8-Puzzle AI-Based Solution

André Luiz Rocha Cabral
Douglas Nícolas Silva Gomes
João Paulo Dias Estevão
Victor Souza Lima

Matéria: Inteligência Artificial
Professora: Cristiane Neri Nobre
Data de entrega: 04/05/2025

Link do executável: Drive
Link do código: GitHub

Neste trabalho foram implementados três diferentes métodos de busca/caminhamento para resolução do puzzle:

1. BFS (Beadth-First Search) ou Busca em largura:

Este método implementa a **busca em largura (BFS)** como estratégia para encontrar uma solução para o problema, explorando o espaço de estados em **níveis crescentes de profundidade**. A BFS garante encontrar a solução com o **menor número de passos** (caso os custos de transição entre estados sejam iguais), mas **nem sempre encontra a solução mais eficiente em termos de custo computacional ou tempo**.

Estruturas utilizadas

- **Fila (queue)**: utilizada para armazenar os estados a serem explorados, obedecendo à ordem FIFO (primeiro a entrar, primeiro a sair). O estado inicial é o primeiro a ser inserido.
- **Dicionário de visitados (`visited`)**: armazena os estados já visitados, mapeando cada estado para seu **estado predecessor**. Isso permite, ao final, reconstruir o caminho que levou até a solução.

Funcionamento do algoritmo

1. O algoritmo inicia com a inserção do estado inicial na fila e seu registro como visitado.
2. Em seguida, entra em um **laço de repetição (loop)** que continua enquanto houver estados na fila.
3. A cada iteração:
    a. O estado mais antigo da fila é removido (respeitando a ordem da BFS).
    b. Verifica-se se este estado é o **estado objetivo** — no caso, representado pela string `"123456780"`.
    c. Se for o estado desejado:
        i. Calcula-se o tempo de execução.
        ii. Reconstrói-se o caminho da solução usando os predecessores armazenados.
        iii. Retorna-se esse caminho, o tempo gasto e o número de estados expandidos.
4. Se ainda **não** for o estado objetivo:
    a. Os **vizinhos** (ou seja, os estados que podem ser alcançados a partir do estado atual com uma única ação) são gerados.
    b. Para cada vizinho:

i. Verifica-se se ainda **não foi visitado**.
ii. Se não foi:
    1. Ele é inserido na fila.
    2. O contador de estados expandidos é incrementado.
    3. O estado é marcado como visitado, registrando seu predecessor.

Este processo se repete até que a fila esteja vazia (sem mais estados a explorar) ou até que a solução seja encontrada.

🚫 *Ausência de heurística*

Este método **não utiliza nenhuma heurística** para guiar a busca. Isso significa que a escolha de qual estado expandir a seguir é **puramente baseada na ordem de inserção na fila**, sem qualquer avaliação da "proximidade" de um estado em relação ao objetivo.

2. GBFS (Greedy Best-First Search) ou Busca Gulosa

Este método implementa a **busca gulosa de melhor primeiro**, uma estratégia que utiliza exclusivamente uma **função heurística** para decidir qual estado explorar a seguir. Ela prioriza os estados que **parecem mais próximos da solução**, mas **ignora o custo real acumulado** do caminho até aquele ponto.

🧱 *Estruturas utilizadas*

- **Fila de prioridade (`open_set`)**: contém os estados a serem explorados, ordenados pela sua heurística. O estado com menor valor heurístico é o primeiro a ser expandido.
- **Conjunto de visitados (`visited_set`)**: garante que cada estado seja expandido no máximo uma vez, evitando ciclos ou repetições.
- **Dicionário de predecessores (`predecessors`)**: armazena para cada estado o seu antecessor, necessário para reconstruir o caminho até a solução.

🔄 *Funcionamento do algoritmo*

1. O estado inicial é adicionado à fila de prioridade com base no seu valor heurístico.
2. Marca-se o estado inicial como visitado e sem predecessor.
3. O algoritmo entra em um **laço de repetição que continua enquanto houver estados na fila**:
    a. Remove o estado com **menor heurística**.
    b. Verifica se é o **estado objetivo** (`"123456780"`).
        i. Se for:
            1. Calcula o tempo total.
            2. Reconstrói o caminho usando os predecessores.
            3. Retorna o caminho, o tempo e o número de nós expandidos.

4. Caso ainda não seja o objetivo:
    a. Gera os **vizinhos** do estado atual.
    b. Para cada vizinho não visitado:
        i. Incrementa o contador de nós expandidos.
        ii. Adiciona o vizinho à fila com base na sua heurística.
        iii. Marca-o como visitado.
        iv. Registra seu predecessor.

Caso a fila se esvazie sem encontrar a solução, a função retorna um caminho vazio.

🧠 *Heurística utilizada: Distância de Manhattan*

A heurística empregada neste algoritmo é a **Distância de Manhattan**, que calcula para cada peça (exceto o espaço vazio `0`) a soma das distâncias verticais e horizontais até sua posição correta no estado objetivo. A soma dessas distâncias para todas as peças compõe o valor heurístico total do estado.

Essa abordagem fornece uma **estimativa admissível** e eficaz da distância até a solução, pois representa o **mínimo número de movimentos necessários** para cada peça alcançar sua posição final, sem considerar bloqueios.

Vantagem: A heurística de Manhattan é simples, rápida de calcular e geralmente conduz a uma busca eficiente e informada.

Limitação: Como a busca gulosa ignora o custo do caminho já percorrido, ela **não garante encontrar a solução ótima**, mesmo com uma boa heurística.

