INSTITUTO MAUÁ DE TECNOLOGIA

ENGENHARIA DA COMPUTAÇÃO

WORDGRAPHLE

Um Jogo de Palavras Baseado em Grafos com Suporte à Língua Portuguesa

Trabalho apresentado como requisito parcial para aprovação na disciplina de Tópicos Avançados em Estruturas de Dados

Enzo Oliveira D'Onofrio - RA: 23.01561-6
Leonardo Souza Olivieri - RA: 23.01512-8
Arthur Gama Ruiz - RA: 23.01445-8
João Vitor Morimoto Sesma - RA: 23.01516-0
Pedro Wilian Palumbo Bevilacqua - RA: 23.01307-9
Felipe Fazio da Costa - RA: 23.00055-4

Orientador: Robson Calvetti

São Caetano do Sul
Novembro de 2025

Resumo

Este trabalho apresenta o desenvolvimento do *WordGraphle*, um jogo de adivinhação de palavras inspirado no popular jogo *Wordle*, com extensões significativas que incluem suporte completo à língua portuguesa (incluindo acentuação), um sistema de sugestões inteligentes baseado em análise de frequência e uma visualização gráfica das relações entre letras através de um modelo de grafo em camadas. O sistema foi implementado em Java, utilizando a biblioteca Swing para a interface gráfica, seguindo os princípios de orientação a objetos e uma arquitetura em camadas. A solução desenvolvida incorpora estruturas de dados avançadas, incluindo grafos direcionados acíclicos (DAGs), algoritmos de normalização de texto e sistemas de restrições para a filtragem eficiente de candidatos. Os resultados demonstram uma aplicação funcional e educativa que combina teoria de grafos, estruturas de dados e engenharia de software.

Palavras-chave: Grafos. Algoritmos. Java. Interface Gráfica. Processamento de Linguagem Natural. Jogos Educativos.

Abstract

This work presents the development of *WordGraphle*, a word-guessing game inspired by the popular *Wordle* game, with significant extensions including full support for the Portuguese language (including accents), an intelligent suggestion system based on frequency analysis, and a graphical visualization of letter relationships through a layered graph model. The system was implemented in Java using the Swing library for the graphical interface, following object-oriented principles and a layered architecture. The developed solution incorporates advanced data structures, including directed acyclic graphs (DAGs), text normalization algorithms, and constraint systems for efficient candidate filtering. The results demonstrate a functional and educational application that combines graph theory, data structures, and software engineering.

Keywords: Graphs. Algorithms. Java. Graphical Interface. Natural Language Processing. Educational Games.

Sumário

1 Introdução

1.1 Contextualização

Com a popularização de jogos de palavras digitais, especialmente após o sucesso global do *Wordle* em 2021 [1], surgiu a oportunidade de desenvolver versões localizadas e aprimoradas desses jogos. O *WordGraphle* representa uma evolução desse conceito, incorporando técnicas avançadas de ciência da computação para criar uma experiência de jogo mais rica e educativa.

Jogos de adivinhação de palavras não são apenas formas de entretenimento, mas também ferramentas valiosas para o desenvolvimento cognitivo, a expansão do vocabulário e a compreensão de padrões linguísticos [2]. A implementação computacional desses jogos oferece oportunidades únicas para aplicar conceitos de estruturas de dados, algoritmos e teoria de grafos de forma prática e tangível.

1.2 Objetivos

O objetivo geral deste projeto é desenvolver um jogo de adivinhação de palavras completo e funcional que demonstre a aplicação prática de conceitos avançados de ciência da computação, incluindo:

- Implementação de estruturas de dados baseadas em grafos para análise linguística.
- Desenvolvimento de algoritmos de sugestão inteligente baseados em frequências.
- Criação de um sistema de restrições para filtragem eficiente de candidatos.
- Construção de uma interface gráfica moderna e responsiva.
- Suporte completo à língua portuguesa, incluindo a normalização de acentos.

