|  |  |
| --- | --- |
|  | **UNIVERSIDADE DO OESTE PAULISTA**  **CIÊNCIA DA COMPUTAÇÃO** |

**VICTOR HUGO DONAIRE DE OLIVEIRA**

**RELATÓRIO DO 8 PUZZLE**

Presidente Prudente - SP

2025

**INTRODUÇÃO**

O presente trabalho vai apresentar o desenvolvimento de uma aplicação de software para a resolução do quebra-cabeça de 8 peças (8-Puzzle), um problema clássico no campo da Inteligência Artificial e da teoria de algoritmos. O principal objetivo foi criar uma ferramenta visual e interativa capaz de encontrar a sequência ótima de movimentos para solucionar o puzzle a partir de uma configuração inicial arbitrária. Para alcançar este fim, foi implementada uma arquitetura cliente-servidor, com a lógica central de resolução alocada em um backend robusto desenvolvido em Java com o framework Spring Boot. A interface gráfica do usuário foi construída como um cliente web moderno em React (Next.js), que se comunica com o backend para todas as operações lógicas.

A solução do problema é alcançada através da implementação de algoritmos de busca heurística. Foram explorados dois algoritmos distintos: o A\* (A-Estrela), conhecido por garantir a otimalidade da solução, e o Greedy Best-First Search, uma alternativa mais rápida. A eficiência destes algoritmos foi avaliada com o uso de duas funções heurísticas diferentes: a contagem de "Peças Fora do Lugar" e a "Distância de Manhattan". Este relatório detalha a arquitetura do sistema, explica a lógica dos algoritmos implementados e, por fim, apresenta uma análise comparativa de desempenho com base em métricas coletadas pela própria aplicação, como tempo de execução e nós visitados, a fim de determinar a combinação mais eficaz para a resolução do 8-Puzzle.

