**Universidade Federal de Juiz de Fora**
**Departamento de Ciência da Computação**
**INTELIGÊNCIA ARTIFICIAL**

# N - Puzzle
# Algoritmos de Busca

## Integrantes do Grupo:

Breno Montanha - 202265513B

Lucas Henrique Nogueira - 202265515B

Professor: Saulo Moraes Villela

Relatório do trabalho final da disciplina DCC014 - Inteligência Artificial, parte integrante da avaliação da mesma.

Juiz de Fora

Agosto de 2025

# 1 Introdução

Este relatório tem como objetivo apresentar o desenvolvimento e a avaliação de uma suíte de algoritmos de busca para a solução do problema clássico N-Puzzle. Este problema consiste em reorganizar um conjunto de peças numeradas em uma grade, a partir de uma configuração inicial, para alcançar um estado objetivo predefinido, utilizando um espaço vazio para movimentar as peças. O N-Puzzle é um problema canônico no campo da Inteligência Artificial, servindo como um ambiente de teste robusto para o estudo e a comparação de algoritmos de busca, avaliando sua eficiência e eficácia na exploração de espaços de estados complexos [2].

No decorrer deste trabalho, foram implementados e analisados os sete algoritmos de busca estipulados. O primeiro grupo, de busca não informada, abrange os métodos: Backtracking, Busca em Largura (BFS), Busca em Profundidade (DFS) e Busca Ordenada. O segundo grupo, de busca informada, é composto pelos algoritmos: Busca Gulosa (*Greedy Search*), A* e IDA*. Para guiar os algoritmos de busca informada, foram implementadas e avaliadas diversas funções heurísticas , a saber: Distância de Manhattan , Distância Euclidiana , Peças Fora do Lugar (*Misplaced Tiles*) , Conflito Linear (*Linear Conflict*) , Ciclos de Permutação (*Permutation Cycles*) e Soma Ponderada (*Weighted Sum*). O desempenho dos métodos foi sistematicamente comparado com base em métricas de eficiência e qualidade da solução.

## 1.1 Organização da Equipe e Divisão de Tarefas

O desenvolvimento do projeto foi um esforço colaborativo, com as tarefas distribuídas entre os membros da equipe para garantir a cobertura de todas as frentes de trabalho, desde a pesquisa teórica até a implementação e documentação. A Tabela 1 detalha as principais responsabilidades de cada integrante.

Tabela 1: Divisão de tarefas entre os integrantes da equipe.

| **Integrante** | **Responsabilidades Principais** |
|---|---|
| Lucas | • Arquitetura do sistema e desenvolvimento do Backend em C++.<br>• Desenvolvimento do Frontend web e da comunicação com o Backend.<br>• Implementação e depuração dos algoritmos de busca (Backtracking, DFS, A*).<br>• Pesquisa, implementação e teste das funções heurísticas(Manhattan, Conflito Linear, Soma Ponderada).<br>• Gerenciamento do repositório e do Makefile. |
| Breno | • Pesquisa, implementação e teste das funções heurísticas(Misplaced Tiles, Ciclos de Permutação e Euclidiana).<br>• Implementação e depuração dos algoritmos de busca (BFS, Ordenada, IDA*, Gulosa).<br>• Criação da interface de linha de comando (CLI).<br>• Planejamento dos experimentos e análise dos resultados.<br>• Otimização de Algoritimos para maior velocidade de execução |

A estrutura do relatório está organizada da seguinte forma: a Seção 2 descreve formalmente o problema do N-Puzzle. A Seção 3, são detalhados os algoritmos e as heurísticas implementadas. Na Seção 4, apresenta a arquitetura da solução desenvolvida. A Seção 5 descreve os experimentos computacionais e a análise comparativa dos resultados. Por fim, a Seção 6 apresenta as conclusões do estudo.

# 2 Descrição do Problema

O N-Puzzle é um quebra-cabeça de deslizamento que consiste em uma grade de dimensão $n \times m$, preenchida com $N = n \times m - 1$ peças numeradas de forma única e um espaço vazio. O objetivo do jogo é, a partir de um estado inicial, aplicar uma sequência de movimentos para alcançar um estado final predeterminado. Um movimento é definido como o deslizamento de uma peça adjacente para a posição do espaço vazio.