Os objetivos específicos incluem:

1. Projetar e implementar uma arquitetura modular em camadas.
2. Criar um modelo de grafo direcionado acíclico (DAG) para representar transições entre letras.
3. Desenvolver um algoritmo de avaliação de palpites com feedback colorido.
4. Implementar um sistema de restrições acumulativas para o tratamento de letras repetidas.
5. Visualizar graficamente as relações entre letras em tempo real.
6. Gerar sugestões inteligentes baseadas em análise estatística.

1.3 Justificativa

Este projeto se justifica tanto do ponto de vista educacional quanto técnico. Do ponto de vista educacional, oferece uma aplicação prática e interativa de conceitos fundamentais da ciência da computação. Do ponto de vista técnico, demonstra como técnicas avançadas de estruturas de dados e algoritmos podem ser aplicadas para resolver problemas reais de forma elegante e eficiente.

Além disso, o suporte à língua portuguesa, incluindo o tratamento adequado de acentuação, preenche uma lacuna importante, uma vez que muitas implementações similares focam exclusivamente no inglês.

2 Fundamentação Teórica

2.1 Teoria de Grafos

Um grafo $G = (V, E)$ é uma estrutura matemática composta por um conjunto de vértices (ou nós) V e um conjunto de arestas E que conectam pares de vértices [3]. No contexto do *WordGraphle*, utilizamos um tipo especial de grafo chamado Grafo Direcionado Acíclico (DAG - *Directed Acyclic Graph*).

2.1.1 Grafos Direcionados Acíclicos (DAGs)

Um DAG é um grafo direcionado que não contém ciclos, ou seja, não é possível começar em um vértice e retornar a ele seguindo as arestas direcionadas [4]. No *WordGraphle*, o grafo é organizado em camadas, onde:

- Cada camada representa uma posição na palavra (0 a $L-1$, onde L é o comprimento da palavra).
- Cada nó em uma camada representa uma letra do alfabeto (A-Z).
- Arestas direcionadas conectam nós da camada i para nós da camada $i+1$.
- O peso de cada aresta representa a frequência dessa transição no dicionário.

Formalmente, o grafo pode ser descrito como:

$$G = (V, E, w) \tag{1}$$

onde:

$$V = \{(pos, letra) \mid 0 \leq pos < L,\ A \leq letra \leq Z\} \tag{2}$$

$$E = \{((pos, a), (pos + 1, b)) \mid 0 \leq pos < L - 1\} \tag{3}$$

$$w : E \to \mathbb{N} \tag{4}$$

A função peso $w(e)$ para uma aresta $e = ((pos, a), (pos + 1, b))$ é definida como:

$$w(e) = |\{w \in D \mid w[pos] = a \wedge w[pos + 1] = b\}| \tag{5}$$

onde D é o conjunto de palavras no dicionário.

2.2 Sistemas de Restrições

Um sistema de restrições é um conjunto de condições que devem ser satisfeitas simultaneamente [5]. No *WordGraphle*, utilizamos três tipos de restrições:

2.2.1 Restrições Posicionais

Para cada posição pos ($0 \leq pos < L$), podemos ter:

- Uma letra fixa $fixed[pos] \in \{-1, 0..25\}$ (onde -1 indica a ausência de restrição).
- Um conjunto de letras banidas $bannedPos[pos] \subseteq \{0..25\}$.

2.2.2 Restrições Globais de Contagem

Para cada letra l ($0 \leq l < 26$):

- Contagem mínima: $minCount[l] \in \mathbb{N}$
- Contagem máxima: $maxCount[l] \in \mathbb{N} \cup \{-1\}$ (onde -1 indica a ausência de um limite superior).

2.2.3 Validação de Candidatos

Uma palavra w satisfaz as restrições C se e somente se:

$$\begin{aligned} &\forall pos : (fixed[pos] \neq -1) \Rightarrow (w[pos] = fixed[pos]) \\ &\wedge \forall pos : w[pos] \notin bannedPos[pos] \\ &\wedge \forall l : count(l, w) \geq minCount[l] \\ &\wedge \forall l : (maxCount[l] \neq -1) \Rightarrow (count(l, w) \leq maxCount[l]) \end{aligned} \tag{6}$$

onde $count(l, w)$ é o número de ocorrências da letra l na palavra w.

