UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Sara Rosado Rodrigues Muniz

Desenvolvimento de um Sistema com Interface Gráfica integrada a um Algoritmo Genético Aplicado ao Problema do *8-Puzzle*

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Sara Rosado Rodrigues Muniz

Desenvolvimento de um Sistema com Interface Gráfica integrada a um Algoritmo Genético Aplicado ao Problema do *8-Puzzle*

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Prof.ª Dr.ª Christiane Regina Soares Brasil

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2025

Sara Rosado Rodrigues Muniz

Desenvolvimento de um Sistema com Interface Gráfica integrada a um Algoritmo Genético Aplicado ao Problema do *8-Puzzle*

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 19 de setembro de 2025:

Prof.ª Dr.ª Christiane Regina Soares Brasil
Orientadora

Prof. Dr. Bruno Augusto Nassif Travençolo

Prof. Dr. Rafael Dias Araújo

Uberlândia, Brasil
2025

Resumo

Este trabalho apresenta o desenvolvimento de um sistema com uma interface gráfica para o jogo *8-Puzzle*, integrando-a a um Algoritmo Genético (AG) com o objetivo de oferecer uma visualização interativa e didática do processo de resolução. Para isso, foi utilizada a linguagem de programação *Java*, juntamente com as bibliotecas *JavaFX* e *Scene Builder*, estruturadas segundo o padrão *Model-View-Controller* (MVC). A aplicação permite que o usuário configure parâmetros do AG, como taxas de mutação, *crossover*, elitismo, tamanho da população e número de gerações, acompanhando em tempo real as animações do tabuleiro. Foram realizados experimentos com diferentes configurações, e os resultados demonstraram que o AG integrado à interface é capaz de encontrar soluções adequadas com uma quantidade reduzida de movimentos e mantém desempenho uniforme de acordo com a configuração dos parâmetros. A solução desenvolvida atingiu os objetivos propostos, resultando em uma ferramenta modular e interativa, que contribui tanto para o estudo de algoritmos evolutivos quanto para a compreensão prática do *8-Puzzle*.

Palavras-chave: Algoritmos Genéticos. *8-Puzzle*. Interface Gráfica. *JavaFX*.

Abstract

This study presents the development of a system with a graphical interface for the 8-Puzzle game, integrated with a Genetic Algorithm (GA) to provide an interactive and instructive visualization of the resolution process. The implementation employed the Java programming language with the JavaFX and Scene Builder libraries, structured according to the Model-View-Controller (MVC) design pattern. The application allows users to configure GA parameters – including mutation rate, crossover, elitism, population size, and number of generations – while observing real-time board animations. Experiments with different configurations demonstrated that the GA embedded in the interface can efficiently identify solutions with fewer moves, maintaining stable performance across parameter settings. The proposed solution resulted in a modular and interactive tool that supports both the study of evolutionary algorithms and the practical understanding of the 8-Puzzle.

Keywords: *Genetic Algorithms. 8-Puzzle. Graphical User Interface. JavaFX.*

Lista de ilustrações

Figura 1 – **Fluxograma de um Algoritmo Genético.** 14
Figura 2 – **Quebra-cabeça de 8 peças.** 16
Figura 3 – **Modelagem da interface para resolução do *8-Puzzle*.** 26
Figura 4 – **Interface Gráfica para o *8-Puzzle*.** 27
Figura 5 – **Diagrama de fluxo do funcionamento da interface.** 29
Figura 6 – **Interface Gráfica com labels dinâmicos.** 30
Figura 7 – **Resultado da melhor execução do *8-Puzzle* com taxa de *crossover* 100%.** 45
Figura 8 – **Resultado da solução ótima não encontrada de uma das execuções do *8-Puzzle* com taxa de *crossover* 100%.** 46

Lista de tabelas

Tabela 1 – Taxa de mutação = 1%. 37
Tabela 2 – Taxa de mutação = 3%. 37
Tabela 3 – Taxa de mutação = 5%. 38
Tabela 4 – Taxa de mutação = 15%. 38
Tabela 5 – Taxa de mutação = 30%. 39
Tabela 6 – Taxa de elitismo = 1%. 40
Tabela 7 – Taxa de elitismo = 5%. 40
Tabela 8 – Taxa de elitismo = 15%. 41
Tabela 9 – Taxa de elitismo = 20%. 41
Tabela 10 – Taxa de *crossover* = 80%. 42
Tabela 11 – Taxa de *crossover* = 90%. 43
Tabela 12 – Taxa de *crossover* = 95%. 43
Tabela 13 – Taxa de *crossover* = 100%. 44

Lista de abreviaturas e siglas

AGs	Algoritmos Genéticos
AEs	Algoritmos Evolutivos
API	*Application Programming Interface*
CSS	*Cascading Style Sheets*
FXML	*FX Markup Language*
GUI	*Graphical User Interface*
IDE	Ambientes de Desenvolvimento Integrado
IU	Interface do Usuário
MVC	*Model-View-Controller*
UI	*User Interface*

Sumário

1 INTRODUÇÃO 10
1.1 Objetivos 11
1.1.1 Objetivo Geral 11
1.1.2 Objetivo Específico 11
1.2 Justificativa 12
1.3 Organização do texto 12

2 FUNDAMENTAÇÃO TEÓRICA 13
2.1 Algoritmos Evolutivos e Genéticos 13
2.2 O 8-Puzzle 15
2.3 Trabalhos Relacionados 16
2.4 Considerações Finais 18

3 FERRAMENTAS UTILIZADAS PARA INTERFACE GRÁFICA DO JOGO *8-PUZZLE* 20
3.1 Linguagem de Programação Java 20
3.2 JavaFX 22
3.3 NetBeans IDE 23
3.4 JavaFX Scene Builder 23
3.5 Padrão Model-View-Controller (MVC) 24
3.6 Considerações Finais 24

4 DESENVOLVIMENTO 25
4.1 Disposição do Projeto 25
4.2 Construção da Interface Gráfica 26
4.3 Animações, Interatividade e Integração 27
4.3.1 Fluxo da Interface e suas Animações 28
4.3.2 Recursos e Ferramentas 29
4.3.3 Interatividade e *Feedback* Visual da Interface 30
4.3.4 Comunicação entre Algoritmo e Interface 31
4.4 Construção do Algoritmo 31
4.4.1 Organização e Estrutura de Classes 31
4.4.2 Classe PuzzleModel: Modelagem do tabuleiro 32
4.4.3 Classe Cromossomo: Representação do Indivíduo 33
4.4.4 Classe AlgoritmoGenetico: Implementação do Solucionador 34
4.5 Considerações Finais 35

5 RESULTADOS 36
5.1 Experimentos variando a taxa de mutação 36
5.1.1 Análise dos resultados 39
5.2 Experimentos variando a taxa de elitismo 39
5.2.1 Análise de resultados 41
5.3 Experimentos variando a taxa de *crossover* 42
5.3.1 Análise de resultados 46
5.4 Considerações Finais 47

6 CONCLUSÃO 49

REFERÊNCIAS 50

1 Introdução

Os softwares estão amplamente presentes no cotidiano das sociedades contemporâneas, desempenhando um papel fundamental em atividades que variam desde as mais simples — como o uso de uma calculadora digital ou a comunicação entre pessoas em locais distantes — até as mais complexas, como a operação de Sistemas de Gestão Hospitalar ou a automação de estoques empresariais (CARMO, 2017). Nesse contexto, a crescente adoção de dispositivos digitais pelos indivíduos tem intensificado a demanda por interfaces gráficas cada vez mais eficientes, intuitivas e acessíveis (SANTANA et al., 2017).

No âmbito dos jogos digitais, o apelo por interfaces amigáveis torna-se ainda mais relevante, uma vez que interfaces bem projetadas facilitam a compreensão tanto do problema quanto dos objetivos do jogo, por meio de recursos visuais (DANTAS et al., 2013). Nesse sentido, o objetivo deste trabalho foi desenvolver uma interface gráfica simples e interativa para a visualização do processo de busca por soluções do problema do *8-Puzzle*, de forma a estimular o entendimento e a aprendizagem de técnicas de otimização computacional aos interessados no tema.

O *8-Puzzle* é um problema combinatório clássico na área da computação, frequentemente utilizado em estudos relacionados à inteligência artificial e a algoritmos de busca (JUNIOR; GUIMARAES, 2018). O problema consiste na simulação de um quebra-cabeça em um tabuleiro de dimensões 3x3, com 8 peças e um espaço vazio que permite o deslocamento das demais peças. Deste modo, o objetivo do problema é, dado um estado inicial do tabuleiro com as peças de 1 a 8 embaralhadas, encontrar a ordem crescente das peças, movimentando-as por meio do espaço vazio.

Este problema pode ser solucionado por diferentes algoritmos de busca, como o Algoritmo A*, que sempre buscará o melhor caminho para alcançar a solução (YANG et al., 2023), além de algoritmos de busca em profundidade e largura. Neste Trabalho de Conclusão de Curso, o algoritmo utilizado para solucionar o quebra-cabeça de 8 peças foi o Algoritmo Genético (AG), uma vez que apresenta uma grande vantagem em comparação a outras alternativas, pois realiza uma busca abrangente das possíveis soluções, por meio da análise não apenas de um indivíduo, mas de uma população com vários indivíduos (JUNIOR; GUIMARAES, 2018).

A fim de visualizar o resultado obtido pelo AG, foi necessário desenvolver uma interface para o mesmo. Esta interface foi elaborada para facilitar a compreensão do AG aplicado ao problema *8-Puzzle*, proporcionando a interatividade do usuário com o algoritmo, uma vez que o usuário tem a possibilidade de realizar os ajustes aos parâmetros dos operadores genéticos. Com a visualização da evolução do jogo, pretende-se promover a

aprendizagem sobre esta técnica de otimização, mediante a resposta em tempo real em relação ao deslocamento das peças e possíveis soluções para o quebra-cabeça, possibilitando que o usuário analise os melhores valores para cada parâmetro a partir dos resultados obtidos.

As tecnologias que foram utilizadas neste trabalho incluem a linguagem *Java*, com o ambiente de desenvolvimento integrado *NetBeans IDE*, a biblioteca de interface *JavaFX* e a ferramenta de design *JavaFX Scene Builder*. Com o uso destas tecnologias, foi possível alcançar o objetivo principal do trabalho, que foi desenvolver uma interface gráfica para a iteração do usuário na escolha de parâmetros e operadores do AG, e a visualização das soluções do *8-Puzzle*, obtidas por um Algoritmo Genético.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste Trabalho de Conclusão de Curso foi desenvolver um sistema computacional para o jogo *8-Puzzle*, focando em uma interface gráfica capaz de oferecer uma melhor visualização e o entendimento da resolução do problema do *8-Puzzle* com um algoritmo de otimização, mais especificamente o Algoritmo Genético.A proposta da interface visa permitir que os usuários compreendam, de forma visual e interativa, o processo de busca pela solução do problema, ao mesmo tempo em que facilita o estudo de um Algoritmo Genético (AG) aplicado a esse contexto específico. Além disso, os usuários podem ajustar os parâmetros dos operadores genéticos, tornando a experiência de aprendizagem mais dinâmica e auxiliando na análise dos resultados obtidos pelo algoritmo.

1.1.2 Objetivo Específico

Os objetivos específicos deste Trabalho de Conclusão de Curso foram:

- Investigar as tecnologias, como *JavaFX e NetBeans IDE*, e suas funcionalidades disponíveis para o desenvolvimento de interfaces gráficas;
- Realizar um estudo sobre o problema do *8-Puzzle*, analisando suas principais características e estratégias de resolução;
- Explorar e buscar entender os conceitos envolvendo Algoritmos Genéticos (AGs), com o foco em como aplica-los na resolução do *8-Puzzle*.

1.2 Justificativa

Problemas como o *8-Puzzle* são amplamente utilizados para testar e comparar algoritmos de busca, sendo aplicados em problemas reais como: planejamento de rotas (GPS, robótica, drones), inteligência artificial em jogos, planejamento automatizado (logística, manufatura, etc.), movimentação de robôs em espaços restritos, entre outros. Embora o *8-Puzzle* seja um quebra-cabeça simples em sua definição, sua resolução sem uma representação visual pode não ser trivial para iniciantes. Em razão disto, a análise visual do problema não apenas facilita a compreensão dos processos de busca por possíveis soluções, mas também enriquece o aprendizado do algoritmo utilizado, permitindo que os usuários interajam ativamente na escolha dos parâmetros e operadores. Esta interatividade torna o estudo mais dinâmico e envolvente, promovendo uma compreensão clara do funcionamento do AG, com a possibilidade de estender a aplicação para outros algoritmos em trabalhos futuros.