Matematicamente, o problema pode ser formalizado da seguinte maneira:

- **Estado:** Uma configuração específica das $N$ peças e do espaço vazio na grade.
- **Espaço de Estados:** O conjunto de todos os estados alcançáveis a partir do estado inicial através de movimentos válidos.
- **Ação (ou Operador):** A troca de posição entre uma peça e o espaço vazio adjacente (vertical ou horizontalmente).
- **Custo da Ação:** O custo associado a cada movimento. Neste trabalho, adota-se um custo uniforme igual a 1 para qualquer ação.
- **Estado Final:** O estado que se deseja alcançar para finalizar a execução.

O problema consiste em encontrar uma sequência de ações que transforme o estado inicial $s_{inicial}$ no estado final $s_{final}$. Uma solução é considerada ótima se o custo total do caminho, que corresponde à soma dos custos das ações (ou seja, o número de movimentos), for mínimo. O custo do caminho até um estado $s_{final}$, denotado por $g(s_{final})$, é a soma dos custos das ações desde $s_{inicial}$ até $s_{final}$.

Para os algoritmos de busca informada, utiliza-se uma **função heurística**, $h(s)$, que estima o custo do caminho mais barato do estado $s$ até o estado final. A combinação do custo real e da estimativa heurística resulta na função de avaliação $f(s)$, utilizada por algoritmos como o A* e o IDA* para ordenar os nós a serem explorados:

$$f(s) = g(s) + h(s) \tag{1}$$

O problema N-Puzzle é classificado como NP-difícil, uma vez que o tamanho do espaço de estados cresce de forma fatorial com o número de peças. A viabilidade de uma solução para uma dada configuração inicial depende da paridade do número de inversões (pares de peças que estão em ordem trocada em relação ao objetivo).

# 3 Algoritmos e Heurísticas Implementadas

Para a solução do problema do N-Puzzle, foram implementados sete diferentes algoritmos de busca, divididos em duas categorias principais: busca não informada e busca informada. Adicionalmente, um conjunto de seis funções heurísticas foi desenvolvido para guiar os algoritmos de busca informada. O código fonte pode ser encontrado no repositório do GitHub [1]

## 3.1 Algoritmos de Busca Não Informada

Os algoritmos de busca não informada, ou busca cega, exploram o espaço de estados sem qualquer conhecimento sobre a distância ou o custo para se chegar ao estado objetivo. A ordem de exploração é determinada unicamente pela estrutura da árvore de busca.

### 3.1.1 Backtracking

**Algorithm 1:** Busca por Backtracking

**Input:** Tabuleiro atual `board`, Pai `state`
**Output:** Solução encontrada ou falha

```
1  if board é o tabuleiro objetivo then
2  |  return path
3  end
4  foreach ação a em Ações(U, D, L, R) do
5  |  child_board ← Move(board, a)
6  |  if child_board não está em path then
7  |  |  Adiciona child_board a path
8  |  |  result ← Backtracking(child_board, board)
9  |  |  if result não é falha then
10 |  |  |  return result
11 |  |  end
12 |  |  Remove child_board de path ;                    // Backtrack
13 |  end
14 end
15 return falha
```

### 3.1.2 Busca em Largura (BFS - Breadth-First Search)

**Algorithm 2:** Busca em Largura (BFS)

**Input:** Tabuleiro inicial `board`
**Output:** Solução ou falha

```
1  lista_abertos ← Fila() com o nó inicial
2  visitados ← Conjunto() com o tabuleiro inicial
3  while lista_abertos não está vazia do
4  |   node ← Desenfileirar(lista_abertos)
5  |   if node.board é o tabuleiro objetivo then
6  |   |   return Solução(path)
7  |   end
8  |   foreach ação a em Ações(U, D, L, R) do
9  |   |   child_board ← Move(board, a)
10 |   |   if child_board não está em visitados then
11 |   |   |   Adicionar child_board a visitados
12 |   |   |   Enfileirar(child, lista_abertos)
13 |   |   end
14 |   end
15 end
16 return falha
```

### 3.1.3 Busca em Profundidade (DFS - Depth-First Search)

**Algorithm 3:** Busca em Profundidade Limitada (DFS)

**Input:** Tabuleiro inicial `board`, Limite de profundidade `limit`

**Output:** Solução ou falha

```
1  lista_abertos ← Pilha() com o nó inicial
2  visitados ← Conjunto() com o tabuleito inicial
3  while lista_abertos não está vazia do
4      node ← Desempilhar(lista_abertos)
5      if node.board é o tabuleiro objetivo then
6          return Solução(path)
7      end
8      if node.depth < limit then
9          foreach ação a em Ações(U, D, L, R) do
10             child_board ← Move(board, a)
11             if child_board não está em visitados then
12                 Adicionar child_board a visitados
13                 Empilhar(child, lista_abertos)
14             end
15         end
16     end
17 end
18 return falha
```

