# Lista 9 - Implementação de Puzzle
# Inteligência Artificial

**Ana Carolina Couto Machado, Caroline Freitas Alvernaz e Guilherme Gomes Bruzzi**

[1]Instituto de Ciências Exatas e Informática – PUC Minas

## 1. Proposta

Desenvolvimento de um quebra-cabeça de 8 peças dispostas em um tabuleiro 3x3 em que deve-se movimentar uma peça por vez a fim de reorganizar as peças. Para isso, devem ser aplicados três algoritmos de busca para que o puzzle realize a solução automática e apresente a quantidade de movimentos usados por cada método e o tempo necessário para executar cada um. Além disso, foram utilizadas três heurísticas diferentes: Manhattan, a de peças fora do lugar e a distância euclidiana.

## 2. Métodos utilizados

Para este trabalho, foram escolhidos os algoritmos Busca Gulosa, Busca Uniforme e A*.

### 2.1. Busca Gulosa

A busca gulosa, ou best-first greedy search, expande os nós de menor custo, avaliando-os de acordo com a função heurística apenas e, portanto, não garante a solução ótima e não é completa, porém é um algoritmo rápido se a heurística for boa. A função do algoritmo é descrita por: **f(n) = h(n)**, sendo n o número de vértices e h(n) a heurística aplicada.

#### Descrição do Funcionamento da Busca Gulosa

1. Inicia-se a busca registrando o tempo inicial e inicializando:
    - a fila de prioridade vazia; e
    - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
    - um estado;
    - o caminho percorrido até aquele estado; e
    - o custo acumulado (não utilizado na prioridade).

   Essa função insere na fila de prioridade uma tupla contendo:
    - a prioridade, dada exclusivamente pela heurística escolhida;
    - o custo acumulado;
    - o estado atual; e
    - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó com menor heurística (menor valor estimado até o objetivo) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:
    - o caminho completo até a solução;
    - o número de passos (comprimento do caminho); e
    - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
    - ele é adicionado ao conjunto de visitados;

    - todos os estados vizinhos são gerados;
    - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
    - uma lista vazia;
    - zero passos; e
    - o tempo total de execução.

### 2.2. Busca Uniforme

A busca uniforme utiliza-se do algoritmo de Dijkstra para encontrar a solução: utiliza uma fila de prioridade que organiza os custos acumulados do caminho desde o início do percurso até o vértice alcançado. Assim, analisa-se apenas os pesos das arestas g(n), tendo sua função definida por **f(n) = g(n)**. Assim, diferente da busca gulosa, é garantida a busca ótima, se existir, e completa, pois todos os custos reais são analisados e, por isso, em contrapartida, se torna um algoritmo mais lento que a busca gulosa.

#### Descrição do Funcionamento da Busca Uniforme

1. Inicia-se a busca registrando o tempo inicial e inicializando:
    - a fila de prioridade vazia; e
    - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
    - um estado;
    - o caminho percorrido até aquele estado; e
    - o custo acumulado.

   Essa função insere na fila de prioridade uma tupla contendo:
    - a prioridade (igual ao custo, no caso da busca uniforme);
    - o custo acumulado;
    - o estado atual; e
    - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó de menor custo (ou seja, a menor prioridade) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:
    - o caminho completo até a solução;
    - o número de passos (comprimento do caminho); e
    - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
    - ele é adicionado ao conjunto de visitados;
    - todos os estados vizinhos são gerados;
    - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
    - uma lista vazia;
    - zero passos; e
    - o tempo total de execução.

### 2.3. Busca A*

Por fim, o algoritmo A* utiliza-se das duas métricas: custo real e estimativa de custo e, portanto, sua função é descrita por **f(n) = h(n) + g(n)**. O A* também garante a solução ótima, caso exista, e a busca completa, quando a heurística é admissível.

#### Descrição do Funcionamento da Busca A*

1. Inicia-se a busca registrando o tempo inicial e inicializando:
   - a fila de prioridade vazia; e
   - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
   - um estado;
   - o caminho percorrido até aquele estado; e
   - o custo acumulado.