1.3 Organização do texto

Este trabalho foi estruturado da seguinte maneira:

- Capítulo 1 – **Introdução**: apresenta o tema, a justificativa, os objetivos e a contextualização do problema abordado.
- Capítulo 2 - **Fundamentação teórica**: reúne os principais conceitos e teorias envolvendo Algoritmos Genéticos, e discorre os principais trabalhos relacionados que servem de base para o desenvolvimento do estudo e implementação da interface.
- Capítulo 3 - **Ferramentas utilizadas**: descreve os procedimentos, técnicas e ferramentas utilizadas no desenvolvimento da pesquisa.
- Capítulo 4 - **Desenvolvimento**: detalha implementação da solução proposta, incluindo a modelagem, a arquitetura e o funcionamento do sistema, tanto da interface quanto do algoritmo.
- Capítulo 5 - **Resultados**: apresenta os experimentos realizados, a análise dos dados obtidos e a avaliação da solução proposta.
- Capítulo 6 - **Conclusão**: relata as conclusões obtidas, as contribuições do trabalho e sugestões para trabalhos futuros.

Esta estruturação foi adotada de forma a garantir clareza e fluidez na apresentação da pesquisa. A divisão em capítulos permite que o leitor acompanhe de maneira sequencial desde a contextualização teórica até a implementação e análise dos resultados, garantindo uma compreensão mais objetiva do trabalho como um todo.

2 Fundamentação Teórica

Este capítulo abordará os tópicos essenciais para a elaboração deste Trabalho de Conclusão de Curso, tais como a definição de Algoritmos Evolutivos e Genéticos, a descrição do problema *8-Puzzle*, bem como uma coletânea de alguns trabalhos relacionados à interface gráfica aplicada a jogos bidimensionais.

Primeiramente, será apresentada a fundamentação teórica de Algoritmos Evolutivos, focando em Algoritmos Genéticos.

2.1 Algoritmos Evolutivos e Genéticos

Os Algoritmos Evolutivos (AEs) são métodos de otimização e, segundo Jong (2006), estão diretamente relacionados à Teoria da Evolução de Charles Darwin, uma vez que são inspirados nos princípios da seleção natural e da evolução biológica descritos em sua obra *A Origem das Espécies* (1859). De forma geral, os AEs utilizam mecanismos como adaptação, reprodução e sobrevivência dos mais aptos para buscar soluções em espaços complexos de problemas. Dentro desta abordagem evolutiva encontram-se diferentes formas de utiliza-la, como estratégias evolutivas, programação genética e os algoritmos genéticos.

Os Algoritmos Genéticos (AGs), por sua vez, representam uma das técnicas mais conhecidas e aplicadas dos algoritmos evolutivos. O conceito de AGs foi estruturado e proposto pelo cientista John Henry Holland e seus alunos na década de 1970 (GABRIEL; DELBEM, 2008), com o objetivo de alcançar soluções para um problema de maneira eficiente. Sua principal diferença quando comparado a um AE é o fato de exigir a reprodução (cruzamento) para gerar descendentes, o que o deixa mais fiél aos conceitos da genética. Por outro lado, os algoritmos evolutivos dão liberdade para gerar novo individuo de outras maneiras, sem ser necessariamente através do *crossover*.

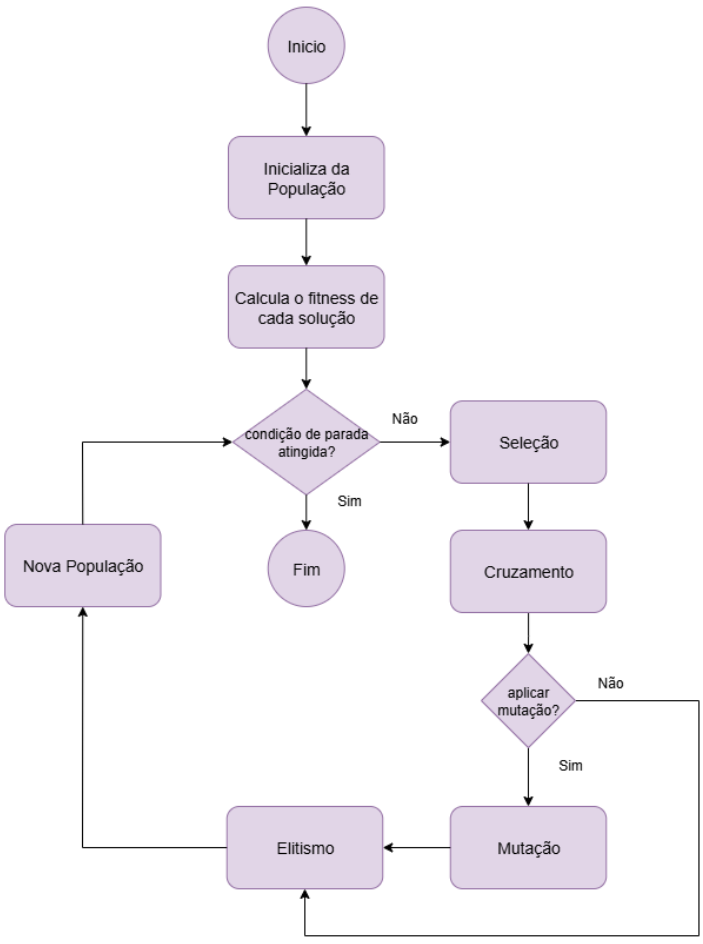
Segundo Jong (2006), os sistemas evolutivos e, consequentemente, os algoritmos genéticos são constituídos pelos seguintes componentes:

- **População** - A população é um conjunto de indivíduos, que representam as possíveis resoluções para um problema.
- **Função de Aptidão** - Está relacionada com a capacidade de um indivíduo em sobreviver e se reproduzir. A função de aptidão é responsável por avaliar as soluções candidatas presentes na população, por meio de um valor (*fitness*) que medirá a adequação.

- **Operadores Genéticos** - Os operadores genéticos são responsáveis por promover variabilidade e hereditariedade no conjunto de soluções. Para que isso ocorra, são aplicados os operadores de recombinação *(crossover)* e mutação, além dos métodos de seleção, que representam o processo de seleção natural (elitismo).

A Figura 1 ilustra o comportamento típico de um AG, evidenciando seu ciclo de execução. O processo inicia-se com a **inicialização da população**, onde um conjunto de cromossomos, ou seja, indivíduos é gerado. Em seguida, realiza-se o cálculo do *fitness*, etapa em que se avalia a aptidão de cada solução candidata em relação ao problema. Caso alguma delas atenda aos critérios desejados, o processo de busca pode ser encerrado; caso contrário, aplicam-se os **operadores genéticos** — como seleção, cruzamento e mutação — sobre a população. Os indivíduos resultantes, denominados descendentes, são então selecionados para compor a nova geração por meio do elitismo. Esse ciclo repete-se de forma iterativa até que uma solução satisfatória seja encontrada ou que um **critério de parada** previamente definido seja atingido.

Figura 1 – **Fluxograma de um Algoritmo Genético.**



Fonte: Elaborado pela autora.

Quanto aos operadores genéticos utilizados nos AGs, são mecanismos essenciais para a evolução da população ao longo das gerações. Os três mais encontrados na implementação desses algoritmos são:

- **Cruzamento** - O operador de cruzamento, do inglês *crossover* e também conhecido como recombinação, realiza a combinação de características de dois ou mais indivíduos para criar descendentes, promovendo a exploração do espaço de busca. Assim, o objetivo do cruzamento é produzir filhos com características potencialmente melhores que as dos pais (GOLDBERG, 1989)
- **Mutação** - A mutação introduz pequenas alterações aleatórias em um indivíduo, garantindo diversidade e evitando a convergência prematura. Logo, este operador corresponde à chance de um gene sofrer alteração durante o processo evolutivo. Geralmente, essa probabilidade tende a ser baixa, pois taxas muito elevadas tendem a comprometer a qualidade das soluções, produzindo indivíduos menos adaptados que seus pais (GABRIEL; DELBEM, 2008).
- **Elitismo** - O elitismo, por sua vez, é um método de seleção que assegura que uma parcela dos melhores indivíduos seja preservada entre as gerações, mantendo soluções de alta qualidade no processo evolutivo (GABRIEL; DELBEM, 2008).

Entender os principais conceitos envolvendo Algoritmos Genéticos é fundamental para contextualizar o presente trabalho e compreender o método de resolução escolhido para o problema do quebra-cabeça de 8 peças.

A seguir, será explicado o jogo *8-Puzzle* utilizado neste trabalho.

2.2 O 8-Puzzle

O *8-Puzzle* é um jogo de tabuleiro em forma de quebra-cabeça que, segundo Junior e Guimaraes (2018), está fortemente presente na área de Inteligência Artificial, uma vez que jogos de tabuleiro representam um problema combinatório de alta complexidade. Isto significa que, à medida que a dimensão do tabuleiro aumenta, a dificuldade em encontrar uma solução ótima também cresce, tornando um relevante objeto de estudo nesta área de otimização computacional.

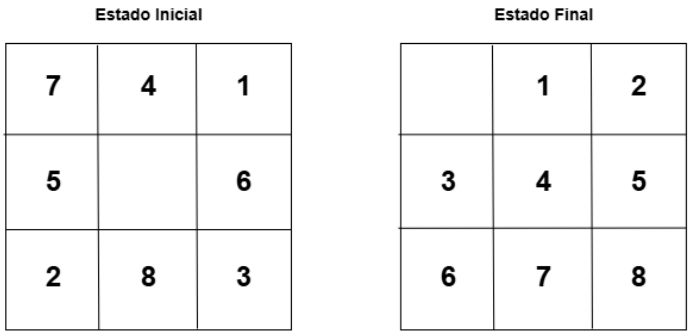
O objetivo do jogo é, a partir de um estado inicial embaralhado, mover as peças utilizando o espaço vazio até que todas fiquem organizadas em ordem crescente, formando a configuração de 0 a 8, em que o 0 representa o espaço vazio localizado na primeira posição. Essa definição corresponde ao estado final adotado neste trabalho.

O problema pode ser resolvido com a aplicação de diversos algoritmos de otimização , como o Algoritmo A*, *Backtracking* e Algoritmos Genéticos. Nesse contexto,

os Algoritmos Genéticos se destacam como uma excelente opção para a resolução do *8-Puzzle*, pois, segundo Bhasin e Singla (2012), eles são conhecidos por serem mais eficientes que algoritmos randômicos, devido à sua capacidade de lidar com problemas complexos de forma rápida. Deste modo, os AGs se apresentam como uma abordagem poderosa para a resolução do *8-Puzzle*, aproveitando estratégias de otimização como a utilização de operadores genéticos.

Neste trabalho, o tabuleiro foi utilizado com uma dimensão 3x3, contendo 8 peças e um espaço vazio, como mostrado na Figura 2.

Figura 2 – **Quebra-cabeça de 8 peças.**



Fonte: Elaborado pela autora.

A seguir, serão discutidos trabalhos relacionados que abordam o desenvolvimento de interfaces gráficas aplicadas a jogos bidimensionais, destacando abordagens e ferramentas utilizadas para atingir o objetivo.

2.3 Trabalhos Relacionados

Nesta seção, serão apresentados trabalhos que tiveram como um de seus propósitos o desenvolvimento de uma interface gráfica, visando alcançar um objetivo específico, demonstrar e justificar um determinado argumento. Tais trabalhos foram escolhidos devido à semelhança com a proposta do projeto em questão.

1. DIGIAMPIETRI e KROPIWIEC (2008)

   Nesse artigo, o objetivo foi solucionar as problemáticas de aprendizagem dos alunos nos cursos de computação por meio de um ambiente de desenvolvimento de jogos digitais. Para concretizar este ambiente, foi desenvolvido um servidor de jogos implementado em *Java*, no qual os jogos comunicam-se com ele via mensagens *SOAP*. Além disso, foram criadas aplicações como *Deflexion*, *Multi-Jogadora*, *Bots Jogadores de Jogo da Velha*, *Newmings*, *Gerador de Sudoku* e *Jogos de Prolog*, que interagem com o servidor.

2. MEDINA e MÜLLER (2009)

   Neste projeto, foi desenvolvida uma interface gráfica de um jogo de tabuleiro de damas utilizando a biblioteca gráfica *Allegro*, na linguagem de programação *C++*. O objetivo foi elaborar um jogo educativo empregando *Algoritmos Genéticos* para buscar os melhores lances que solucionem o jogo. Os resultados obtidos foram satisfatórios, demonstrando que os *Algoritmos Genéticos* são excelentes para problemas que envolvem agentes inteligentes, onde o jogador humano não consegue prever a jogada de seu adversário.