**DESENVOLVIMENTO**

1. Tela do Aplicativo

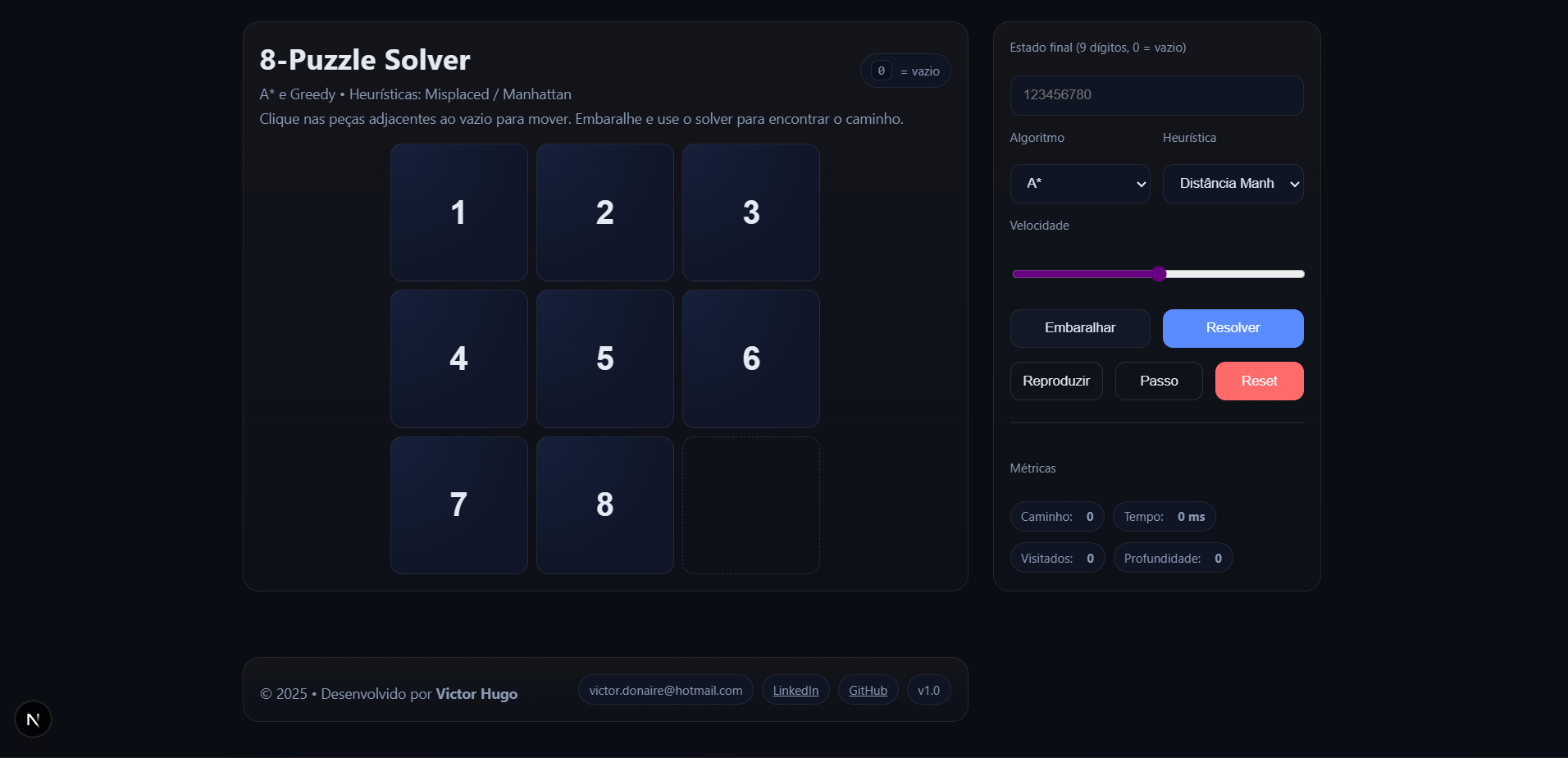


Imagem 1 – Tela inicial do sistema

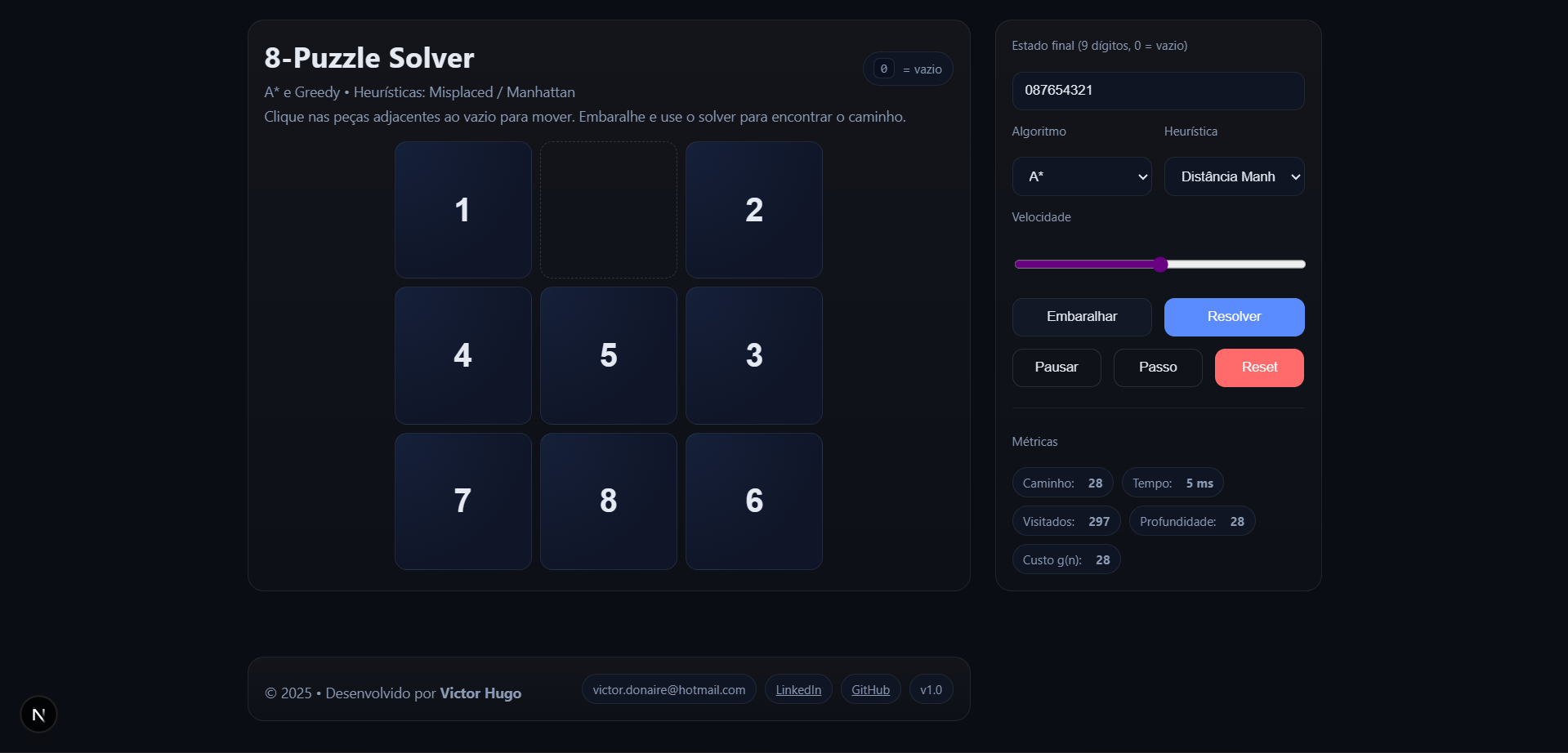


Imagem 2 – Tela Inicial em execução

O presente o trabalho apresenta uma tela totalmente intuitiva, em relação para facilitar o usuário, usar de forma eficaz o sistema, onde assim ao invés de mostrar os passos do tabuleiro, o usuário pode escolher em reproduzir de forma automática o sistema, ou tem a opção de executar passo a passo de acordo com o algoritmo solicitado, e dai ele pode observar como esta sendo feito, e também ali em “Métricas” ele pode observar os dados daquela solicitação efetuada pelo mesmo.

1. Front-end
   1. Explicação

**1. Gerenciamento de Estado (useState)**

No topo do arquivo, você verá várias linhas com useState. Esta é a "memória" do seu componente. Cada useState guarda uma informação importante:

* **state**: É a variável mais importante. Ela guarda a "foto" atual do tabuleiro que o usuário está vendo na tela. Qualquer mudança nela redesenha o tabuleiro.
* **path**: Guarda a "receita" da solução. É a lista com a sequência completa de tabuleiros (passo a passo) que o backend envia após resolver o puzzle.
* **metrics**: Guarda os números de desempenho (tempo, nós visitados) para serem exibidos na tela após a resolução.
* **busy**: É uma "trava" (true/false). Quando está true, os botões são desabilitados para que o usuário não possa clicar em nada enquanto o backend está trabalhando.
* **Outros estados (goal, algo, heur, playing, cursor, speed)**: Guardam as opções que o usuário seleciona nos menus (algoritmo, heurística), o estado da reprodução automática (tocando ou pausada) e a velocidade.

**2. Ações do Usuário (As Funções Principais)**

Existem três funções principais que são acionadas pelos botões e lidam com a comunicação com a API.

* **onShuffle() (Embaralhar):**
  + **O que faz:** Quando o usuário clica em "Embaralhar", esta função é chamada.
  + **Como funciona:** Ela faz uma chamada fetch para o endpoint GET /api/shuffle do seu backend. O backend gera um novo tabuleiro aleatório e o envia de volta. A função então atualiza o estado state com este novo tabuleiro, fazendo com que a tela mostre a nova configuração.