   Essa função insere na fila de prioridade uma tupla contendo:
   - a prioridade, dada pela soma do custo acumulado e da heurística utilizada;
   - o custo acumulado;
   - o estado atual; e
   - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó com menor valor de prioridade (custo + heurística) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:
   - o caminho completo até a solução;
   - o número de passos (comprimento do caminho); e
   - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
   - ele é adicionado ao conjunto de visitados;
   - todos os estados vizinhos são gerados;
   - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
   - uma lista vazia;
   - zero passos; e
   - o tempo total de execução.

## 3. Heurísticas utilizadas

Os algoritmos de busca Gulosa e A* utilizam heurísticas para ajudar a encontrar o caminho mais rápido até a solução, sendo elas formas de estimar o quão longe o estado atual está do objetivo.

### 3.1. Heurística de Manhattan

A heurística de Manhattan foi escolhida para ser aplicada ao jogo porque representa com precisão a quantidade mínima de movimentos necessários para levar cada peça à sua posição correta, somando apenas deslocamentos horizontais e verticais. É especialmente eficaz no 8-puzzle, pois esse tipo de movimento é exatamente o permitido no jogo.

Essa heurística é admissível porque nunca superestima o custo real para alcançar o estado objetivo. Ela sempre fornece uma estimativa menor ou igual ao número real de movimentos necessários, pois ignora obstáculos ou restrições que possam aumentar o caminho, garantindo assim que a solução encontrada pelo algoritmo será ótima.

### 3.2. Heurística de Peças Fora do Lugar

A heurística de peças fora do lugar foi utilizada como uma alternativa mais simples, que apenas contabiliza o número de peças que não estão em sua posição final, sem considerar a distância que precisam percorrer. É uma heurística de fácil implementação e baixa complexidade computacional, o que a torna útil em situações onde o desempenho computacional é mais importante do que a precisão da estimativa.

Ela é considerada admissível porque nunca superestima o custo real, ou seja, cada peça fora do lugar contribui com 1 unidade na estimativa, embora na prática possa ser necessário mais de um movimento para posicioná-la corretamente. Assim, ela fornece uma estimativa otimista e válida para algoritmos como A*, garantindo correção na busca por soluções ótimas.

### 3.3. Heurística Euclidiana

A heurística euclidiana foi incluída com o objetivo de realizar comparações e estudos experimentais, mesmo que o movimento diagonal não seja permitido no 8-puzzle. Ela calcula a distância em linha reta entre a posição atual de cada peça e sua posição final, como se o movimento pudesse ocorrer em qualquer direção contínua.

Apesar disso, ela também é admissível, pois a distância euclidiana entre dois pontos em uma matriz 3x3 nunca será maior do que a quantidade de passos permitidos (horizontal/vertical) para chegar até o destino. Ou seja, embora a heurística seja menos apropriada para este tipo de jogo, ela ainda fornece uma estimativa que não supera o custo real, mantendo a admissibilidade.

## 4. Puzzle

O puzzle foi desenvolvido em linguagem Python e possui como interface inicial a figura abaixo que é composta pelo tabuleiro embaralhado inicialmente de maneira aleatória.

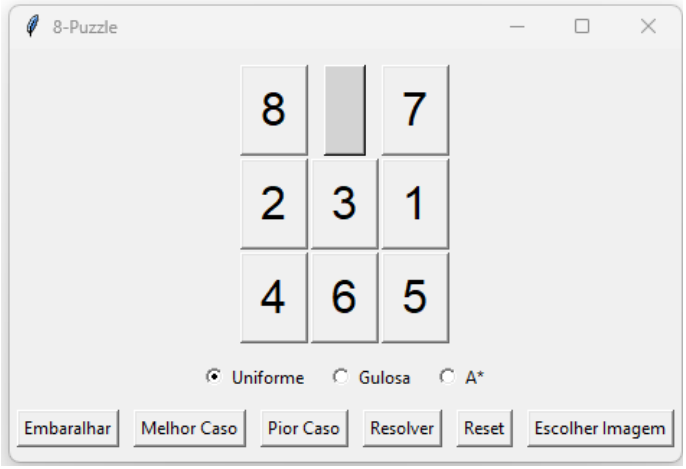


**Figura 1. Interface inicial do puzzle.**

Para iniciar o puzzle, deve-se selecionar o algoritmo a ser utilizado para chegar na solução e clicar na opção ”Resolver”para iniciar a ordenação, ou embaralhar o puzzle

novamente de maneira aleatória, ou escolher o pior caso (tabuleiro ordenado de forma decrescente), ou o melhor caso (tabuleiro ordenado de forma crescente) para realizar o teste e, após, fazer a seleção do algoritmo e depois clicar em ”Resolver”. Para os algoritmos de buscas Gulosa e A* deve-se escolher a heurística desejada. Após, o puzzle começa a realizar sua resolução de acordo com o algoritmo escolhido e, após, aparece um ”pop-up”indicando quantos movimentos foram necessários e o tempo gasto de processamento, conforme mostra a imagem abaixo.