3. DANTAS et al. (2013)

   Nesse trabalho, foi proposto o desenvolvimento de um jogo lúdico que apoie o processo de aprendizagem dos estudantes de programação. Neste sentido, foi criado um jogo denominado *Robotimov* utilizando a ferramenta *Unity*, uma engine de desenvolvimento de jogos que permite criar jogos 2D e 3D de forma versátil e facilitada.

4. ANTAS, SOUTO e VALENTIM (2015)

   Este artigo teve como objetivo propor uma interface adaptável para jogos eletrônicos, com a finalidade de proporcionar uma experiência diferenciada para pessoas com deficiência de mobilidade ou sensorial. Neste sentido, para legitimar o projeto, foi desenvolvido um jogo de dama e um *Blackjack* na linguagem *C#* e fazendo uso da *namespace Speech*, de forma que o usuário possa realizar movimentos no jogo por *clicks* ou comandos de voz.

5. RISSETTI, MACHADO e MIRANDA (2017)

   Nesse artigo, foram abordadas as dificuldades vivenciadas na área da informática, envolvendo o aprendizado de programação. Assim, o trabalho consistiu em desenvolver uma *API* (*Application Programming Interface*) para a implementação de Jogos 2D, por meio da computação gráfica e de ferramentas como o *JavaFX*, a fim de auxiliar no desenvolvimento do raciocínio lógico dos estudantes da área de tecnologia e auxiliar no aprendizado.

6. SANTANA et al. (2017)

   Esse trabalho teve como objetivo o desenvolvimento de um jogo de tabuleiro digital denominado "*Blinds, Basic Education (BBE)*", semelhante a um jogo da velha, com o intuito de auxiliar pessoas com deficiência visual no processo de aprendizagem da programação. Assim, para a concretização desse projeto, foram usadas as ferramentas *JavaFX*, para a criação da interface gráfica, a linguagem de programação *Java*, além do *MBROLA* e *Cloud Garden* para a sintetização de voz.

7. RIBEIRO (2018)

Nesse trabalho, RIBEIRO (2018) desenvolveu uma ferramenta educativa que auxilia no processamento auditivo e alfabetização de crianças. Assim, o principal objetivo foi a criação de uma interface gráfica para um jogo de memória auditiva, responsável por auxiliar na melhora da memória de trabalho e do processamento auditivo, fatores relacionados com a capacidade de alfabetização.

Para a implementação do software, foram utilizadas ferramentas como a linguagem de programação *Java*, a biblioteca *JavaFX*, o framework *JavaFX Scene Builder*, a IDE *NetBeans*, a plataforma de versionamento *GitHub*, entre outras.

8. SILVA (2020)

   Nesse artigo, foi abordado um problema de otimização denominado "caixeiro viajante". O objetivo principal envolveu a utilização de *algoritmos genéticos* e a proposta de duas soluções para resolver essa problemática. A primeira abrangeu o desenvolvimento de um aplicativo *mobile* capaz de marcar rotas em um mapa, e a segunda referiu-se à criação de um serviço web para executar o *algoritmo genético*.

9. FARIAS et al. (2022)

   Nesse artigo, o objetivo principal foi a criação de um software denominado "*MOTRIZ*", capaz de auxiliar idosos a melhorarem suas habilidades motoras e cognitivas. Assim, para implantar uma aplicação desktop capaz de disponibilizar jogos que ajudariam a alcançar tal objetivo, foi desenvolvida uma interface utilizando a ferramenta *JavaFX Scene Builder*, na versão 1.1, juntamente com a linguagem *Java*.

Este Trabalho de Conclusão de Curso teve como principais referências para o desenvolvimento da interface os trabalhos de RISSETTI, MACHADO e MIRANDA (2017), RIBEIRO (2018) e FARIAS et al. (2022), os quais contribuíram para o entendimento das metodologias e das ferramentas aplicáveis à criação de interfaces gráficas em jogos bidimensionais.

2.4 Considerações Finais

Este capítulo apresentou os principais conceitos teóricos que fundamentam o desenvolvimento deste trabalho, abordando o problema do *8-Puzzle*, os Algoritmos Genéticos (AGs) como método de resolução e trabalhos relacionados ao desenvolvimento de interface gráfica voltada para jogos bidimensionais. O *8-Puzzle* foi caracterizado como um problema clássico de busca em Inteligência Artificial, cuja solução pode ser otimizada por meio de técnicas evolutivas, como os AGs.

Em relação aos AGs, foram destacados seus componentes essenciais, como população, função de aptidão e operadores genéticos, além de seu funcionamento iterativo, que

simula o processo de seleção natural de Charles Darwin. Paralelamente a isto, a análise de trabalhos relacionados demonstrou a viabilidade do uso de interfaces gráficas em jogos e sistemas educativos, reforçando a relevância da proposta deste projeto.

Desta forma, este capítulo não apenas justifica a escolha metodológica adotada, como também fornece recursos e sugestões para a implementação prática da solução, que será detalhada nos capítulos seguintes.

3 Ferramentas utilizadas para Interface Gráfica do jogo *8-Puzzle*

Neste capítulo, são apresentadas as ferramentas utilizadas no desenvolvimento do sistema de busca por soluções para o problema do *8-Puzzle*, com base nos trabalhos de RISSETTI, MACHADO e MIRANDA (2017), RIBEIRO (2018) e FARIAS et al. (2022), os quais contribuíram para o entendimento das tecnologias aplicadas à elaboração de interfaces gráficas para jogos bidimensionais.

Primeiramente, é descrita a linguagem de programação utilizada neste Trabalho de Conclusão de Curso, seguida pelo detalhamento das ferramentas.

3.1 Linguagem de Programação Java

Java[1] é uma linguagem de programação orientada a objetos que atualmente é aplicada em diversas áreas da computação, além de ser conhecida por sua portabilidade, segurança e robustez. Esta linguagem foi criada pela *Sun Microsystems* na década de 1990 e derivou-se de linguagens como *C* e *C++*. Por esse motivo, *Java* possui uma sintaxe familiar, mas, ao mesmo tempo, caracteriza-se por ser mais simples do que as citadas anteriormente, já que não envolve características complexas que podem impactar na curva de aprendizado, como o uso de ponteiros (ORACLE, 1996).

Para o desenvolvimento de programas em *Java*, é de boa prática fazer uso de Ambientes de Desenvolvimento Integrado (*IDEs*), que são responsáveis por fornecer ferramentas e suporte para o desenvolvimento de software. Entre as IDEs mais conhecidas estão *NetBeans*, *Eclipse*[2] e *IntelliJ IDEA*[3].

Neste trabalho, a linguagem *Java* foi escolhida justamente por sua simplicidade e ampla comunidade de suporte que é oferecida, além de possuir uma grande variedade de bibliotecas e frameworks que facilitam o desenvolvimento de interfaces gráficas, assim como o *Swing* e o *JavaFX*. Junto a isso, aplicações desenvolvidas em *Java* possuem alta portabilidade, permitindo sua execução em diferentes sistemas operacionais. Essa característica, combinada com a facilidade de integração com diversas ferramentas de design e desenvolvimento, torna *Java* uma escolha versátil e eficaz para a criação de interfaces gráficas, como a proposta no presente projeto.

[1] https://www.oracle.com/br/java/
[2] https://eclipseide.org/
[3] https://www.jetbrains.com/idea/

A linguagem *Java* pode ser implementada em quatro diferentes plataformas, também conhecidas como edições *Java*, de acordo com a necessidade de cada tipo específico de desenvolvimento de aplicação. Entre essas plataformas descritas nos documentos oficiais da empresa *Oracle* estão:

- **Java SE (Standard Edition)** - É a plataforma base do *Java*, onde sua *API* (*Application Programming Interface*) é responsável por fornecer as principais funcionalidades da linguagem *Java* e também por definir tipos, objetos, classes e bibliotecas usadas para o desenvolvimento de interfaces gráficas, aplicações de rede, acesso a banco de dados, entre outras.

- **Java EE (Enterprise Edition)** - Esta plataforma foi construída com base na edição *Java SE*, o que significa que o *Java EE* também utiliza as funcionalidades básicas e bibliotecas disponibilizadas pelo *Java SE*. Ademais, a plataforma *Enterprise Edition* foi desenvolvida com o objetivo de oferecer recursos de desenvolvimento para aplicações de rede de grande escala, capazes de promover escalabilidade, confiabilidade e segurança.

- **JavaFX** - É uma biblioteca utilizada para desenvolver interfaces gráficas avançadas e interativas, porém também é considerada uma plataforma *Java*, uma vez que o *JavaFX* é responsável por fornecer uma vasta possibilidade de ferramentas para a criação de aplicações *RIA* (*Rich Internet Applications*). Assim, essa plataforma é uma grande auxiliadora durante a criação de *GUIs* (*Graphical User Interfaces*) que demandam gráficos, animações, multimídia, entre outras.

- **Java ME (Micro Edition)** - A plataforma *Micro Edition* fornece uma *API* de baixo consumo de recursos para a execução de softwares desenvolvidos na linguagem *Java*, especialmente voltados para aplicações móveis que serão executadas em dispositivos de pequeno porte, como telefones celulares.

Certamente, cada plataforma possui suas especificidades para que funcione de acordo com a necessidade do desenvolvedor. Por outro lado, algo em comum entre as quatro plataformas anteriormente descritas é o fato de todas apresentarem uma *JVM* (*Java Virtual Machine*) encarregada por simular um hardware específico, possibilitando que a aplicação rode em diferentes sistemas operacionais. Neste trabalho, a fim de atingir o objetivo principal do projeto, será utilizada a plataforma *Java SE* na versão 8, juntamente com o *JavaFX*.

3.2 JavaFX

Originalmente, o *Swing* foi considerado a principal biblioteca para a criação de GUIs em linguagem de programação *Java*. No entanto, em 2008, a *Sun Microsystems*, posteriormente adquirida pela *Oracle*, lançou o *JavaFX*[4] que, segundo DEITEL e DEITEL (2016), seria "a ferramenta do futuro" para a criação de interfaces em *Java*. Os autores destacam que o *JavaFX* oferece vantagens em relação ao *Swing*, como maior usabilidade, ferramentas gráficas e de multimídia mais avançadas, ampla comunidade de suporte, além de ter sido projetado para melhorar a segurança de *threads*.

Neste contexto, o *JavaFX* foi escolhido como a biblioteca base para auxiliar no desenvolvimento deste trabalho, não apenas pelas vantagens previamente mencionadas, mas também pelos benefícios adicionais que oferece para projetos que envolvem a criação de interfaces gráficas, como componentes de *User Interface (UI)*, estilização por meio de *Cascading Style Sheets (CSS)* e a possibilidade de integração com ferramentas de design, como o *Scene Builder*.

Em seu livro, DEITEL e DEITEL (2016) descreve um pouco da estrutura e de alguns componentes da biblioteca *JavaFX* para um entendimento inicial da ferramenta:

- **Stage** - Também conhecido como palco, é uma classe pertencente ao pacote *javafx.stage* responsável por representar a janela em que a interface é exposta. Nessa janela serão exibidos elementos como botões, gráficos e componentes de interface.
- **Scene** - É uma classe do *JavaFX* que pertence ao pacote *javafx.scene*, encarregada de representar o conteúdo visual dentro de uma janela *Stage*. Logo, a *Scene* (cena) atua como um contêiner para os itens gráficos que compõem a GUI, como caixas de texto, botões e painéis.
- **Node** - Presente no pacote *javafx.scene*, *Node* é a classe base para todos os elementos gráficos em *JavaFX*. Cada elemento gráfico da interface é representado por um nó dentro de uma *Scene*. Esses nós formam o grafo de cena, que organiza todos os componentes visuais da GUI de maneira hierárquica. O grafo de cena começa com um nó raiz, que se ramifica em nós filhos, possibilitando a inclusão e a organização dos componentes visuais da interface.

A biblioteca JavaFX inclui muitas outras classes e componentes que ajudam no desenvolvimento de interfaces além das mencionadas anteriormente. Contudo, ao considerar as classes *Stage*, *Scene* e *Node*, é possível ter uma visão geral de como iniciar e estruturar o projeto.

[4] https://www.oracle.com/java/technologies/javase/javafx-docs.html

3.3 NetBeans IDE

O *NetBeans IDE*[5] é um ambiente de desenvolvimento integrado (IDE) de código aberto e gratuito, amplamente utilizado por desenvolvedores para criar, editar, compilar e executar códigos em diversas linguagens, como *Java*, *PHP*, *C++*, e *JavaScript*. De acordo com RIBEIRO (2018), essa IDE oferece uma apresentação de código satisfatória no que se refere à qualidade dos recursos visuais e estruturais, além de possuir uma excelente integração com a linguagem *Java*. Por essas razões, o *NetBeans* foi escolhido como a ferramenta ideal para o desenvolvimento da interface do *8-Puzzle*.