2.3 Normalização de Texto

A normalização de texto é o processo de transformar texto em uma forma canônica padronizada [6]. No *WordGraphle*, utilizamos as seguintes etapas:

1. Decomposição Unicode NFD (*Normalization Form Canonical Decomposition*).
2. Remoção de marcas diacríticas combinantes.
3. Conversão de ç/Ç para c/C.
4. Remoção de caracteres não alfabéticos.
5. Conversão para maiúsculas.

Este processo garante que palavras como "PÁTIO"e "PATIO"sejam tratadas como equivalentes na lógica do jogo, preservando a forma original apenas para exibição.

2.4 Análise de Frequências

A análise de frequências é fundamental para o sistema de sugestões. Utilizamos duas abordagens:

2.4.1 Frequências Posicionais Simples

Dado um conjunto de candidatos C, a frequência da letra l na posição pos é:

$$freq[pos][l] = \frac{|\{w \in C \mid w[pos] = l\}|}{|C|} \tag{7}$$

2.4.2 Frequências Baseadas no Grafo

O grafo fornece uma estimativa de frequência mais sofisticada que considera a estrutura global do dicionário:

$$score[pos][l] = \begin{cases} \sum_b w((pos, l), (pos+1, b)) & \text{se } pos = 0 \\ \frac{\sum_a w((pos-1,a),(pos,l)) + \sum_b w((pos,l),(pos+1,b))}{2} & \text{se } 0 < pos < L-1 \\ \sum_a w((pos-1, a), (pos, l)) & \text{se } pos = L-1 \end{cases} \tag{8}$$

normalizado por posição para $\sum_l score[pos][l] = 1$.

3 Arquitetura do Sistema

3.1 Visão Geral

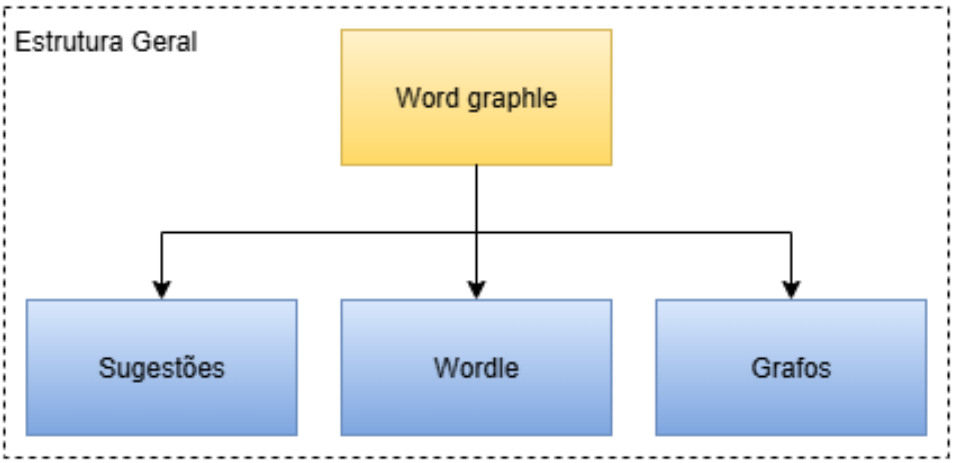

Figura 1: Estrutura conceitual de alto nível do aplicativo.

O *WordGraphle* foi desenvolvido seguindo uma arquitetura em camadas, separando responsabilidades e facilitando a manutenção e a extensão. A arquitetura segue o padrão MVC (*Model-View-Controller*) adaptado, com as seguintes camadas principais:

1. **Camada de Aplicação**: Ponto de entrada do sistema.
2. **Camada de Interface (UI)**: Componentes visuais e interação com o usuário.
3. **Camada de Lógica (Engine)**: Motor do jogo e regras de negócio.
4. **Camada de Modelo**: Estruturas de dados e modelos de domínio.
5. **Camada de Resolução (Solver)**: Algoritmos de sugestão e filtragem.
6. **Camada de Grafo**: Estrutura de grafo e análises.