### 3.1.4 Busca Ordenada (Uniform Cost Search)

**Algorithm 4:** Busca Ordenada

**Input:** Tabuleiro inicial `board`

**Output:** Solução ou falha

```
1  lista_abertos ← FilaDePrioridade() com o nó inicial (prioridade por g(s))
2  visitados ← Conjunto() com o tabuleiro inicial ;
3  while lista_abertos não está vazia do
4      node ← ExtrairMínimo(lista_abertos)
5      if node.board é o estado objetivo then
6          return Solução(path)
7      end
8      foreach ação a em Ações(U, D, L, R) do
9          child_board ← Move(board, a)
10         if child_board não está em visitados then
11             Adicionar child_board a visitados
12             Inserir(child, lista_abertos)
13         end
14         else if nó com mesmo estado em lista_abertos tem g maior then
15             Substituir nó na lista_abertos pelo child
16         end
17     end
18 end
19 return falha
```

## 3.2 Algoritmos de Busca Informada

Os algoritmos de busca informada utilizam conhecimento específico do problema, na forma de uma função heurística $h(s)$, para guiar a exploração do espaço de estados de forma mais eficiente em direção ao objetivo.

### 3.2.1 Busca Gulosa (Greedy Best-First Search)

**Algorithm 5:** Busca Gulosa

**Input:** Tabuleiro inicial `board`
**Output:** Solução ou falha

```
1  lista_abertos ← FilaDePrioridade() com o nó inicial (prioridade por h(s))
2  visitados ← Conjunto() com o tabuleiro inicial ;
3  while lista_abertos não está vazia do
4      node ← ExtrairMínimo(lista_abertos)
5      if node.board é o estado objetivo then
6          return Solução(path)
7      end
8      foreach ação a em Ações(U, D, L, R) do
9          child_board ← Move(board, a)
10         if child_board não está em visitados then
11             Adicionar child_board a visitados
12             Inserir(child, lista_abertos)
13         end
14     end
15 end
16 return falha
```

### 3.2.2 Busca A*

**Algorithm 6:** Busca A*

**Input:** Tabuleiro inicial `board`

**Output:** Solução ou falha

```
1  lista_abertos ← FilaDePrioridade() com o nó inicial (prioridade por f(s))
2  visitados ← Conjunto() com o tabuleiro inicial ;
3  while lista_abertos não está vazia do
4      node ← ExtrairMínimo(lista_abertos)
5      if node.board é o estado objetivo then
6          return Solução(path)
7      end
8      foreach ação a em Ações(U, D, L, R) do
9          child_board ← Move(board, a)
10         if child_board não está em visitados then
11             Adicionar child_board a visitados
12             Inserir(child, lista_abertos)
13         end
14         else if nó com mesmo estado em lista_abertos tem f maior then
15             Substituir nó na lista_abertos pelo child
16         end
17     end
18 end
19 return falha
```

### 3.2.3 Busca IDA* (Iterative Deepening A*)

**Algorithm 7:** Busca IDA*

**Input:** Tabuleiro inicial `board`, Heurística `h`, Patamar `p`

**Output:** Solução ou falha

```
1  if board é o tabuleiro objetivo then
2  |  return path
3  end
4  if p ≤ board.cost then
5  |  foreach ação a em Ações(U, D, L, R) do
6  |  |  child_board ← Move(board, a)
7  |  |  if child_board não está em path then
8  |  |  |  Adiciona child_board a path
9  |  |  |  result ← BuscaIDA*(child_board, board)
10 |  |  |  if result não é falha then
11 |  |  |  |  return result
12 |  |  |  end
13 |  |  |  Remove child_board de path
14 |  |  end
15 |  end
16 end
17 else
18 |  Atualiza patamar para próxima execução
19 end
20 return falha
```

## 3.3 Funções Heurísticas

As funções heurísticas estimam o custo de se chegar do estado atual ao estado objetivo. A qualidade da heurística impacta diretamente a performance dos algoritmos de busca informada.