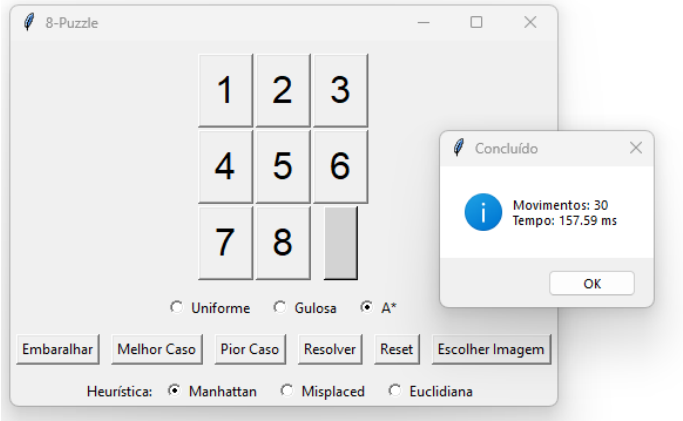


**Figura 2. Interface após a resolução do puzzle via Busca A* com heurística de Manhattan.**

Além disso, o usuário ainda tem a opção de escolher uma imagem para inserir no puzzle para que a foto seja reorganizada, conforme mostrado a seguir.

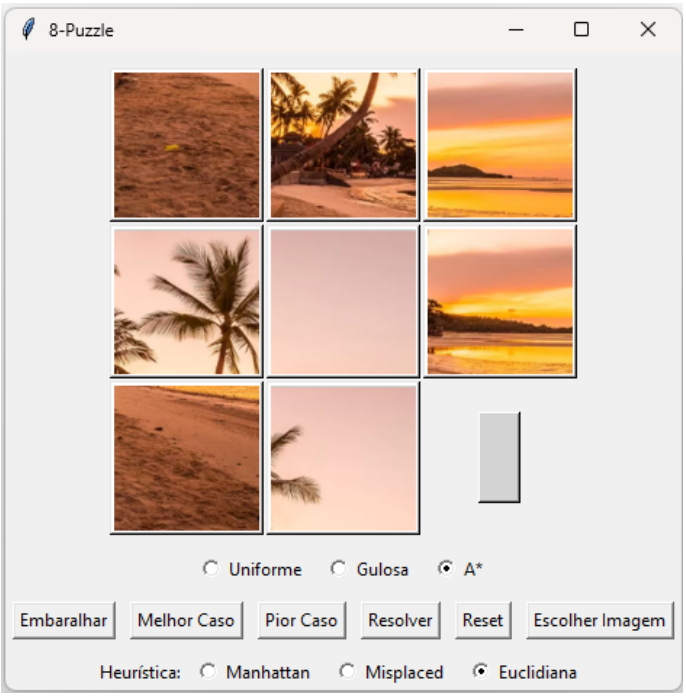


**Figura 3. Interface com figura escolhida pelo usuário.**

Ainda, para que se possa ser feita a comparação entre os métodos, deve-se clicar no botão ”Restart”para que o puzzle volte a ficar embaralhado como antes de iniciar a ordenação com o algoritmo escolhido. Após, é possível escolher um novo método a ser aplicado com o puzzle embaralhado da mesma forma de antes.

## 5. Resultados

Após a utilização do puzzle e realização de alguns testes, foi possível notar que o algoritmo de busca gulosa possui uma alta eficiência, uma vez que é, geralmente, o mais rápido entre os três analisados, porém tem uma baixa eficácia, já que não garante solução ótima nem busca completa. Já o algoritmo de busca uniforme tem a menor eficiência dos três, mas garante a solução ótima e busca completa. Por fim, o algoritmo A* tem uma média eficiência, contudo uma alta eficácia, pois, sendo a heurística admissível, garante solução ótima e busca completa.

