao cálculo de custo.
- **Busca Gulosa (Greedy BFS)**: Pode encontrar soluções rapidamente, mas não garante a solução mais curta pois segue a heurística de forma cega, o que reduz sua velocidade quando o caminho escolhido é muito maior que a solução ótima.
- **Subida de Encosta (Hill Climbing)**: Pode ficar preso em ótimos locais locais e não garantir a solução ótima.
- ***A (AStar)*:** Combina o custo real com a heurística e geralmente encontra soluções eficientes em termos de tempo e memória.

Os resultados podem ser influenciados pela configuração inicial do tabuleiro, pela heurística escolhida e pela estratégia de busca. Portanto, vamos analisar os gráficos abaixo para mostrar como esses parâmetros afetam a solução.

Gráfico de tempo por profundidade da resposta:

	BFS	IDS	UCS	A*	HC	GDFS
7	0.0020	0.0009	0.0012	0.0007	0	2.2411
10	0.0375	0.0111	0.0126	0.0005	0	0.0722
13	0.2843	0.0635	0.0226	0.0018	0	0.1548
16	6.9762	0.3140	0.2286	0.0079	0	0.0475



> O algoritmo de Hill Climbing não conseguiu encontrar nenhuma solução visto que ficou preso em mínimos locais em **todos** os testes

Os resultados da resolução do 8-puzzle com vários algoritmos de busca revelam que o A* é geralmente o mais eficiente para profundidades rasas, seguido pelo BFS, IDS e UCS. No entanto, à medida que a profundidade da solução aumenta, o tempo de execução de todos os algoritmos cresce significativamente. O algoritmo BFS tem um aumento muito alto no tempo de execução para respostas em profundidades maiores que 12, tornando-se muito ineficiente em comparação com os outros. Em contrapartida o A* manteve-se com um tempo de execução satisfatório na maioria das profundidades, sendo o melhor dos algoritmos para a resolução do problema.

Gráfico de tempo por profundidade (Heurística 1):

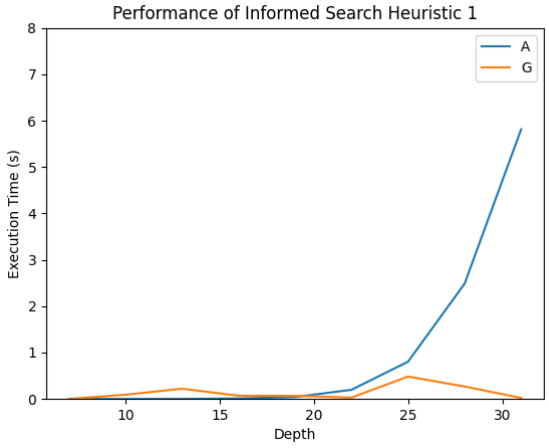
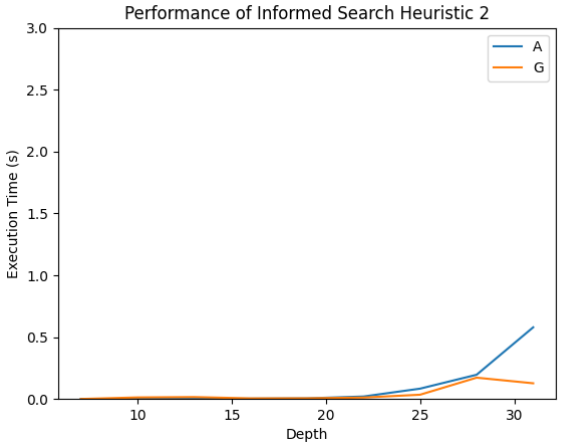


Gráfico de tempo por profundidade (Heurística 2)



O tempo de execução dos algoritmos A* e Greedy Best-First Search (GreedyBFS) foi significativamente melhor ao usar a heurística 2 em comparação com a heurística 1 devido às propriedades das heurísticas escolhidas.

Especificamente, o A* reduz significativamente o tempo de execução para soluções em profundidades maiores ao utilizar a heurística 2, uma vez que consegue manter um foco mais estreito nas partes do espaço de busca que são mais promissoras, o que é fundamental para melhorar o desempenho em quebra-cabeças complexos como o 8-puzzle. Em contraste, a heurística 1 fornece menos informações e pode levar a um maior número de expansões de estados, tornando-o menos eficaz em profundidades maiores.

Conclusão

Neste trabalho, abordamos a resolução do 8-Puzzle utilizando diversos algoritmos de busca e heurísticas. Avaliamos o desempenho dos algoritmos A*, BFS, IDS, Dijkstra, Hill Climbing e Greedy Best-First Search. Constatamos que o A* se destacou, especialmente ao empregar a heurística 2 (distância de Manhattan), reduzindo significativamente o tempo de execução, tornando-se uma escolha eficaz para soluções em profundidades maiores dado que é completo e ótimo. No entanto, a seleção do algoritmo e da heurística apropriados pode variar dependendo da configuração inicial do tabuleiro e dos objetivos específicos. Este trabalho destaca a importância de escolher estrategicamente as abordagens para otimizar a resolução de problemas de IA.