3.4 JavaFX Scene Builder

O *JavaFX Scene Builder*[6] é uma ferramenta de desenvolvimento visual para interfaces gráficas em *JavaFX* que se integra facilmente ao *NetBeans IDE*. Essa tecnologia permite que os desenvolvedores criem interfaces de maneira intuitiva, arrastando e soltando componentes visuais na área de design sem escrever nenhuma linha de código, além de oferecer personalização e estilização por meio de propriedades CSS.

Em seu trabalho, focado no desenvolvimento de um jogo para auxiliar no processamento auditivo e na alfabetização de crianças, RIBEIRO (2018) ressalta algumas vantagens do *JavaFX Scene Builder* na criação de interfaces gráficas. Entre elas, destaca-se a geração automática de arquivos FXML (*FX Markup Language*) à medida que a interface é criada e modificada, o que facilita a identificação, o entendimento e a alteração do código. Além disso, a possibilidade de visualização em tempo real da interface é extremamente valiosa durante o desenvolvimento de um software, uma vez que permite ajustes e melhorias antes da integração com o código *Java*, sem haver a necessidade de compilar e executar a aplicação.

CARMO (2017), que visou desenvolver em seu trabalho de conclusão de curso dois softwares com a mesma interface, porém utilizando duas bibliotecas de interface gráfica distintas (*JavaFX* e *Swing*), reforça que o fato de o *Scene Builder* gerar automaticamente um código FXML facilita o desenvolvimento da GUI ao ajudar na organização da codificação, visto que ele é encarregado de separar a parte visual do software da parte lógica, ou seja, do código *Java*.

Portanto, considerando que o uso de ferramentas como o *Scene Builder* pode ajudar significativamente no processo de desenvolvimento, permitindo que o desenvolvedor crie a interface de forma visual e, em seguida, gere automaticamente o código necessário, facilitando a construção e os testes, essa ferramenta foi escolhida como suporte na materialização da GUI proposta no presente Trabalho de Conclusão de Curso.

[5] https://netbeans.org/
[6] https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html

3.5 Padrão Model-View-Controller (MVC)

Quanto a uma forma de padronizar e organizar o *software*, foi escolhido o padrão de projeto *Model-View-Controller (MVC)*, que segundo Gamma (2009), surgiu com o intuito de aumentar a flexibilidade e o reuso da aplicação ao fragmentá-la em três componentes principais:

- **Model** - Representa o objeto de aplicação, sendo responsável pela manipulação dos dados e pela lógica do *software*, sem se preocupar com a interface.
- **View** - Corresponde à parte visual, exibindo as informações ao usuário.
- **Controller** - Atua como mediador entre os dois outros objetos, gerenciando eventos e definindo como a interface reage às ações do usuário.

Para compreender o funcionamento do padrão e como ele foi implementado na estrutura do projeto, será apresentada uma explicação mais detalhada na seção "Disposição do Projeto"do capítulo de desenvolvimento.

3.6 Considerações Finais

Este capítulo apresentou as ferramentas e metodologias utilizadas no desenvolvimento da interface gráfica para o *8-Puzzle*. A linguagem *Java* foi escolhida por sua portabilidade e robustez, enquanto a biblioteca *JavaFX* destacou-se como ideal para a criação da GUI, oferecendo recursos avançados de animação e interatividade.

Para agilizar e dar suporte ao longo do desenvolvimento, foram utilizadas ferramentas como o *NetBeans IDE* e o *JavaFX Scene Builder*, que facilitaram a construção visual da interface por meio de arquivos FXML. Além disso, o padrão MVC foi adotado para organizar o código, separando a lógica do jogo e o controle de interações.

Essas escolhas garantiram um desenvolvimento eficiente, resultando em uma aplicação modular, de fácil manutenção e expansão, que será detalhada nos capítulos seguintes.

4 Desenvolvimento

Este capítulo apresenta o processo de desenvolvimento da interface do jogo *8-Puzzle*, utilizando o Algoritmo Genético como solucionador do problema. São detalhados a organização da interface, o funcionamento do sistema e as etapas de implementação, com o objetivo de oferecer uma visão clara e abrangente da construção técnica realizada. Além disso, discute-se a elaboração do AG aplicado à busca por soluções do jogo em questão.

A interface foi implementada com recursos da biblioteca *JavaFX*, e o código desenvolvido encontra-se disponível em: <https://github.com/sararrmuniz/Interface_8_Puzzle>. Cabe ressaltar que a implementação do algoritmo foi realizada com base no trabalho de conclusão de curso intitulado "Algoritmo Genético aplicado ao problema do *8-Puzzle*", desenvolvido pela aluna Laura Rosado Rodrigues Muniz, graduanda do curso de Sistemas de Informação na Universidade Federal de Uberlândia (UFU). O código correspondente ao Algoritmo Genético utilizado como referência encontra-se disponível em: <https://github.com/laurarrmuniz/8_puzzle>.

4.1 Disposição do Projeto

Devido à escolha das ferramentas mencionadas no Capítulo 3, a interface gráfica para o *puzzle* foi estruturada e organizada em pacotes seguindo o padrão MVC (Model-View-Controller), visando facilitar a manutenção e uma possível expansão do sistema em ocasiões futuras. Os principais pacotes são:

- **puzzle.model** - É composto pelas classes *PuzzleModel.java*, *AlgoritmoGenetico.java* e *Cromossomo.java*. O componente modelo contém toda a lógica e regras do jogo, como embaralhar e trocar as peças, além de gerenciar o estado do tabuleiro e implementar o algoritmo genético para resolver o *puzzle*. Tudo isto ocorre independente da interface.

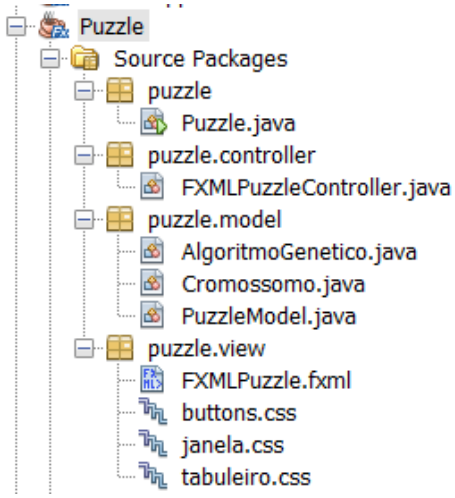
- **puzzle.view** - Define o layout da interface gráfica, sendo ela composta pelo arquivo *FXMLPuzzle.fxml* e os arquivos CSS para estilização. Na *View*, não é desenvolvida nenhuma lógica de aplicação, recebendo apenas interações do usuário e repassando-as ao *controller*.

- **puzzle.controller** - É composto pela classe *FXMLPuzzleController.java*, responsável por gerenciar animações, transições e eventos, como clique de botões, além de

intermediar a comunicação entre a interface e a lógica do jogo ao atualizar a *View* com dados do *Model*.

- **puzzle** - Contém a classe principal *Puzzle.java*, cuja sua única função é inicializar a aplicação.

Figura 3 – **Modelagem da interface para resolução do *8-Puzzle*.**



Fonte: Elaborado pela autora.

Esta organização permite um desenvolvimento mais modular, ou seja, o programa é estruturado de forma organizada e dividido em partes independentes, facilitando a manutenção e dando possibilidade de adicionar novas funcionalidades ou modificar o *software* sem comprometer sua estrutura.

4.2 Construção da Interface Gráfica

A estruturação gráfica da interface foi construída utilizando a linguagem de marcação FXML, muito presente em aplicações *JavaFX*, capaz de separar a lógica visual do código *Java*. O *layout* principal foi composto por um nó raiz do tipo *StackPane*, que é um contêiner de *layout* que organiza os outros nós (DEITEL; DEITEL, 2016).

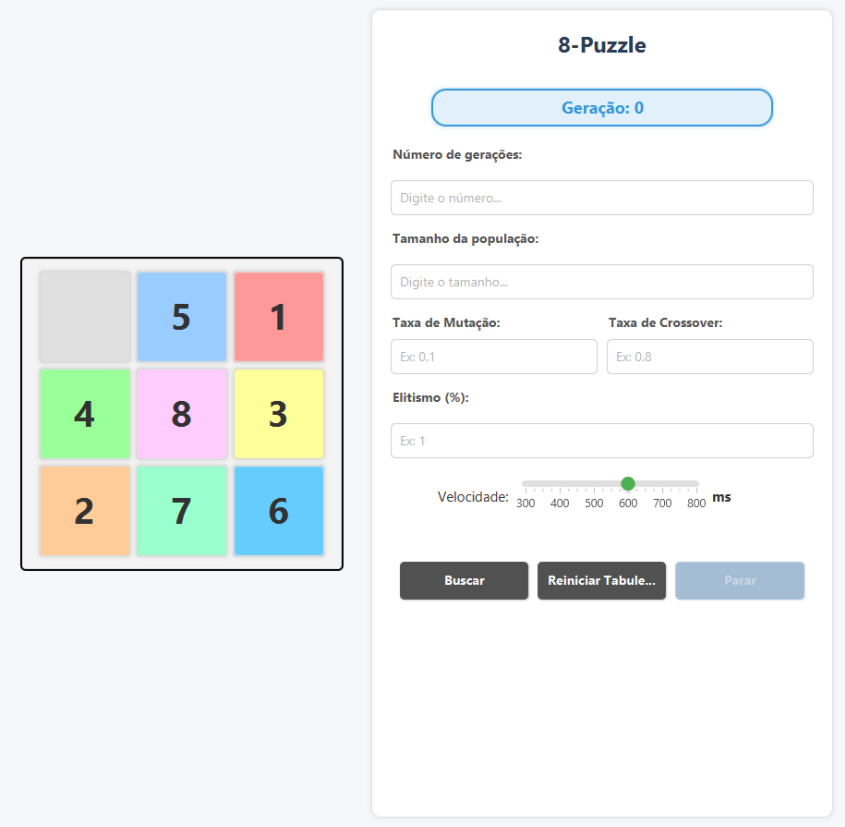
- **Tabuleiro do *8-Puzzle*** - Representado por um *GridPane* centralizado. Dentro deste contêiner, cada célula do tabuleiro é retratada por um *Pane* estilizado, que contém um *Label* indicando o número da peça correspondente. As posições das peças são organizadas em uma matriz 3x3, com espaçamento entre as células e estilos visuais aplicados via arquivos CSS.

- **Painel de controle lateral** - Organizado em um contêiner vertical (*VBox*) e localizado à direita do tabuleiro. Esta área de controle contém: um título *Label* com o

nome do jogo, um contador de gerações abaixo deste título, um conjunto de campos de entrada do tipo *TextField* para que o usuário informe os parâmetros do algoritmo genético, três botões responsáveis por reiniciar o tabuleiro, iniciar e parar a busca, um rótulo de mensagem *messageLabel*, utilizado para exibir avisos ou informações sobre o andamento da busca, e um *Slider* que permite o usuário ajustar a velocidade da animação.

Este contêiner encapsula um outro, denominado *Hbox*, que engloba dois elementos centrais (Figura 4):

Figura 4 – **Interface Gráfica para o *8-Puzzle*.**



Fonte: Elaborado pela autora.

A aplicação de estilos foi realizada por meio de três arquivos CSS de nome *tabuleiro.css*, *janela.css*, *buttons.css*, responsáveis por definir características como cores, tamanhos e alinhamento dos componentes visuais. Esta separação permite uma construção visual da interface consistente, limpa e organizada.

4.3 Animações, Interatividade e Integração

Esta seção descreve como os aspectos de interatividade, animações e integração direta da interface com o Algoritmo Genético foram desenvolvidas com o intuito de tornar

o processo de resolução do *8-Puzzle* mais claro, atrativo e didático.

O objetivo é apresentar, de forma organizada, o fluxo completo de funcionamento da interface — desde a inserção dos parâmetros pelo usuário até a execução das animações que ilustram cada passo da busca, destacando os recursos técnicos empregados e as estratégias adotadas para manter a aplicação responsiva e intuitiva.