### 3.3.1 Peças Fora do Lugar (Misplaced Tiles)

Esta heurística simplesmente conta o número de peças que não estão em sua posição de destino. É computacionalmente barata, mas pouco informativa.

$$h(s) = \sum_{p=1}^{N} \begin{cases} 1 & \text{se a peça } p \text{ não está na posição correta} \\ 0 & \text{caso contrário} \end{cases}$$

### 3.3.2 Distância de Manhattan

Calcula a soma das distâncias horizontais e verticais de cada peça de sua posição atual até sua posição objetivo. É uma heurística admissível e mais informativa que a contagem de peças fora do lugar.

$$h(s) = \sum_{p=1}^{N}(|x_p - x'_p| + |y_p - y'_p|)$$

Onde $(x_p, y_p)$ é a posição atual da peça $p$ e $(x'_p, y'_p)$ é sua posição objetivo.

### 3.3.3 Distância Euclidiana

Similar à Distância de Manhattan, mas calcula a distância em linha reta (a hipotenusa) entre a posição atual e a posição objetivo de cada peça.

$$h(s) = \sum_{p=1}^{N}\sqrt{(x_p - x'_p)^2 + (y_p - y'_p)^2}$$

### 3.3.4 Conflito Linear (Linear Conflict)

É um refinamento da Distância de Manhattan. Além da distância, adiciona uma penalidade (normalmente 2 movimentos) para cada par de peças que estão na mesma linha ou coluna de seus objetivos, mas em posições invertidas, obrigando que uma saia do caminho para a outra passar.

### 3.3.5 Ciclos de Permutação (Permutation Cycles)

Esta heurística analisa a permutação das peças como um conjunto de ciclos. Para cada peça fora do lugar, ela segue a peça que ocupa sua posição correta, formando um ciclo. O custo é a soma do número de peças fora do lugar mais o número de ciclos.

### 3.3.6 Soma Ponderada (Weighted Sum)

Não é uma heurística única, mas uma abordagem que combina múltiplas heurísticas através de uma soma ponderada. Nesse trabalho, foi combinar a Distância de Manhattan com a Peças Fora do Lugar para obter uma estimativa mais precisa.

$$h(s) = w_1 \cdot h_1(s) + w_2 \cdot h_2(s) + \ldots$$

Onde $w_i$ são os pesos e $h_i(s)$ são as diferentes funções heurísticas.

# 4 Arquitetura da Solução Desenvolvida

A solução desenvolvida para o problema do N-Puzzle foi projetada para oferecer duas formas de interação: uma via terminal de comando e outra através de uma interface gráfica web. Para isso, a arquitetura foi dividida em um Backend, responsável pela lógica dos algoritmos, e um Frontend, para a interface gráfica web.

## 4.1 Backend

O núcleo da solução reside no Backend, onde toda a lógica dos algoritmos de busca foi implementada em C++.

- **Linguagem e Performance:** A escolha do C++ se justifica pela alta performance e pelo controle de baixo nível sobre o gerenciamento de memória, características essenciais para lidar com a complexidade e o grande volume de nós gerados durante a execução das buscas.
- **Servidor Web:** Para a versão com interface gráfica, o web framework Crow foi utilizado para expor os algoritmos de C++ como um serviço. O Crow permitiu a criação de um servidor HTTP capaz de receber requisições do Frontend, invocar as funções de busca e retornar os resultados.
- **Estruturas de Dados:** A eficiência da implementação dependeu fortemente da escolha de estruturas de dados adequadas para cada tarefa:
  - **Hash:** Devido ao fato de que com o hash, conseguimos transformar um vetor, no qual o custo de comparação é $O(n)$ para um número que pode ser comparado em $O(1)$, decidimos usar o hash para determinarmos o fim de jogo, reduzindo a complexidade da função de finalização.
  - **Vetor:** Escolhemos representar o jogo como vetor, e não como matriz, pois algumas funções heuristicas precisariam de representar o tabuleiro como vetor, e para as demais verificações não faria tanta diferença.
  - **Recursão:** Escolhemos utilizar a recursão, apesar de seu maior de consumo de memória em comparação a uma pilha para o caso do Backtracking e IDA* pelo fato de tornar o código mais legível.
  - **Heap:** Gostariamos de ter implementado uma Heap de mínima para aumentar a velocidade de extração dos menores custos(para o caso da busca ordenada, busca gulosa, A* e IDA*), visto que nossa implementação atual, a cada remoção do menor custo, temos de percorrer toda a lista de abertos para descobrir qual é o próximo menor custo, e isso torna acomplexidade de remoção para $O(n)$.