3. A* (A-Star)

O algoritmo **A*** é uma das estratégias de busca mais eficazes para encontrar o **caminho ótimo** em um espaço de estados, como o 8-puzzle. Ele combina duas ideias principais:

- **Custo real do caminho percorrido** até o estado atual (`g(n)`).
- **Estimativa do custo restante até o objetivo**, baseada em uma **função heurística** (`h(n)`).

O A* prioriza os estados com menor valor de `f(n) = g(n) + h(n)`.

🧱 *Estruturas utilizadas*

- **Fila de prioridade (`open_set`)**: armazena os estados a serem explorados, ordenados pelo valor `f(n)`.
- **Dicionário `g_score`**: armazena o custo real acumulado para chegar a cada estado.
- **Conjunto `closed_set`**: guarda estados que já foram completamente explorados.

- **Dicionário `predecessors`**: armazena o predecessor de cada estado, necessário para reconstruir o caminho da solução.

🔄 *Funcionamento do algoritmo*

1. O tempo de execução é iniciado e o custo `g` do estado inicial é registrado como zero.
2. O estado inicial é adicionado à fila de prioridade, com `f = g + h`, onde `h` é o valor da heurística selecionada.
3. A heurística a ser utilizada é determinada por uma **string dinâmica**, como `"heuristic"` ou `"heuristic2"`:
    a. Se o atributo especificado existe, ele será usado.
    b. Caso contrário, assume-se `"heuristic"` como padrão.
4. O algoritmo entra em um **loop principal**, que continua enquanto houver estados na fila:
    a. Remove o estado com menor valor de `f`.
    b. Se já foi explorado (`closed_set`), ignora.
    c. Se é o estado objetivo (`"123456780"`), termina a busca e reconstrói o caminho.
5. Para cada vizinho do estado atual:
    a. Se ainda não foi explorado ou se um caminho melhor foi encontrado:
        i. Atualiza `g_score`, `predecessors` e recalcula `f = g + h`.
        ii. Adiciona o vizinho à fila de prioridade.

Se a fila esvaziar sem atingir o objetivo, retorna um caminho vazio.

🧠 *Heurísticas utilizadas*

Este algoritmo foi configurado para aceitar **múltiplas heurísticas dinamicamente**. Dois métodos heurísticos estão disponíveis:

1. Distância de Manhattan (heuristic)

Calcula a soma das distâncias verticais e horizontais entre a posição atual e a posição objetivo de cada peça, ignorando o espaço vazio.

- Admissível e consistente
- Rápida de calcular
- Pode subestimar situações onde há bloqueios entre peças

2. Manhattan com penalidade por conflitos lineares (heuristic2)

Amplia a heurística de Manhattan, penalizando casos em que duas peças estão na mesma linha ou coluna e **impedem uma à outra de chegar à posição correta**, somando +2 por conflito.

- Ainda admissível
- Mais precisa que Manhattan pura
- Exige mais processamento (duplos `for` em linhas e colunas)

🌐 *Análise de Resultados*

O 8-puzzle é um jogo de tabuleiro 3x3 com oito peças e um espaço vazio, cujo objetivo é atingir uma configuração final por meio de movimentos válidos. Embora existam 9! (362.880) possíveis configurações, apenas metade delas são solucionáveis, formando um componente conexo próprio com 181.440 estados. O diâmetro (a maior distância mínima entre quaisquer dois estados solucionáveis) é 31, ou seja, **qualquer configuração solucionável pode ser resolvida em no máximo 31 movimentos (ótima)**. Esse valor foi determinado por análise computacional exaustiva do espaço de estados.

Estado Inicial para o Pior Caso:

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

Para a mesma configuração do jogo, o algoritmo **A* (**manhattan penality) foi o que melhor desenvolveu o problema, pois encontrou a solução com o **menor número de movimentos (31)**, igual à Busca em Largura, mas com **menor número de estados expandidos** e **tempo de execução significativamente inferior**.

Tabela Comparativa para o Pior Caso:

Critério	Busca em Largura	Busca Gulosa	A* (manhattan)	A* (penality)
Caminho (nº de movimentos)	31	47	31	31
Tempo de execução (s)	11,7995s	0,0072s	1,3975s	0,8419s
Estados expandidos	181.440	161	29757	18578
Solução ótima?	Sim	Não	Sim	Sim

Conclusões:

- **Busca em Largura** é completa e ótima, encontrando a solução mínima (31 movimentos), mas tem **alto custo computacional**, com **o maior número de estados expandidos e maior tempo de execução** entre os três métodos.
- **Busca Gulosa** foi a mais rápida e eficiente em termos de tempo e nós expandidos, mas **sacrificou a qualidade da solução**, encontrando um caminho com mais movimentos (47), o que pode ser ineficiente dependendo do contexto do problema.

- **A*** obteve um **melhor equilíbrio entre custo e qualidade da solução**: manteve o caminho mínimo (31 movimentos) como a Busca em Largura, mas com **menos estados expandidos** e **tempo de execução menor**, o que o torna mais eficiente que os anteriores.
- **A* (penality)** apresentou um desempenho ainda mais eficiente que a versão tradicional com Manhattan puro. Ele manteve o caminho (31 movimentos), mas com **menor número de estados expandidos** (18578) e **tempo de execução reduzido** (0,8419s). Isso demonstra que a penalidade aplicada à heurística contribuiu para guiar a busca de forma mais informada, resultando em **melhor aproveitamento computacional sem comprometer a qualidade da solução**, tornando-o o método **mais eficiente no geral.**