4.3.1 Fluxo da Interface e suas Animações

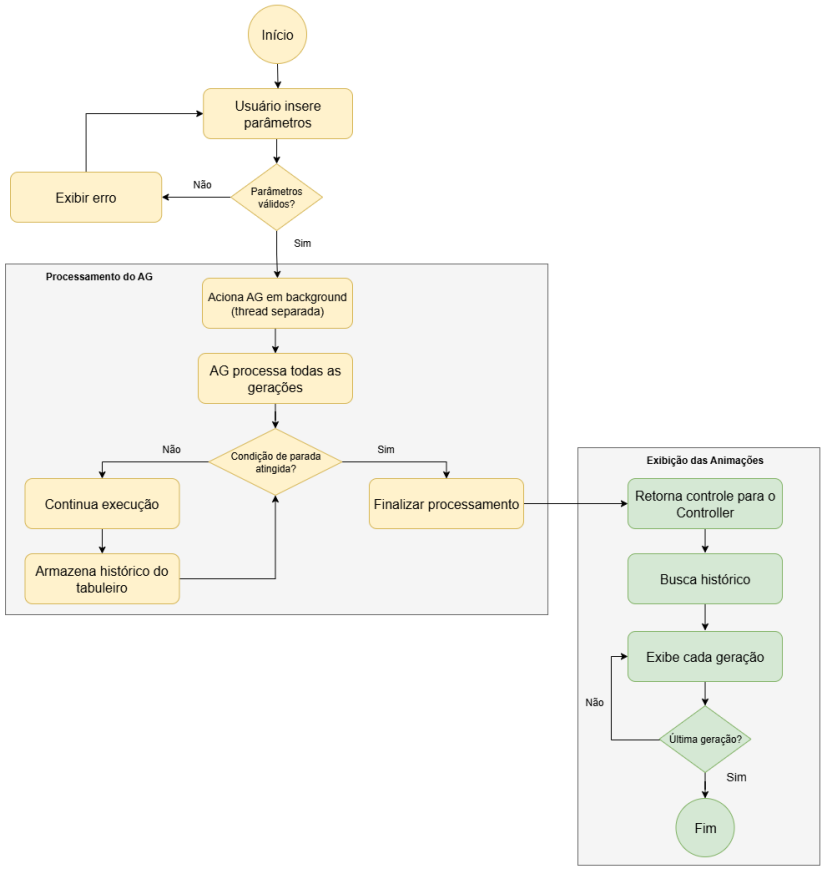
O funcionamento da interface segue um fluxo sequencial que integra a interação do usuário com a execução do algoritmo e a exibição animada dos resultados. Este processo inicia-se no painel lateral, onde o usuário insere os parâmetros de execução, como número de gerações, tamanho da população, taxa de mutação, *crossover* e elitismo. Após a validação destes valores, o botão de busca aciona, no *Controller*, a criação de uma *Thread* dedicada para processamento do AG, garantindo que a *JavaFX Application Thread* permaneça responsável apenas pela atualização visual e interação com o usuário.

Quando a busca é acionada e o AG retorna a melhor solução da geração atual, ou seja, uma sequência de movimentos válidos, o fluxo da animação é iniciado. Essa sequência é enviada ao *Controller*, que agenda cada movimento conforme a velocidade definida pelo usuário no *slider*. Em seguida, as peças são deslocadas visualmente no tabuleiro por meio de transições, enquanto o modelo interno do jogo é atualizado.

A interface, por sua vez, exibe em tempo real informações como geração, número de movimentos e mensagens de status. Esse processo se repete até que todos os movimentos sejam executados, avançando então para a próxima geração ou exibindo o resultado final.

A seguir, na Figura 5, é apresentado o fluxograma do funcionamento da interface quanto ao algoritmo e a exibição e das animações.

Figura 5 – **Diagrama de fluxo do funcionamento da interface.**



Fonte: Elaborado pela autora.

4.3.2 Recursos e Ferramentas

O sistema de animações implementado para esta interface foi pensado com o objetivo de promover uma experiência agradável para o usuário visualizar o processo de resolução do *8-Puzzle* pelo AG, promovendo *feedback* visual em tempo real e interatividade entre sistema e usuário. Por este motivo, as animações foram implementadas para demonstrar a troca de movimento e deslizamento entre as peças de forma suave e agradável.

Quanto ao seu desenvolvimento, o uso de bibliotecas e recursos nativos do *JavaFX* foi essencial. O sistema de animações foi implementado utilizando *Timeline* com *KeyFrame* para controlar o movimento suave das peças, enquanto efeitos visuais complementares como *fade-in* foram criados através de *FadeTransition*.

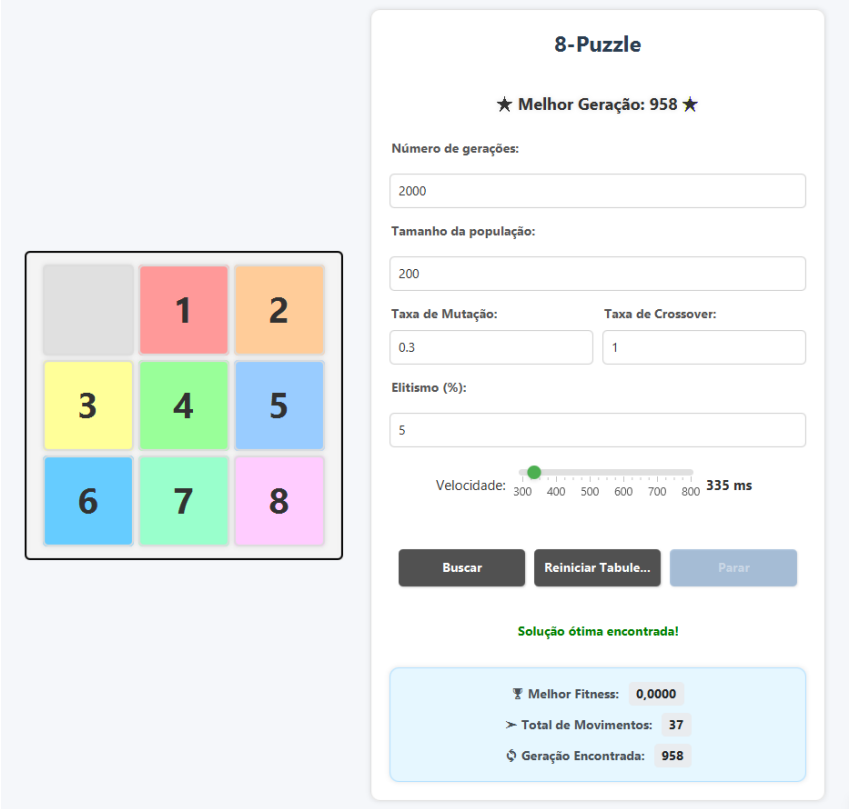
Para garantir que as animações ocorressem em sequência, a utilização do recurso *ScheduledExecutorService* foi de suma importância, tornando possível agendar cada movimento em intervalos regulares. Esse sistema evita sobreposições e garante que o tabuleiro seja atualizado corretamente após cada passo.

4.3.3 Interatividade e *Feedback* Visual da Interface

Quanto à sua interatividade, a interface foi desenvolvida com um painel lateral contendo controles interativos, como campos para ajustar parâmetros (tamanho da população, taxas de mutação, *crossover* e elitismo) e botões para iniciar, parar e reiniciar a busca. Estas características compõem um dos pontos mais fortes da GUI, uma vez que permitem o diálogo direto entre o AG e o usuário, contribuindo para o melhor entendimento do funcionamento do algoritmo, enfatizando a importância do ajuste de parâmetros.

O *feedback* visual da GUI foi projetado com a intenção de promover informações em tempo real para facilitar a compreensão do comportamento do AG ao buscar uma solução. Para isso, foram implementados *Labels* dinâmicos que, a cada iteração, exibem a geração atual, o número de movimentos e se o movimento processado naquele exato momento está sendo pulado, caso ele seja inválido. Ademais, ao final da busca é exibido um contâiner com o melhor fitness, número de movimentos e número da geração onde os melhores resultados foram encontrados. Acima deste container, é exibida uma mensagem responsável por indicar se o AG encontrou a solução ótima ou não, como ilustrado na Figura 6.

Figura 6 – **Interface Gráfica com labels dinâmicos.**



Fonte: Elaborado pela autora.

4.3.4 Comunicação entre Algoritmo e Interface

Para uma boa performance de toda a interface, a arquitetura implementada utiliza um modelo de múltiplas *threads* onde há uma *Thread Principal (JavaFX Application Thread)*, responsável pelo processamento da interface gráfica e animações, e uma *Worker Thread*, encarregada de executar o algoritmo genético em segundo plano.

A comunicação entre *threads* foi desenvolvida através do *Platform.runLater()* para atualizações seguras na GUI, variáveis *volatile* para sincronização de estado entre threads e *callbacks*, que permitem que o algoritmo notifique a interface sobre o progresso sem gerar grandes interrupções durante o processamento do algoritmo e das animações. Esta abordagem arquitetural permite que a interface mantenha sua responsividade mesmo durante processamentos intensivos, enquanto garante a sincronização entre a execução do algoritmo e sua representação visual.

Todos estes componentes foram utilizados com o propósito de tornar a visualização mais dinâmica, educativa e atraente ao usuário, proporcionando não apenas uma ferramenta funcional para resolução do problema, mas também um ambiente de aprendizado sobre o funcionamento de algoritmos genéticos aplicados a problemas de busca.

A combinação desses elementos resultou em uma interface com animações fluídas e chamativas aos olhos do usuário, alinhando eficiência computacional com usabilidade, cumprindo assim os objetivos de demonstrar claramente o processo evolutivo do algoritmo enquanto mantém uma experiência interativa e intuitiva.

Conclui-se, assim, que o desenvolvimento do sistema atendeu aos objetivos propostos, resultando em uma aplicação modular, interativa e responsiva, capaz de integrar de forma eficiente o AG à interface gráfica, proporcionando ao usuário uma experiência visual clara e dinâmica do processo de resolução do *8-Puzzle*.

4.4 Construção do Algoritmo

Por trás da interface desenvolvida neste trabalho encontra-se o algoritmo responsável pelo processamento da solução. O Algoritmo Genético exerce influência direta sobre a implementação da interface gráfica. Isso ocorre porque, para que a GUI funcione adequadamente, é necessário compreender, implementar e integrá-la ao algoritmo. Deste modo, esta seção tem como objetivo apresentar a organização, a implementação e o funcionamento do AG que sustenta a interface gráfica.

4.4.1 Organização e Estrutura de Classes

O AG foi implementado em *Java* e, conforme descrito na Subseção 4.1, foi estruturado segundo o padrão arquitetural MVC. Desta maneira, seus principais componentes

foram desenvolvidos na camada *Model*, onde se concentra a lógica do sistema. As classes responsáveis por compor a implementação do AG são:

- **PuzzleModel**: Representa o estado do tabuleiro e fornece métodos para embaralhar, verificar solucionabilidade e calcular a distância de Manhattan.
- **Cromossomo**: Representa uma solução candidata (sequência de movimentos) e inclui métodos para cálculo de *fitness*, aplicação de movimentos e operadores genéticos.
- **AlgoritmoGenetico**: Coordena a evolução da população, aplicando seleção, *crossover*, mutação e elitismo.

Essa organização promove uma estrutura modular, favorecendo a reutilização e a manutenção do código. Além de facilitar a integração com a interface gráfica, ela também permite futuras expansões, como a substituição do AG por outras técnicas de busca, sem comprometer a arquitetura geral do sistema.

4.4.2 Classe PuzzleModel: Modelagem do tabuleiro

Responsável pela representação do estado do quebra-cabeça de oito peças e pela implementação das operações fundamentais relacionadas à manipulação do tabuleiro, a classe *PuzzleModel* encapsula a lógica central do jogo. Entre suas funcionalidades estão a verificação da solucionabilidade e o cálculo de métricas essenciais para a avaliação de estados.

O tabuleiro é modelado por meio de uma matriz bidimensional 3×3 de inteiros (`int[][] tabuleiro`), onde o valor `0` representa o espaço vazio e os valores de `1` a `8` correspondem às peças numeradas. O estado objetivo é definido pela variável estática `SOLUCAO`, conforme ilustrado a seguir:

```
public class PuzzleModel {
    private int[][] tabuleiro;
    private static final int[][] SOLUCAO = {
        {0, 1, 2},
        {3, 4, 5},
        {6, 7, 8}
    };
}
```

Embora seja inicialmente representado como uma matriz bidimensional, o tabuleiro pode ser convertido para um formato unidimensional. Para isso, os métodos `toArray1D()`

e `fromArray1D()` permitem alternar entre as representações, facilitando a integração com a interface gráfica e com o AG.

A classe também implementa quatro métodos responsáveis pela movimentação do espaço vazio: `moverParaCima()`, `moverParaBaixo()`, `moverParaEsquerda()` e `moverParaDireita()`. Em conjunto com outros métodos auxiliares, voltados para a validação e a permutação das células, esses métodos possibilitam a execução das operações de movimento.