Foram realizados três testes: um com o puzzle desordenado aleatoriamente, um com o puzzle ordenado e outro ordenado de maneira decrescente, representando o pior caso. As interfaces com os estados iniciais estão apresentadas abaixo.

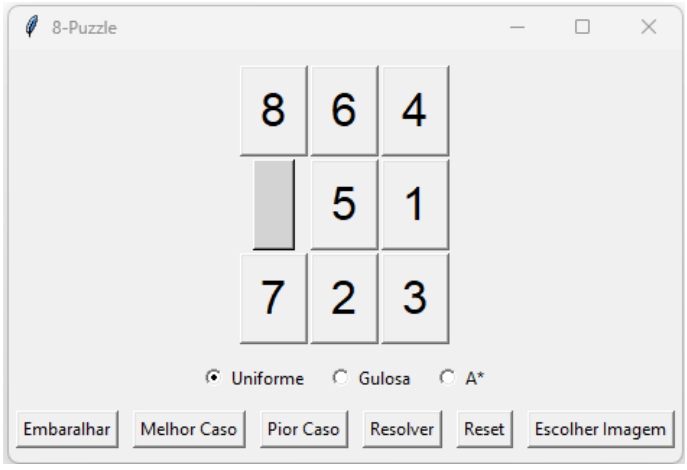


**Figura 4. Interface embaralhada aleatoriamente.**

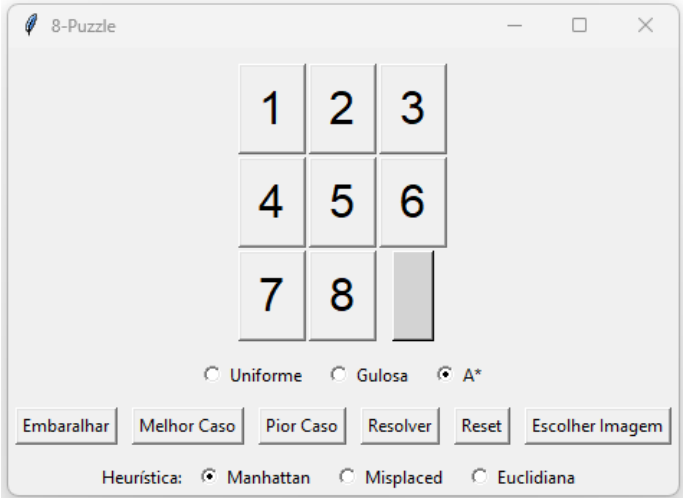


**Figura 5. Interface no melhor caso.**

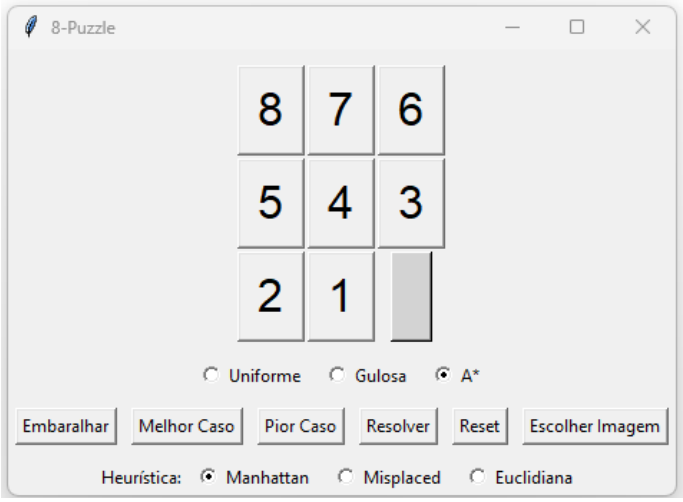


**Figura 6. Interface no pior caso.**

Na tabela a seguir, estão descritos os resultados obtidos para cada situação apresentada anteriormente.