* **move(idx) (Mover Peça Manualmente):**
  + **O que faz:** É chamada quando o usuário clica diretamente em uma peça no tabuleiro.
  + **Como funciona:** Ela envia o tabuleiro atual (state) e o índice da peça clicada (idx) para o endpoint POST /api/move do backend. O backend decide se o movimento é válido e retorna o novo estado do tabuleiro. A função então atualiza o state com a resposta do backend.
* **onSolve() (Resolver):**
  + **O que faz:** A função mais importante, chamada pelo botão "Resolver".
  + **Como funciona:** Ela agrupa todas as informações (state inicial, goal, algo, heur) e as envia para o endpoint POST /api/solve do backend. O backend faz todo o trabalho pesado e, após alguns instantes, retorna a solução completa. A função onSolve recebe essa solução e preenche os estados path (com o passo a passo) and metrics (com as estatísticas), iniciando a reprodução automática.

**3. Lógica Automática e Efeitos (useEffect)**

Existem dois blocos useEffect que cuidam de lógicas que não dependem de um clique direto.

* **O Primeiro useEffect (Busca Inicial):**
  + Este bloco tem um [] no final, o que significa que ele roda **apenas uma vez**, assim que a página carrega.
  + **Sua função:** Chamar a função onShuffle para que o usuário já veja um tabuleiro embaralhado e pronto para jogar assim que abre a aplicação.
* **O Segundo useEffect (Motor de Reprodução do Passo a Passo):**
  + Este é o coração da animação da solução. Ele fica "observando" as variáveis playing e cursor.
  + **Como funciona:** Se playing for true, ele usa setTimeout para criar um timer. Quando o timer dispara, ele atualiza o tabuleiro na tela (setState) com o próximo passo da lista path e avança o cursor. Isso cria o efeito de "filme", mostrando cada passo da solução.

**4. Componentes Visuais (/components)**

O código JSX no return simplesmente organiza a página usando componentes menores e reutilizáveis, passando para eles as informações do useState que eles precisam para se desenhar.