Outro método relevante para o funcionamento do AG é o `embaralhar()`, responsável por gerar estados iniciais válidos a partir da execução de 10 a 100 movimentos aleatórios sobre o estado resolvido. Essa abordagem assegura que todos os estados gerados sejam alcançáveis a partir da configuração objetivo e, portanto, solucionáveis.

Por fim, destaca-se o cálculo heurístico implementado no método *calcularDistanciaManhattan()*, que determina a distância real de cada peça até sua posição correta no estado objetivo. Esse valor é utilizado na avaliação de *fitness* dos cromossomos, sendo fundamental para orientar a busca do algoritmo genético em direção à solução.

4.4.3 Classe Cromossomo: Representação do Indivíduo

A classe *Cromossomo* tem como objetivo representar um indivíduo dentro da população do algoritmo genético. Essa representação é feita por meio de uma sequência de movimentos, que define como o tabuleiro evolui a partir de um estado inicial aleatório.

Cada cromossomo é composto por diferentes elementos que caracterizam sua estrutura genética. O atributo `tabuleiroInicial` armazena o estado inicial do quebra-cabeça, representado por um objeto da classe *PuzzleModel*. A sequência de movimentos é registrada no atributo `movimentos`, definido como uma lista de cadeias de caracteres correspondentes às ações possíveis (*"cima"*, *"baixo"*, *"esquerda"* e *"direita"*). Além disso, o cromossomo mantém os atributos `fitness`, que indica a qualidade da solução; `distancia`, correspondente à distância de Manhattan do estado resultante; e `custoMovimentos`, que atua como penalidade proporcional ao comprimento da sequência.

O processo de inicialização dos cromossomos ocorre por meio da geração de sequências aleatórias de 15 a 40 movimentos. Ademais, uma constante de penalização de 0.0001 é aplicada ao cálculo de *fitness* com o objetivo de favorecer soluções mais curtas e minimizar a tendência de soluções com movimentos redundantes.

A avaliação dos indivíduos é realizada pelo método `calcularFitness()`, cuja fórmula combina a distância de Manhattan obtida após a aplicação dos movimentos com o custo associado ao comprimento da sequência. Portanto, o valor de aptidão é calculado

da seguinte maneira:

$$fitness = distancia + (0.0001 \times \text{número de movimentos}) \tag{4.1}$$

Um valor de *fitness* igual a zero indica que a sequência representa uma solução ótima para o problema.

Para promover a diversidade da população, foram implementados dois operadores genéticos. A mutação atua de duas formas distintas, com igual probabilidade: inserindo um novo movimento no final da sequência ou alterando um movimento já existente. Ambos os casos evitam a ocorrência de movimentos opostos consecutivos por meio do método `isMovimentoOposto()`. O *crossover*, por sua vez, é realizado através de um operador de ponto único, no qual metade dos genes de cada progenitor é combinada. Nesse processo, dá-se preferência ao progenitor com melhor valor de *fitness*, garantindo maior influência de indivíduos mais adaptados.

Por fim, o método `aplicarMovimentos()` é responsável por executar a sequência de ações sobre o estado inicial do tabuleiro, utilizando os métodos de movimentação definidos na classe *PuzzleModel*. O resultado é o estado final alcançado pelo cromossomo, que serve de base para o cálculo da aptidão e para a comparação entre diferentes soluções dentro do processo evolutivo.

4.4.4 Classe AlgoritmoGenetico: Implementação do Solucionador

A classe *AlgoritmoGenetico* gerencia a população de cromossomos e evolui as melhores soluções. Logo, ela implementa o núcleo do algoritmo evolutivo, orquestrando o processo de busca pela solução ótima do *puzzle* através de operações genéticas sobre uma população de cromossomos.

O algoritmo dispõe de parâmetros configuráveis que influenciam diretamente o comportamento da evolução. A taxa de mutação indica a probabilidade de ocorrência de alterações em cada indivíduo da população. A taxa de *crossover* representa a chance de recombinação entre pares de cromossomos selecionados. A taxa de elitismo garante que uma pequena parcela da população mais adaptada seja preservada a cada geração, assegurando a manutenção de soluções de alta qualidade. Estes parâmetros podem ser customizados pelo usuário através da GUI, permitindo experimentação com diferentes configurações e adaptação do algoritmo a diversos cenários de resolução do problema.

A função principal da classe é implementada no método `resolver()`, que segue uma sequência bem definida de etapas. Inicialmente, é gerada a população de cromossomos, com tamanho especificado pelo parâmetro `tamanhoPopulacao`. Em seguida, realiza-se a avaliação do *fitness* de cada indivíduo, calculado de acordo com a Equação 4.1. Caso algum cromossomo apresente valor de *fitness* igual a zero, o processo é interrompido ime-

diatamente, pois significa que a solução ótima foi encontrada. Se isso não ocorrer, aplica-se a seleção por roleta, onde os indivíduos são escolhidos como pais de acordo com uma probabilidade proporcional ao inverso de seu *fitness*. Sobre os pais selecionados incidem as operações de recombinação, mutação e elitismo, retornando então ao ciclo de avaliação até que um dos critérios de parada seja satisfeito.

Este método de seleção pro roleta garante que indivíduos mais aptos tenham maior chance de serem escolhidos, sem descartar completamente as soluções menos adaptadas, preservando assim a diversidade genética da população.

Outro aspecto importante da implementação é a comunicação com a interface gráfica. Por meio do *callback* `atualizacaoUI`, o algoritmo envia o histórico de estados do melhor indivíduo de cada geração, permitindo a visualização em tempo real da evolução da solução. Essa comunicação é realizada pelo método `enviarParaUI()`, que reconstrói o histórico ao aplicar, passo a passo, a sequência de movimentos do cromossomo.

O processo evolutivo é interrompido quando um dos critérios de parada é atendido: a obtenção de uma solução ótima, ou seja, *fitness* igual a zero, o alcance do número máximo de gerações ou a recepção de um sinal de interrupção do usuário durante a execução. Além disso, a classe mantém o registro do melhor indivíduo encontrado ao longo de todas as gerações, armazenado em `melhorGlobal`, e da geração correspondente, em `geracaoEncontrada`. Esses dados fornecem métricas relevantes para a análise do desempenho do algoritmo, permitindo avaliar tanto a qualidade quanto a eficiência do processo de busca.

4.5 Considerações Finais

Este capítulo apresentou o desenvolvimento e a estrutura da GUI para o problema clássico do *8-Puzzle* usando AGs.

Deste modo, a estrutura seguiu o padrão MVC, separando lógica, visualização e controle. A interface foi criada com o auxílio da biblioteca *JavaFX*, contendo um tabuleiro interativo e um painel lateral para ajuste de parâmetros e controle da execução. As animações suaves foram implementadas para mostrar os movimentos das peças, enquanto o sistema de múltiplas *threads* garantiu certa fluidez evitando que a interface não travasse durante o processamento.

O resultado foi uma aplicação funcional e intuitiva, que permite visualizar a resolução do *puzzle* de forma clara e interativa, demonstrando na prática o funcionamento dos algoritmos genéticos.

5 Resultados

Este capítulo apresenta os resultados obtidos a partir dos experimentos realizados com o sistema aplicado ao problema do *8-Puzzle*, utilizando o Algoritmo Genético como solucionador. Os testes foram conduzidos em um computador *Dell Inspiron 5590*, equipado com 16 GB de memória RAM e sistema operacional *Windows*.

Para avaliar o desempenho do AG na resolução do *8-Puzzle* junto à interface, foram realizados diversos testes considerando diferentes parâmetros de configuração. O número máximo de gerações foi fixado em 2000, enquanto o tamanho da população utilizado nos experimentos foi de 200 indivíduos, ambos definidos empiricamente. O experimento foi dividido em três etapas: testes com taxa de mutação, testes com taxa de elitismo e testes com taxa de *crossover*. Cada configuração foi executada 10 vezes para garantir a consistência dos resultados, e o estado inicial dos tabuleiros foi gerado aleatoriamente em cada execução.

Durante os testes, foram registrados dados como a geração em que o melhor valor de *fitness* foi alcançado e o tamanho da solução encontrada, possibilitando uma análise detalhada do desempenho do algoritmo em diferentes cenários. Para facilitar a compreensão, os resultados coletados foram organizados em tabelas. Cabe destacar que, quando indicado o número de gerações igual a 2001, isso significa que a solução ótima não foi encontrada em nenhuma das 2000 gerações executadas. Adicionalmente no último experimento, estão imagens da solução final do tabuleiro com a melhor execução desta configuração e também um caso de insucesso, a fim de mostrar as duas possibilidades de visualização dos resultados.

5.1 Experimentos variando a taxa de mutação

As tabelas a seguir são responsáveis por demonstrar os resultados encontrados para as variações de 1%, 5%, 15% e 30% na taxa de mutação. Os demais parâmetros (taxa de *crossover* e elitismo) foram mantidos fixos nesta etapa com os valores de 100% e 1%, respectivamente, neste experimento.

Tabela 1 – Taxa de mutação = 1%.

Execução	**Geração**	**Tamanho**
1	2001	15
2	6	26
3	2001	24
4	41	38
5	1498	40
6	2001	33
7	4	37
8	2001	35
9	6	16
10	2	36
Média	**260**	**30**
Desvio Padrão	**554**	**9**

Para a taxa de mutação igual a 1% (Tabela 1), o AG não se mostrou capaz de encontrar a solução ótima em quatro execuções. Por outro lado, conseguiu encontrá-la nas outras 6 execuções. O número médio de gerações em que a solução final foi encontrada com sucesso foi de 260, com desvio padrão de 554. O tamanho médio do cromossomo para as 10 execuções foi de 30 movimentos, enquanto o desvio padrão foi de 9.

A execução 10 obteve o melhor desempenho neste experimento, alcançando a solução ótima na segunda geração.

Tabela 2 – Taxa de mutação = 3%.

Execução	**Geração**	**Tamanho**
1	2001	15
2	1	40
3	1151	26
4	2001	31
5	2001	38
6	2	31
7	2001	26
8	21	38
9	369	35
10	2001	22
Média	**309**	**30**
Desvio Padrão	**444**	**8**

Para a taxa de mutação de 3%, o algoritmo não encontrou a solução para o *puzzle* em 5 das 10 execuções totais. O número médio de gerações em que a solução final foi encontrada com sucesso foi de 309, com desvio padrão de 444. O tamanho médio do cromossomo para as 10 execuções foi de 30 movimentos, enquanto o desvio padrão foi de 8. A execução 2 obteve o melhor desempenho neste experimento, alcançando a solução ótima na primeira geração.

Tabela 3 – Taxa de mutação = 5%.

Execução	**Geração**	**Tamanho**
1	2001	36
2	2001	42
3	2001	26
4	6	26
5	4	34
6	34	39
7	34	42
8	2	16
9	20	41
10	4	22
Média	**16**	**32**
Desvio Padrão	**13**	**9**

Na Tabela 3, percebe-se que na solução ótima foi encontrada em 7 execuções, e falhou em resolver o problema em 3. A média de gerações foi de 16 e desvio padrão de 13. A média de movimentos em cada cromossomo foi de 32 e desvio padrão de 9. A execução 8 obteve o melhor desempenho neste experimento, alcançando a solução ótima na segunda geração.

Tabela 4 – Taxa de mutação = 15%.

Execução	**Geração**	**Tamanho**
1	16	43
2	2001	37
3	2001	23
4	3	20
5	1	28
6	1	22
7	1	38
8	2	28
9	2	37
10	2	36
Média	**4**	**32**
Desvio Padrão	**5**	**7**

Para a taxa de mutação de 15% (Tabela 4), o AG não obteve sucesso em 2 das execuções, apresentando um comportamento semelhante ao observado para a taxa de 1%. As soluções foram encontradas, em média, na 4ª geração, com desvio padrão populacional de 5. O tamanho médio dos indivíduos gerados foi de 32 movimentos. De modo geral, tanto o número médio de movimentos quanto os valores de *fitness* obtidos superaram os resultados alcançados com a taxa de 1%.

As execuções 5, 6 e 7 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima logo na primeira geração.

Tabela 5 – Taxa de mutação = 30%.

Execução	**Geração**	**Tamanho**
1	2001	24
2	20	42
3	29	30
4	1080	84
5	30	44
6	2001	36
7	2001	37
8	68	44
9	2001	17
10	13	42
Média	**207**	**40**
Desvio Padrão	**391**	**17**

Para mutação igual a 30%, o algoritmo se mostrou bem sucedido em 6 execuções, com desvio padrão populacional médio de 17 movimentos e média populacional de 40. Quanto ao número médio de gerações até chegar a solução, foi de 207 e desvio padrão de 391 gerações. A execução 10 obteve o melhor desempenho neste experimento, alcançando a solução ótima na 13º geração.