## 4.2 Frontend e Interação com o Usuário

O Frontend é a camada de apresentação da aplicação, responsável por toda a interação com o usuário através do navegador. Foi desenvolvido utilizando tecnologias web padrão (HTML, CSS e JavaScript). A interface guia o usuário através de um fluxo claro para testar os algoritmos:

1. **Configuração do Problema:** O usuário inicia selecionando a opção "Testar Algoritmos de Busca". Em seguida, define o tamanho do tabuleiro (de 2x2 a 5x5) e especifica os estados inicial e final, que podem ser inseridos manualmente ou gerados de forma aleatória pelo sistema.

2. **Seleção do Algoritmo e Heurística:** Com o problema configurado, a interface apresenta um painel de controle onde o usuário pode selecionar um dos sete algoritmos de busca. Caso um algoritmo informado (Greedy, A* ou IDA*) seja escolhido, um painel adicional é exibido para a seleção de uma das seis heurísticas implementadas.

3. **Execução e Visualização:** Após clicar em "Rodar Algoritmo", uma requisição é enviada ao Backend. Ao receber a resposta, o Frontend exibe as estatísticas da execução (tempo, custo, nós expandidos, etc.) e apresenta uma animação do caminho da solução no tabuleiro.

## 4.3 Comunicação entre Camadas

A comunicação entre o Frontend (JavaScript) e o Backend (C++/Crow) é realizada através de uma API (Interface de Programação de Aplicações) baseada no protocolo HTTP, seguindo um modelo de requisição e resposta (Request/Response). O JavaScript coleta os dados do problema (tabuleiros, algoritmo, heurística), envia uma requisição HTTP para um *endpoint* específico no servidor Crow, que por sua vez processa a requisição, executa a busca e retorna a solução e as estatísticas. O Frontend então interpreta esses dados e atualiza a interface para o usuário.

## 4.4 Ambiente de Execução e Compilação

O projeto foi inteiramente desenvolvido em C++ e utiliza um `Makefile` para gerenciar o processo de compilação. O `Makefile` foi projetado para ser compatível com ambientes Linux e Windows, desde que as ferramentas de compilação necessárias estejam disponíveis no sistema.

**Requisitos de Ambiente:** Para compilar e executar o projeto, os seguintes componentes são necessários em cada sistema operacional:

- **No Linux:**
  - Um compilador C++, como o `g++`.
  - O utilitário `make`.
- **No Windows:**
  - Um ambiente de desenvolvimento que forneça as ferramentas GNU, como o **MinGW-w64** (geralmente utilizado através do terminal do **MSYS2**).

**Compilação e Execução:** Com o ambiente devidamente preparado, a compilação e a execução são realizadas através dos seguintes comandos no terminal:

```
# 1. Compilar o projeto usando o Makefile
# Este comando gera os executaveis de acordo com o SO
$ make

# 2. Executar a versao via terminal ou via servidor
$ ./npuzzle_exec      # Terminal
$ ./npuzzle_server    # Servidor

# Se escolher via servidor, acesse em seu navegador:
# http://localhost:18080/
```

Listing 1: Compilação e execução do projeto.

Caso o sistema operacional seja Windows, os executáveis devem ser acompanhados da extensão (`.exe`).

Uma vez que o servidor web é iniciado, a interface gráfica pode ser acessada através de um navegador para a realização de testes interativos.

# 5 Experimentos Computacionais

Nesta seção, serão descritos todos os detalhes relacionados aos experimentos computacionais realizados para avaliar o desempenho dos algoritmos de busca implementados. Para isso, são definidas subseções que abordam as instâncias de teste, o ambiente utilizado e a metodologia de execução.

## 5.1 Descrição das Instâncias

Para os experimentos, foi utilizado um conjunto de instâncias de teste personalizadas, armazenadas no diretório `data/`. Cada instância representa um problema N-Puzzle específico, com estados inicial e final definidos em um arquivo de texto.

O formato dos arquivos de instância é o seguinte:

- A primeira linha contém o número de linhas ($n$) do tabuleiro.
- A segunda linha contém o número de colunas ($m$) do tabuleiro.
- As $n$ linhas seguintes descrevem a configuração do **estado objetivo**, com os números das peças separados por espaços. O número 0 representa o espaço vazio.
- As últimas $n$ linhas descrevem a configuração do **estado inicial** do problema.

A Tabela 2 descreve as instâncias que foram consideradas para a bateria de testes automatizada. Note que instâncias de maior dimensão, como `test10x10.txt` e as variações de 4x4 e 5x5, foram excluídas da execução em lote via script para garantir que os testes fossem concluídos em tempo hábil, dada a complexidade exponencial do problema.