**Tabela 1. Comparação entre os desempenhos dos algoritmos de busca**

| Tipo de ordenação | Algoritmo | Heurística utilizada | Movimentos necessários para chegar à solução | Tempo de execução (em ms) |
|---|---|---|---|---|
| Desordenado aleatoriamente | Busca Uniforme | - | 22 | 566,77 |
| | Busca Gulosa | Manhattan | 48 | 12,70 |
| | | Peças fora do lugar | 28 | 18,37 |
| | | Distância Euclidiana | 68 | 4,47 |
| | Busca A* | Manhattan | 22 | 17,20 |
| | | Peças fora do lugar | 22 | 44,34 |
| | | Distância Euclidiana | 22 | 8,81 |
| Ordenado de maneira descrescente (pior caso) | Busca Uniforme | - | 30 | 1.022,62 |
| | Busca Gulosa | Manhattan | 58 | 3,22 |
| | | Peças fora do lugar | 56 | 6,46 |
| | | Distância Euclidiana | 118 | 18,63 |
| | Busca A* | Manhattan | 30 | 105,88 |
| | | Peças fora do lugar | 30 | 757,66 |
| | | Distância Euclidiana | 30 | 215,60 |

Na tabela apresentada, é possível observar que a Busca Gulosa foi a mais rápida em praticamente todos os cenários testados. Isso ocorre porque esse algoritmo considera apenas o valor da heurística para escolher o próximo estado, ignorando o custo acumulado do caminho. No entanto, essa abordagem faz com que a Busca Gulosa não garanta o menor número de movimentos para resolver o puzzle, não garantindo a solução ótima. Ainda, ela apresentou diferentes quantidades de movimentos conforme a heurística utilizada porque sua decisão de qual caminho seguir depende exclusivamente do valor estimado pela heurística, sem considerar o histórico de passos já realizados. Cada heurística fornece uma forma diferente de estimar a proximidade do estado atual em relação ao objetivo, conforme apresentado anteriormente.

Já os algoritmos de Busca Uniforme e A* apresentaram tempos de execução maiores, especialmente em estados mais desorganizados. A Busca Uniforme avalia todos os caminhos possíveis sem ajuda de heurísticas, o que a torna completa e ótima, mas muito mais lenta, pois explora amplamente o espaço de estados. A Busca A*, embora mais eficiente que a Uniforme, ainda gasta mais tempo que a Gulosa, pois leva em conta tanto o custo acumulado quanto a heurística, o que exige mais processamento por nó analisado.

A performance do A* variou conforme a heurística utilizada. Com a heurística de Manhattan, apresentou o melhor desempenho geral, com tempo reduzido e menor número de movimentos, já que essa estimativa reflete com precisão os movimentos válidos no 8-puzzle. Ao usar a heurística de peças fora do lugar, o A* continuou a encontrar a solução ótima, mas com desempenho levemente inferior, pois essa heurística apenas contabiliza peças fora de posição, sem considerar a distância real. Já com a heurística euclidiana, embora admissível, o desempenho foi o pior, pois ela se baseia em deslocamentos diagonais, que não existem no jogo, tornando a estimativa menos eficaz.

Na Busca Gulosa, a variação no desempenho foi ainda mais evidente. Com a heurística de Manhattan, o algoritmo teve o melhor desempenho entre as três, escolhendo caminhos mais promissores e precisos. Com a heurística de peças fora do lugar, houve um aumento no número de movimentos e tempo de execução, já que a estimativa é menos

informativa. Usando a heurística euclidiana, os resultados foram os menos eficientes, pois o algoritmo tomou decisões baseadas em distâncias irreais para o tipo de movimentação permitida, levando a trajetórias mais longas e soluções menos racionais.

## 6. Conclusão

O algoritmo A*, para os testes realizados, se mostrou com o melhor custo-benefício, uma vez que ele fornece a solução ótima, caso haja solução, e a busca completa em um tempo razoável que, por mais que seja mais demorado que a busca gulosa, não apresenta um tempo médio de execução tão longo quanto a busca uniforme que também apresenta a solução ótima, ou seja, a mesma solução do A*. Por fim, é importante ressaltar que os tempos médios de execução dependem do estado inicial do puzzle, podendo os algoritmos apresentar tempos maiores ou menores que os apresentados e ter seus tempos mais aproximados, ou mais afastados uns dos outros, como é o caso da solução para o melhor caso em que nenhuma movimentação é feita, tendo o tempo de execução próximo de zero para os três algoritmos, porém, para os demais casos, a solução ótima, se existir e dada uma heurística admissível, será a mesma.