5.1.1 Análise dos resultados

Os resultados obtidos evidenciaram que a taxa de mutação exerce forte influência no desempenho do AG. As taxas muito baixas, como 1% e 3%, mostraram maior dificuldade em explorar o espaço de busca, resultando em mais execuções sem solução, além de precisarem de um número médio maior de gerações. Por outro lado, valores intermediários, especialmente 15%, proporcionaram maior diversidade populacional, favorecendo a convergência rápida e a obtenção de soluções mais curtas em termos de número de movimentos.

Ao observar taxas elevadas, como 30%, embora garantam certo nível de diversidade, também podem gerar instabilidade na convergência, o que reflete em maior variabilidade nos resultados e em soluções com cromossomos mais longos. De forma geral, considerando a qualidade das soluções e o número de execuções bem-sucedidas, e comparando as 5 diferentes configurações para os experimentos com foco na taxa de mutação, a taxa de 15% apresentou o desempenho mais consistente, atingindo a solução ótima em 8 das 10 execuções realizadas, sem comprometer a eficiência do processo evolutivo.

5.2 Experimentos variando a taxa de elitismo

Nos experimentos realizados com foco na taxa de elitismo, foram testadas 4 configurações variáveis: 1%, 5%, 15% e 20% de elitismo. Quanto aos demais parâmetros, permaneceram estáticos para as quatro configurações testadas, com os valores de 15% de

mutação, 100% de taxa de *crossover*, população inicial de 200 movimentos e 200 gerações. Nas Tabelas 6, 7, 8 e 9 estão contidos os resultados destes testes.

Tabela 6 – Taxa de elitismo = 1%.

Execução	**Geração**	**Tamanho**
1	54	32
2	2001	29
3	2001	25
4	6	23
5	11	31
6	2001	33
7	53	42
8	191	18
9	2001	18
10	1136	36
Média	**242**	**32,9**
Desvio Padrão	**405**	**16**

Para a taxa de elitismo de 1%, o AG mostrou dificuldade em encontrar a solução em 4 execuções. Por outro lado, o algoritmo encontrou a solução ideal nas outras 6 rodadas. O desvio padrão geracional foi de 405 e média de gerações de 242. O tamanho médio da população foi de 32,9 movimentos e desvio padrão populacional de 16. A execução 4 obteve o melhor desempenho neste experimento, alcançando a solução ótima na sexta geração.

Tabela 7 – Taxa de elitismo = 5%.

Execução	**Geração**	**Tamanho**
1	86	34
2	5	26
3	19	30
4	4	17
5	5	34
6	2001	32
7	2001	16
8	5	24
9	2001	29
10	3	16
Média	**18**	**25**
Desvio Padrão	**28**	**6,88**

Na Tabela 7, percebe-se que, para a taxa de elitismo de 5%, o AG mostrou bons resultados, alcançando a solução em 7 das 10 execuções, com número médio de gerações de 18 e desvio padrão geracional de 28. A média para o tamanho do cromossomo foi de 25 movimentos e desvio padrão populacional de 6,88. A execução 10 obteve o melhor desempenho neste experimento, alcançando a solução ótima na terceira geração.

Tabela 8 – Taxa de elitismo = 15%.

Execução	**Geração**	**Tamanho**
1	2001	43
2	2001	30
3	2	30
4	5	15
5	5	15
6	4	30
7	29	41
8	2001	15
9	2001	29
10	2	41
Média	**9**	**28**
Desvio Padrão	**10**	**10,65**

Para a taxa de elitismo igual a 15% (Tabela 8), o AG encontrou a solução ótima do problema em 6 das 10 execuções realizadas. O número médio de gerações necessárias foi igual a 9, com desvio padrão populacional de 10,65. O tamanho médio encontrado para os indivíduos foi de 28 movimentos. As execuções 3 e 10 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima na segunda geração.

Tabela 9 – Taxa de elitismo = 20%.

Execução	**Geração**	**Tamanho**
1	2001	20
2	2001	19
3	1	31
4	1	36
5	1	28
6	2	38
7	2001	16
8	1	24
9	17	41
10	2001	39
Média	**4**	**29,2**
Desvio Padrão	**6**	**8,68**

Com a taxa de elitismo de 20% (Tabela 9), o AG obteve a solução ótima em 6 das 10 execuções. A média de gerações necessárias para alcançar a solução foi de 4, com desvio padrão populacional de 8,68, enquanto o tamanho médio dos indivíduos gerados foi de 29,2 movimentos. As execuções 3, 4, 5 e 8 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima logo na primeira geração.

5.2.1 Análise de resultados

A análise dos resultados obtidos evidencia que a taxa de elitismo exerce influência direta no desempenho do algoritmo genético aplicado ao problema do *8-Puzzle*.

Quando o elitismo foi fixado em 1%, observou-se dificuldade em encontrar a solução em diversas execuções, apresentando média elevada de gerações (243) e desvio padrão alto de 404. O aumento para 5% melhorou a eficiência, reduzindo o número médio de gerações para 18 e com menor variação, embora ainda tenham ocorrido execuções sem solução ideal.

Na taxa de 20%, o desempenho foi superior, com média de apenas 4 gerações e resultados mais consistentes, mostrando-se o valor mais eficiente neste experimento. O elitismo de 15% não apresentou resultados muito diferentes da taxa anterior, porém elevou a média para 9 gerações.

Desta forma, conclui-se que, para o conjunto de parâmetros testado, a taxa de elitismo de 20% proporcionou melhor equilíbrio entre exploração e preservação das melhores soluções.

5.3 Experimentos variando a taxa de *crossover*

Nesta etapa, foram realizados experimentos variando a taxa de *crossover* entre quatro estados: 80%, 90%, 95% e 100%, mantendo-se fixos os demais parâmetros previamente definidos (15% de taxa de mutação e 20% de elitismo). O objetivo foi avaliar o impacto dessa variação na qualidade das soluções obtidas, considerando métricas como número de execuções bem-sucedidas, geração média de convergência, tamanho médio dos indivíduos. As tabelas a seguir mostram os resultados obtidos neste experimento.

Tabela 10 – Taxa de *crossover* = 80%.

Execução	**Geração**	**Tamanho**
1	19	32
2	12	33
3	1	26
4	1	26
5	2001	25
6	2001	16
7	1	20
8	1	25
9	2	22
10	15	45
Média	**5**	**23,9**
Desvio Padrão	**6**	**7,78**

Percebe-se na Tabela 10, que entre as 10 execuções totais, o AG conseguiu chegar na solução do *puzzle* em 8 tentativas, e falhou em encontrá-la nas outras duas. Quanto ao número médio de gerações necessárias para alcançar a solução final, este foi de 5 gerações, com média de 24,9 movimentos. Para a taxa de *crossover* de 80%, o desvio populacional foi de 7,78. As execuções 3, 4, 7 e 8 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima logo na primeira geração.

Tabela 11 – Taxa de *crossover* = 90%.

Execução	**Geração**	**Tamanho**
1	1	37
2	1	23
3	3	27
4	2001	34
5	2001	18
6	38	36
7	26	41
8	2001	31
9	1	17
10	2001	15
Média	**11,6**	**27,9**
Desvio Padrão	**15**	**8,8**

Para a taxa de *crossover* de 90%, o AG se mostrou apto a encontrar a solução em 6 execuções com média geracional 11,6 gerações. Para esta configuração, a média de movimentos foi de 27,9 por cromossomo e o desvio padrão populacional de 8,8. As execuções 1, 2 e 9 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima logo na primeira geração.

Tabela 12 – Taxa de *crossover* = 95%.

Execução	**Geração**	**Tamanho**
1	5	29
2	2	21
3	2001	18
4	18	33
5	1	20
6	2	29
7	2001	17
8	2	20
9	42	34
10	4	18
Média	**10**	**23,9**
Desvio Padrão	**13**	**6,27**

Para a taxa de *crossover* igual a 95% (Tabela 12), o AG não se mostrou capaz de encontrar a solução ótima em 2 execuções, conseguindo encontrá-la nas outras 8 execuções. O número médio de gerações em que a solução final foi encontrada com sucesso foi de 10, com desvio padrão de 13. O tamanho médio do cromossomo para as 10 execuções foi de 23,9 movimentos, enquanto o desvio padrão foi de 6,27. A execução 5 obteve o melhor desempenho neste experimento, alcançando a solução ótima na primeira geração.

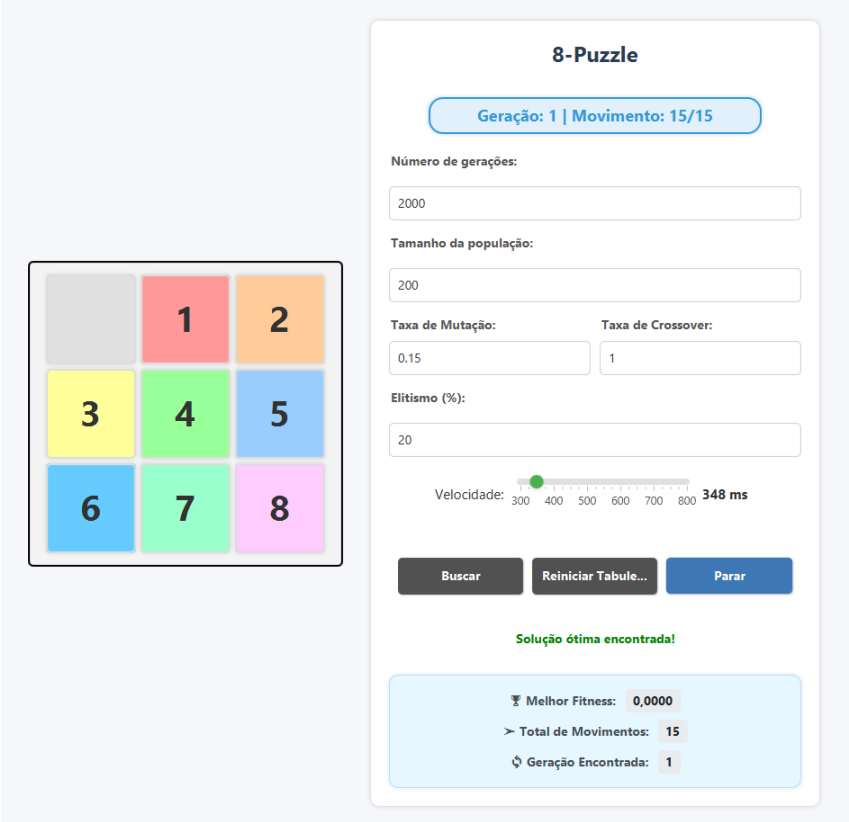
Tabela 13 – Taxa de *crossover* = 100%.

Execução	**Geração**	**Tamanho**
1	6	26
2	2001	33
3	2001	29
4	1	15
5	2001	28
6	1	31
7	3	31
8	1	17
9	34	36
10	4	88
Média	**7**	**26,6**
Desvio Padrão	**11**	**6,31**

Na Tabela 13 observa-se que a taxa de cruzamento de 100% obteve o número médio de 7 gerações para chegar até a solução. Das 10 tentativas de busca, cada uma para um estado inicial aleatório do tabuleiro, o AG encontrou ótimas soluções em 7 e falhou nas outras 3. A média de tamanho dos cromossomos foi de 26,6 e desvio padrão da população de 6,31. As execuções 4, 6 e 8 apresentaram o melhor desempenho neste experimento, alcançando a solução ótima logo na primeira geração.

A Figura 7 mostra na interface o melhor resultado obtido das 10 execuções, explicitando: os valores escolhidos pelo usuário de cada parâmetro desta configuração, a geração em que foi encontrada a melhor solução, o total de movimentos desta solução e o *fitness*, que neste caso foi 0.