Tabela 2: Exemplo de instâncias utilizadas nos testes automatizados.

| **Nome do Arquivo** | **Dimensão** |
|---|---|
| test3x3_1.txt | 3x3 |
| test3x3_2.txt | 3x3 |
| test3x3_3.txt | 3x3 |
| test3x3_4.txt | 3x3 |

## 5.2 Ambiente Computacional do Experimento

Os algoritmos foram implementados em C++ e compilados com o `g++`. Os testes foram executados em um ambiente compatível com scripts de shell Bash (Linux ou Windows com MSYS2). As especificações das máquinas utilizadas para os testes são:

- **Máquina 1:**
  - **Processador:** Intel Core i7-12700H
  - **Memória RAM:** 40 GB DDR5 4800MHz
  - **Sistema Operacional:** Ubuntu 25.04
- **Máquina 2:**
  - **Processador:** AMD Ryzen 5 5500U
  - **Memória RAM:** 8 GB DDR4 3200MHz
  - **Sistema Operacional:** Manjaro 25.0.4

## 5.3 Metodologia de Execução e Coleta de Dados

Para garantir a execução sistemática e a coleta organizada dos resultados, foi desenvolvido um script de shell, `exec_all.sh`. Este script automatiza o processo de teste da seguinte forma:

1. Itera sobre todos os arquivos de instância no diretório `data/`.
2. Para cada instância, executa todos os 7 algoritmos de busca implementados.
3. Para os algoritmos de busca informada, executa cada um deles com todas as 6 heurísticas.
4. A execução é feita através do programa `npuzzle_exec`, que recebe três argumentos da linha de comando: o caminho da instância, um ID para o algoritmo e um ID para a heurística.
5. A saída de cada execução (contendo as estatísticas de tempo, custo, nós expandidos, etc.) é redirecionada para um arquivo de texto único, organizado em um diretório `output/`, permitindo a posterior análise e comparação dos dados.

Tabela 3: Mapeamento de ID para Algoritmo de Busca.

| **ID** | **Algoritmo** |
|---|---|
| 1 | Backtracking |
| 2 | Busca em Largura (BFS) |
| 3 | Busca Ordenada |
| 4 | Busca em Profundidade (DFS) |
| 5 | Busca Gulosa (Greedy) |
| 6 | Busca IDA* |
| 7 | Busca A* |

Os IDs dos algoritmos e heurísticas passados como argumento para o executável seguem o mapeamento descrito nas Tabelas 3 e 4.

Tabela 4: Mapeamento de ID para Heurística (usado com IDs de algoritmo 5-7).

| **ID** | **Heurística** |
|---|---|
| 1 | Distância de Manhattan |
| 2 | Distância Euclidiana |
| 3 | Peças Fora do Lugar |
| 4 | Conflito Linear |
| 5 | Ciclos de Permutação |
| 6 | Soma Ponderada |

## 5.4 Resultados

Tabela 5: Resultados comparativos dos algoritmos não informados.

| **Instância** | **Algoritmo** | **Custo** | **Nós Exp.** | **Tempo (ms)** |
|---|---|---|---|---|
| test3x3_1 | Backtracking | 40 | 1171775 | 1967.9 |
| | BFS | **18** | 19266 | 48.0 |
| | Order Search | **18** | 19266 | 946.3 |
| | DFS | 38 | 6773 | 10.5 |
| test3x3_2 | Backtracking | 40 | 1116921 | 1870.9 |
| | BFS | **20** | 49475 | 121.8 |
| | Order Search | **20** | 49475 | 5736.5 |
| | DFS | 40 | 67077 | 114.2 |
| test3x3_3 | Backtracking | 40 | 36454 | 63.7 |
| | BFS | **22** | 91302 | 232.1 |
| | Order Search | **22** | 91302 | 15896.7 |
| | DFS | 38 | 62360 | 104.8 |
| test3x3_4 | Backtracking | 39 | 1505079 | 2605.1 |
| | BFS | **25** | 159599 | 486.1 |
| | Order Search | **25** | 159599 | 32563.2 |
| | DFS | 41 | 43714 | 86.9 |

Tabela 6: Resultados da Gulosa com diferentes heurísticas.