* **<Board />**: Recebe o state e desenha o tabuleiro.
* **<Controls />**: Recebe e modifica estados como algo e heur, e chama as funções como onSolve e onShuffle.
* **<Stats />**: Recebe as metrics e as exibe.

1. Back-end
   1. Explicação

**1. A Arquitetura em Camadas**

O projeto foi organizado em três camadas principais, cada uma com uma responsabilidade clara:

* **Camada de API (api/PuzzleController.java):** Pense nela como a **"recepcionista"**. Sua única função é receber as requisições HTTP do frontend, entender o que ele quer (resolver, embaralhar, etc.), chamar o especialista certo para fazer o trabalho e, por fim, devolver a resposta para o frontend. Ela não sabe como resolver o puzzle, apenas gerencia a comunicação.
* **Camada de Serviço (core/SolverService.java):** Esta é a camada do **"especialista"**, o cérebro de verdade. É aqui que toda a lógica de negócio e os algoritmos complexos estão implementados. Esta camada não sabe nada sobre a internet ou requisições HTTP; ela apenas recebe dados, processa e devolve um resultado.
* **Camada de Dados/DTO (diretorio/):** Esta é a **"linguagem comum"** ou o "contrato" entre o frontend e o backend. Os arquivos aqui (SolveRequest, MoveRequest, etc.) garantem que ambos os lados entendam perfeitamente a estrutura dos dados que estão trocando.

**2. O Coração da Lógica: SolverService.java**

Esta é a classe mais importante e contém toda a inteligência do projeto, dividida em três funcionalidades principais:

1. **solve() (Resolver o Puzzle):**
   * Implementa os algoritmos de busca **A\*** e **Greedy**.
   * Utiliza uma **PriorityQueue** (Fila de Prioridade) como uma "lista de tarefas inteligente", que sempre mantém o estado mais promissor (com o menor custo f) no topo, pronto para ser analisado.
   * Usa um **HashSet** como uma "memória" para guardar todos os estados já visitados, evitando trabalho repetido e loops infinitos.
   * A lógica segue o ciclo: pegar o melhor estado da fila, gerar seus vizinhos, calcular o custo de cada vizinho usando as **heurísticas** (Manhattan ou Peças Fora do Lugar) e adicionar os vizinhos válidos de volta na fila. O processo se repete até a solução ser encontrada.
2. **getShuffledBoard() (Embaralhar):**
   * Para garantir que todo puzzle enviado ao frontend tenha solução, este método começa com o tabuleiro já resolvido.
   * Em seguida, ele simula um grande número de movimentos aleatórios e válidos (ex: 150 movimentos) para embaralhar o tabuleiro de forma realista.
   * O resultado é um tabuleiro aleatório que garantidamente é solucionável.
3. **processMove() (Processar Movimento Manual):**
   * Esta função recebe o estado atual do tabuleiro e o índice da peça que o usuário clicou.
   * Ela consulta uma tabela de movimentos pré-definida (MOVES) para verificar se a peça clicada é vizinha do espaço vazio.
   * Se o movimento for válido, ela realiza a troca e retorna o novo estado do tabuleiro. Se for inválido, retorna o tabuleiro sem alterações.

**3. As Peças de Suporte**

* **PuzzleState.java:** É uma classe de suporte fundamental. Ela funciona como uma "ficha de informações" para cada passo na busca. Cada objeto PuzzleState armazena um estado do tabuleiro, seus custos (g, h, e f), e uma referência ao seu "pai", o que é essencial para conseguir reconstruir o caminho da solução no final.
* **PuzzleController.java:** É o orquestrador que expõe os três métodos do SolverService para o mundo exterior através dos endpoints da API:
  + POST /api/solve: Aciona o método solve().
  + GET /api/shuffle: Aciona o método getShuffledBoard().
  + POST /api/move: Aciona o método processMove().

**CONCLUSÃO**

Com isso, podemos concluir, com o desenvolvimento deste projeto resultou na criação bem-sucedida de uma aplicação funcional para a resolução do 8-Puzzle, validando na prática os conceitos teóricos de busca de Inteligência Artificial. A análise comparativa demonstrou que a heurística da Distância de Manhattan é superior à de Peças Fora do Lugar, e que o algoritmo A\*, apesar de mais metódico, garante a solução ótima que nem sempre é encontrada pelo algoritmo Greedy, mais rápido. Desta forma, a combinação do A\* com a Distância de Manhattan provou ser a estratégia mais eficiente e confiável para encontrar o menor caminho de solução para o problema proposto.

**REFERÊNCIA BIBLIOGRÁFICAS**

* PENMAN, Tristan. **N-Puzzle Solver**. Disponível em: <https://tristanpenman.com/demos/n-puzzle/>. Acesso em: 6 out. 2025.