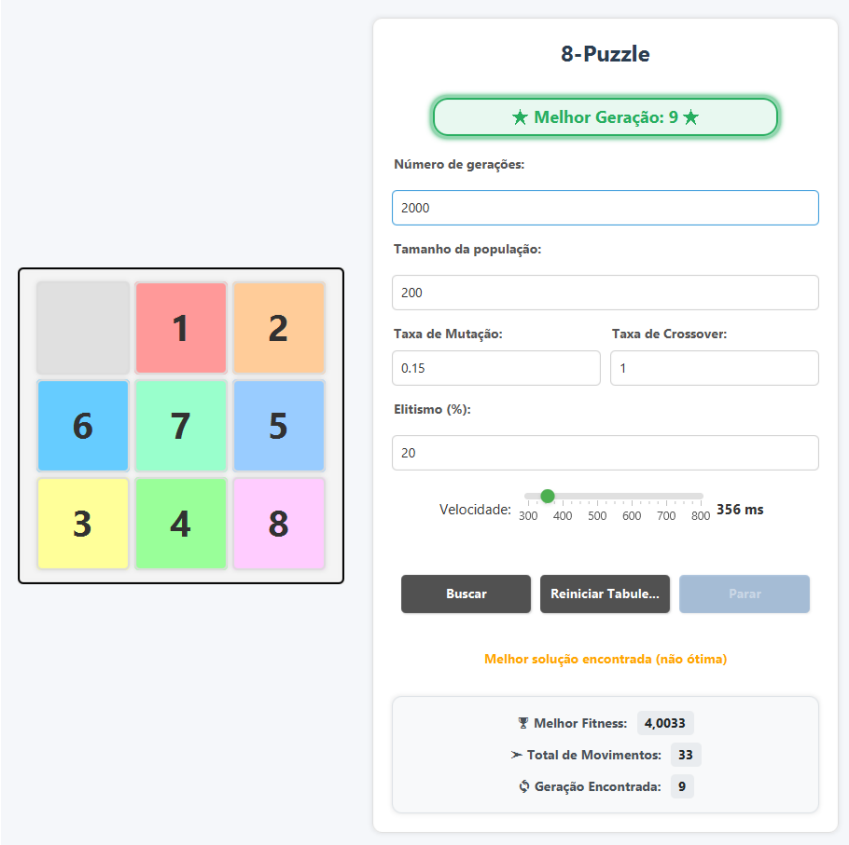
Figura 7 – **Resultado da melhor execução do *8-Puzzle* com taxa de *crossover* 100%.**



Fonte: Elaborado pela autora.

A Figura 8 mostra na interface um caso em que não foi encontrada a solução ótima com esta configuração, expondo: os valores digitados pelo usuário de cada parâmetro, a geração em que foi encontrada a melhor solução nesta execução, o total de movimentos desta solução e o *fitness*, que neste caso foi de 4,0033.

Figura 8 – **Resultado da solução ótima não encontrada de uma das execuções do *8-Puzzle* com taxa de *crossover* 100%.**



Fonte: Elaborado pela autora.

5.3.1 Análise de resultados

Os resultados mostram que, na etapa dos experimentos realizados com valores variados de *crossover*, destacou-se principalmente a taxa de 80%. Esta configuração apresentou o melhor equilíbrio geral entre taxa de sucesso, qualidade das soluções e consistência dos resultados, destacando-se pela menor média de gerações necessárias para alcançar a solução (5), pelo menor número médio de movimentos (23,9, empatado com a taxa de 95%) e pela maior consistência em encontrar a solução ótima logo na primeira geração (quatro execuções).

A taxa de 95% de *crossover* apresentou desempenho próximo, também com 8 sucessos em 10 execuções e média de 23,9 movimentos. No entanto, exigiu em média 10 gerações para convergir até a solução, o que indica maior dificuldade em relação à taxa de 80%.

A taxa de 100% de *crossover* apresentou um desempenho intermediário, obtendo sucesso em 7 execuções e alcançou a solução em média na 7ª geração, com cromossomos de 26,6 movimentos. Apesar de algumas execuções alcançarem a solução ótima já na primeira geração, a taxa de falhas por execução foi mais alta, o que compromete sua eficiência.

Por fim, a taxa de 90% de *crossover* foi a que apresentou os resultados menos consistentes, obtendo sucesso em apenas 6 execuções, com maior número médio de movimentos (27,9) e maior tempo para chegar ao sucesso (11,6 gerações). Embora tenha apresentado bons resultados em alguns casos específicos — como nas execuções que encontraram a solução ótima já na primeira geração —, mostrou-se menos eficiente em termos gerais.

Dessa forma, pode-se concluir que a taxa de 80% de *crossover* foi a mais eficaz, conciliando alta taxa de sucesso, menor número de gerações e soluções curtas, configurando-se como a escolha mais adequada dentre os quatro cenários avaliados.

5.4 Considerações Finais

A partir dos experimentos realizados, observou-se que a variação nos parâmetros do algoritmo genético impacta diretamente o processo de busca por soluções para o *8-Puzzle*. Além disso, constatou-se que a presença de uma interface, ao mediar a interação entre o usuário e o sistema, contribui significativamente para tornar o aprendizado mais acessível e intuitivo.

Entre todas as configurações com taxa de mutação, o valor de 15% demonstrou ser o mais adequado, uma vez que se mostrou capaz de encontrar a solução em um número menor de gerações e também baixa variabilidade, indicando estabilidade ao longo das 10 execuções. Para os experimentos realizados com o elitismo, a taxa de 20% teve o melhor desempenho com média de 4 gerações e aproximadamente 29 movimentos por cromossomo. Por fim, na etapa dos experimentos realizados com valores variados de *crossover*, a configuração que mais se destacou por seu equilíbrio entre taxa de sucesso, qualidade das soluções e consistência dos resultados foi a de 80%.

Com base nos resultados obtidos, evidencia-se que o ajuste adequado dos parâmetros do algoritmo genético é fundamental para assegurar tanto a eficiência quanto a qualidade na busca por soluções do problema. Os experimentos demonstraram que uma taxa de mutação de 15%, combinada com elitismo de 20% e uma taxa de *crossover* de 80%, oferece o melhor equilíbrio entre taxa de sucesso, rapidez em encontrar a solução e simplicidade das soluções. Estes resultados destacam a relevância de um ajuste eficaz dos parâmetros, já que mesmo pequenas alterações podem influenciar consideravelmente o desempenho do AG.

Além dos resultados quantitativos obtidos, a interface desempenhou papel fundamental no processo, uma vez que possibilitou a visualização e compreensão do funcionamento do AG de maneira clara e intuitiva. Ao permitir que o usuário acompanhasse, em tempo real, a evolução das gerações, a convergência da população e a qualidade das soluções, a interface contribuiu não apenas para a usabilidade do sistema, mas também

para a formação de um entendimento mais sólido sobre o impacto dos parâmetros no desempenho do AG.

Portanto, os experimentos não só comprovam a eficácia dos algoritmos genéticos na resolução do problema, mas também a aplicabilidade das interfaces na resolução de problemas de otimização complexos, além de fornecerem base para futuros refinamentos e comparações com outras técnicas de busca.

6 Conclusão

O presente trabalho teve como objetivo desenvolver um sistema para o problema do *8-Puzzle* com uma interface gráfica, integrando-a a um algoritmo genético com o intuito de oferecer ao usuário uma visualização clara e interativa do processo de resolução. A proposta mostrou-se eficaz, atendendo aos objetivos definidos inicialmente, uma vez que foi possível tanto implementar um solucionador baseado em algoritmos genéticos quanto disponibilizar um ambiente de fácil utilização para a experimentação e a análise do AG pelo usuário.

A interface desenvolvida permite ao usuário configurar parâmetros essenciais do Algoritmo Genético (AG), como taxa de mutação, taxa de cruzamento e elitismo, além de acompanhar em tempo real a evolução das soluções geradas. Essa abordagem resultou não apenas em uma ferramenta prática para a resolução do *8-Puzzle*, mas também em um recurso didático que contribui significativamente para a compreensão dos mecanismos de funcionamento dos AGs. Por meio dos experimentos, pode-se comprovar a eficiência do AG em buscar soluções para o problema do *8-Puzzle*, especialmente quando os valores dos parâmetros foram ajustado de modo adequado.

Para trabalhos futuros, destacam-se as seguintes possibilidades: a implementação e comparação com outros algoritmos de busca clássicos; a extensão da interface para *puzzles* mais complexos, como o de 15 peças; a inserção de uma nova funcionalidade que permita o usuário obter os *logs* do console com o intuito de acompanhar a evolução em cada geração; e a aplicação de operadores genéticos mais sofisticados, capazes de explorar melhor o espaço de busca.

Conclui-se que os objetivos propostos foram atingidos, resultando em uma ferramenta interativa que une teoria e prática, contribuindo para o estudo de algoritmos genéticos e para a compreensão do processo de resolução do *8-Puzzle*.

Referências

ANTAS, R. A. M.; SOUTO, G.; VALENTIM, R. Interface adaptável para jogos digitais: jogando com a voz. In: **Anais do Simpósio Brasileiro de Jogos para Computadores e Entretenimento Digital (SBGames)**. [S.l.: s.n.], 2015. Citado na página 17.

BHASIN, H.; SINGLA, N. Genetic based algorithm for n-puzzle problem. **International Journal of Computer Applications**, Citeseer, v. 51, n. 22, 2012. Citado na página 16.

CARMO, N. D. **Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação)**. Dissertação (Mestrado) — Centro Universitário de Caratinga, 2017. Citado 2 vezes nas páginas 10 e 23.

DANTAS, V. F. et al. Combinando desafios e aventura em um jogo para apoiar a aprendizagem de programação em vários níveis cognitivos. In: **Anais do Simpósio Brasileiro de Informática na Educação (SBIE)**. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 10 e 17.

DEITEL, P.; DEITEL, H. **Java: como programar**. 10. ed. [S.l.]: Pearson Education, 2016. Citado 2 vezes nas páginas 22 e 26.

DIGIAMPIETRI, L. A.; KROPIWIEC, D. D. Desenvolvimento de jogos para o aperfeiçoamento na aprendizagem de disciplinas de ciência da computação. In: **VII Simpósio Brasileiro de Jogos para Computadores e Entretenimento Digital (SBGames)**. [S.l.: s.n.], 2008. Citado na página 16.

FARIAS, A. A. d. et al. Desenvolvimento de jogo digital como estratégia de melhoria na cognição e motricidade de idosos. **Brazilian Journal of Development**, v. 8, n. 11, p. 72529–72540, 2022. Citado 2 vezes nas páginas 18 e 20.

GABRIEL, P. H. R.; DELBEM, A. C. B. **Fundamentos de algoritmos evolutivos**. [S.l.], 2008. Citado 2 vezes nas páginas 13 e 15.

GAMMA, E. **Padrões de projetos: soluções reutilizáveis**. [S.l.]: Bookman editora, 2009. Citado na página 24.

GOLDBERG, D. E. Genetic algorithm in search, optimization and machine learning, addison. **W esley Publishing Company, R eading, MA**, v. 1, n. 98, p. 9, 1989. Citado na página 15.

JONG, K. A. D. **Evolutionary Computation: A Unified Approach**. Cambridge, MA: MIT Press, 2006. Citado na página 13.

JUNIOR, N. F.; GUIMARAES, F. G. Problema 8-Puzzle: Análise da solução usando Backtracking e Algoritmos Genéticos. **PPGCC - Programa de Pós-Graduação em Ciência da Computação, UFOP - Universidade Federal de Ouro Preto**, Ouro Preto, Minas Gerais, Brasil, 2018. Citado 2 vezes nas páginas 10 e 15.

MEDINA, J.; MÜLLER, R. M. A utilização de algoritmos genéticos no desenvolvimento de jogos. In: **Anais do Encontro Nacional de Iniciação Científica (ENIC)**. [S.l.: s.n.], 2009. Citado na página 17.

ORACLE. **The Java Language Environment – Design Goals**. 1996. Acesso em: 17 set. 2025. Disponível em: <https://www.oracle.com/java/technologies/simple-familiar.html>. Citado na página 20.

RIBEIRO, I. F. **Desenvolvimento de software educativo para o processamento auditivo: uma ferramenta para a alfabetização**. Dissertação (Mestrado) — Universidade Estadual do Sudoeste da Bahia, 2018. Citado 4 vezes nas páginas 17, 18, 20 e 23.

RISSETTI, G.; MACHADO, F.; MIRANDA, P. Fx canvas2d: uma api de jogos bidimensionais para auxiliar na aprendizagem de programação. In: **Anais do Congresso Brasileiro de Informática na Educação (CBIE)**. [S.l.: s.n.], 2017. Citado 3 vezes nas páginas 17, 18 e 20.

SANTANA, K. et al. Blinds, basic education: jogo digital inclusivo para auxiliar o processo de ensino-aprendizagem das pessoas com deficiência visual. In: **Anais do Simpósio Brasileiro de Informática na Educação (SBIE)**. [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 10 e 17.

SILVA, A. B. **Solução mobile e uso de tecnologia API Rest para problema de roteirização**. Tese (Doutorado) — Programa de Pós-Graduação em Pesquisa Operacional e Inteligência Computacional, Universidade Candido Mendes, 2020. Citado na página 18.

YANG, C. M.; PEK, V.; SALAM, Z. A. A.; LING, S. H. Solver of 8-puzzle with genetic algorithm. **Journal of Applied Technology and Innovation (e-ISSN: 2600-7304)**, v. 7, n. 1, p. 28, 2023. Citado na página 10.