| **Instância** | **Heurística** | **Custo** | **Nós Exp.** | **Tempo (ms)** |
|---|---|---|---|---|
| test3x3_1 | Manhattan | 20 | 66 | 0.3 |
| | Euclidean | 20 | 85 | 0.6 |
| | Misplaced Tiles | 44 | 814 | 3.8 |
| | Linear Conflict | **18** | 48 | 0.3 |
| | Permutation Cycles | 52 | 1315 | 13.1 |
| | Weighted Sum | 20 | 100 | 0.4 |
| test3x3_2 | Manhattan | **26** | 56 | 0.2 |
| | Euclidean | 32 | 56 | 0.4 |
| | Misplaced Tiles | 36 | 249 | 1.0 |
| | Linear Conflict | 32 | 47 | 0.4 |
| | Permutation Cycles | 48 | 466 | 3.5 |
| | Weighted Sum | 54 | 371 | 2.2 |
| test3x3_3 | Manhattan | **22** | 37 | 0.2 |
| | Euclidean | **22** | 53 | 0.3 |
| | Misplaced Tiles | 30 | 680 | 3.9 |
| | Linear Conflict | **22** | 35 | 0.3 |
| | Permutation Cycles | 74 | 314 | 3.3 |
| | Weighted Sum | **22** | 56 | 0.3 |
| test3x3_4 | Manhattan | 41 | 225 | 1.1 |
| | Euclidean | 63 | 315 | 1.5 |
| | Misplaced Tiles | 57 | 720 | 3.0 |
| | Linear Conflict | **35** | 58 | 0.4 |
| | Permutation Cycles | 53 | 525 | 5.1 |
| | Weighted Sum | 55 | 358 | 1.9 |

Tabela 7: Resultados do IDA* com diferentes heurísticas.

| **Instância** | **Heurística** | **Custo** | **Nós Exp.** | **Tempo (ms)** |
|---|---|---|---|---|
| test3x3_1 | Manhattan | **18** | 485 | 1.7 |
| | Euclidean | **18** | 9192 | 40.5 |
| | Misplaced Tiles | **18** | 3561 | 8.9 |
| | Linear Conflict | **18** | 247 | 2.1 |
| | Permutation Cycles | **18** | 21591 | 188.0 |
| | Weighted Sum | **18** | 4077 | 11.0 |
| test3x3_2 | Manhattan | **20** | 538 | 1.6 |
| | Euclidean | **20** | 11456 | 49.9 |
| | Misplaced Tiles | **20** | 10111 | 22.8 |
| | Linear Conflict | **20** | 265 | 2.3 |
| | Permutation Cycles | **20** | 57536 | 503.3 |
| | Weighted Sum | **20** | 4431 | 17.4 |
| test3x3_3 | Manhattan | **22** | 1052 | 2.4 |
| | Euclidean | **22** | 44803 | 197.7 |
| | Misplaced Tiles | **22** | 29849 | 57.6 |
| | Linear Conflict | **22** | 627 | 3.4 |
| | Permutation Cycles | **22** | 161796 | 1398.4 |
| | Weighted Sum | **22** | 9673 | 25.8 |
| test3x3_4 | Manhattan | **25** | 9617 | 24.9 |
| | Euclidean | **25** | 373563 | 1595.6 |
| | Misplaced Tiles | **25** | 188244 | 367.6 |
| | Linear Conflict | **25** | 3797 | 20.4 |
| | Permutation Cycles | **25** | 964475 | 9007.5 |
| | Weighted Sum | **25** | 80034 | 225.5 |

Tabela 8: Resultados do A* com diferentes heurísticas.

| **Instância** | **Heurística** | **Custo** | **Nós Exp.** | **Tempo (ms)** |
|---|---|---|---|---|
| test3x3_1 | Manhattan | **18** | 399 | 2.0 |
| | Euclidean | **18** | 451 | 2.8 |
| | Misplaced Tiles | **18** | 1583 | 13.0 |
| | Linear Conflict | **18** | 223 | 1.6 |
| | Permutation Cycles | **18** | 2679 | 25.6 |
| | Weighted Sum | **18** | 457 | 2.5 |
| test3x3_2 | Manhattan | **20** | 416 | 1.6 |
| | Euclidean | **20** | 572 | 4.1 |
| | Misplaced Tiles | **20** | 3630 | 29.2 |
| | Linear Conflict | **20** | 270 | 1.4 |
| | Permutation Cycles | **20** | 6342 | 95.0 |
| | Weighted Sum | **20** | 675 | 3.7 |
| test3x3_3 | Manhattan | **22** | 767 | 4.6 |
| | Euclidean | **22** | 1144 | 8.3 |
| | Misplaced Tiles | **22** | 8951 | 150.0 |
| | Linear Conflict | **22** | 479 | 2.5 |
| | Permutation Cycles | **22** | 15265 | 637.0 |
| | Weighted Sum | **22** | 1387 | 6.5 |
| test3x3_4 | Manhattan | **25** | 4509 | 39.2 |
| | Euclidean | **25** | 6519 | 78.9 |
| | Misplaced Tiles | **25** | 40052 | 4628.9 |
| | Linear Conflict | **25** | 2190 | 24.0 |
| | Permutation Cycles | **25** | 58938 | 10918.8 |
| | Weighted Sum | **25** | 7679 | 126.5 |

# 6 Conclusões e trabalhos futuros

A implementação e análise comparativa de diferentes algoritmos de busca permitiu extrair conclusões valiosas sobre suas características de desempenho e corretude. Os algoritmos de busca informada, especificamente o A* e o IDA*, demonstraram ser as abordagens mais eficientes, apresentando os melhores resultados em termos de tempo de execução para encontrar soluções em tabuleiros complexos. Um dos principais méritos desses algoritmos reside na sua capacidade de garantir a solução ótima, desde que a função heurística utilizada seja admissível, um fator crucial para problemas onde o menor caminho é um requisito.

Observou-se também que a estratégia de Busca Ordenada, ao ser aplicada em um cenário com custo de ação unitário, comportou-se de maneira análoga à Busca em Largura. No entanto, seu desempenho foi inferior, caracterizando-se como uma "Busca em Largura piorada", provavelmente devido à sobrecarga computacional de manter uma estrutura de dados ordenada (como uma lista) em vez de uma simples fila (FIFO), que é otimizada para essa tarefa específica.

Finalmente, uma observação interessante foi que algoritmos com heurísticas não admissíveis, embora não ofereçam garantias teóricas de optimalidade, foram capazes de encontrar a solução ótima para configurações específicas de tabuleiro. Isso evidencia que, na prática, a otimalidade pode ser alcançada por acaso, mas a confiabilidade permanece exclusiva dos métodos admissíveis.

Para trabalhos futuros, sugere-se:

- **Otimização da Estrutura de Dados para Buscas com Custo:** A maior limitação de desempenho nas buscas baseadas em custo (como A* e Ordenada) foi o uso de uma lista simples para armazenar os nós abertos. Uma melhoria prioritária é substituir a lista por uma estrutura de dados mais eficiente, como uma Min-Heap (Pilha de Mínimos). Essa alteração reduzirá a complexidade de tempo para selecionar o próximo nó de menor custo de $O(n)$ para $O(logn)$, resultando em uma aceleração drástica do tempo total de busca.

- **Implementação de um Verificador de Ancestralidade:** Para otimizar ainda mais a exploração da árvore de busca, pretende-se implementar um mecanismo de verificação de ancestralidade. Atualmente, um nó pode gerar seu próprio "pai"como sucessor, criando ciclos curtos e redundantes. Ao impedir que o estado imediatamente anterior seja gerado, é possível podar uma parte significativa de ramos triviais da árvore, reduzindo o número de nós expandidos e, consequentemente, o tempo de busca, sem comprometer a completude ou otimalidade do algoritmo.

- **Análise de Novas Heurísticas:** Explorar e implementar heurísticas mais sofisticadas, como as Pattern Databases, poderia aprimorar ainda mais o desempenho do A* e IDA*, permitindo a resolução de tabuleiros ainda mais complexos em um tempo viável.
- **Otimização de Memória:** Com uma melhoria na representação utilizada, seria possível aumentar os tabuleiros escolhidos para execução e assim ter resultados para tabuleiros de 4x4, 5x5 ou até mesmo maiores, como é o caso do 10x10, que foi separado, mas não foi executado por excesso de memória.

Com essas melhorias, espera-se que os algoritmos possam ser melhorados para a solução do n-puzzle, permitindo uma resolução mais rápida e com custos ótimos, assim como a possibilidade de executar o código para tabuleiros maiores.

# Referências

[1] Repository. https://github.com/Lucas-Henriquee/n-puzzle-explorer-ai, 2025. Acesso em: 11 de Agosto de 2025.

[2] Saulo Moraes Villela. Notas de aula: Inteligência artificial. Material de aula da disciplina, 2025. Material fornecido durante o primeiro semestre.