3.2 Organização de Pacotes

O projeto está organizado nos seguintes pacotes Java:

```
wordgraphle/
wordgraphle/
|-- WordGraphleApp.java        (main)
|-- engine/
|   `-- GameEngine.java        (game engine)
|-- model/
|   |-- Dictionary.java        (dictionary)
|   |-- Constraints.java       (constraints)
```

```
|   |-- Feedback.java          (feedback)
|   `-- FeedbackColor.java     (color enum)
|-- graph/
|   `-- GraphModel.java        (graph model)
|-- solver/
|   `-- Solver.java            (suggestions)
`-- ui/
    |-- WordGraphleFrame.java  (main window)
    `-- GraphPanel.java        (graph view)
```

Listing 1: Estrutura de Pacotes

3.3 Diagrama de Classes Simplificado

As principais classes e suas relações são:

- `WordGraphleApp`: Classe principal com método `main()`.
- `GameEngine`: Gerencia o estado do jogo, a palavra secreta e a avaliação de palpites.
- `Dictionary`: Carrega e gerencia as palavras do dicionário.
- `Constraints`: Armazena e gerencia as restrições acumuladas.
- `GraphModel`: Implementa o grafo em camadas.
- `Solver`: Fornece filtragem de candidatos e sugestões.
- `WordGraphleFrame`: Interface gráfica principal.
- `GraphPanel`: Visualização gráfica do grafo.

4 Implementação

4.1 Camada de Modelo

4.1.1 Dictionary (Dicionário)

A classe `Dictionary` é responsável por carregar e gerenciar o vocabulário do jogo. Suas principais características incluem:

```
public final class Dictionary {
    private final int L;                        // Tamanho das palavras
    private final List<String> words;           // Palavras canônicas (A-Z)
    private final LinkedHashMap<String,String> display; // Mapeamento para exibição

    public static String normalize(String w) {
        // Normalização NFD + remoção de diacríticos + uppercase
    }

    public static Dictionary loadFromFile(File f, int L) throws IOException {
        // Carrega de arquivo UTF-8
    }
}
```

Listing 2: Estrutura da Classe Dictionary

A normalização utiliza a API de Unicode do Java:

```
public static String normalize(String w) {
    if (w == null) return "";
    String s = Normalizer.normalize(w, Normalizer.Form.NFD)
            .replaceAll("\\p{InCombiningDiacriticalMarks}+", "")
            .replace('ç','c').replace('Ç','C')
            .replaceAll("[^A-Za-z]", "")
            .toUpperCase(Locale.ROOT);
    return s;
}
```

Listing 3: Algoritmo de Normalização

Este método permite que palavras como "ÁGUA", "NÚMERO"e "AÇÃO"sejam internamente tratadas como "AGUA", "NUMERO"e "ACAO", preservando as formas originais apenas para exibição.

4.1.2 Constraints (Restrições)

A classe `Constraints` encapsula todas as restrições acumuladas durante o jogo:

```java
public class Constraints {
    private final int L;
    public final int[] fixed;          // -1 ou letra fixa (0..25) por
        posição
    public final boolean[][] bannedPos; // [L][26] letras banidas por
        posição
    public final int[] minCount;       // [26] mínimo de ocorrências
       por letra
    public final int[] maxCount;       // [26] máximo (-1 = ilimitado)

    public static int idx(char c) {
        char u = Character.toUpperCase(c);
        if (u < 'A' || u > 'Z') return -1;
        return u - 'A';
    }
}
```

Listing 4: Estrutura da Classe Constraints

A função `idx()` converte caracteres para índices numéricos, facilitando o acesso aos arrays.

4.1.3 Feedback e FeedbackColor

Estas classes encapsulam o resultado da avaliação de um palpite:

```java
public enum FeedbackColor {
    GREEN, YELLOW, GRAY;

    public static java.awt.Color toAwt(FeedbackColor c) {
        return switch (c) {
            case GREEN  -> new java.awt.Color(83, 141, 78);
            case YELLOW -> new java.awt.Color(181, 159, 59);
            case GRAY   -> new java.awt.Color(58, 58, 60);
        };
    }
}

public class Feedback {
    public final String guess;
    public final FeedbackColor[] colors;
}
```

Listing 5: Classes de Feedback

4.2 Camada de Lógica (Engine)

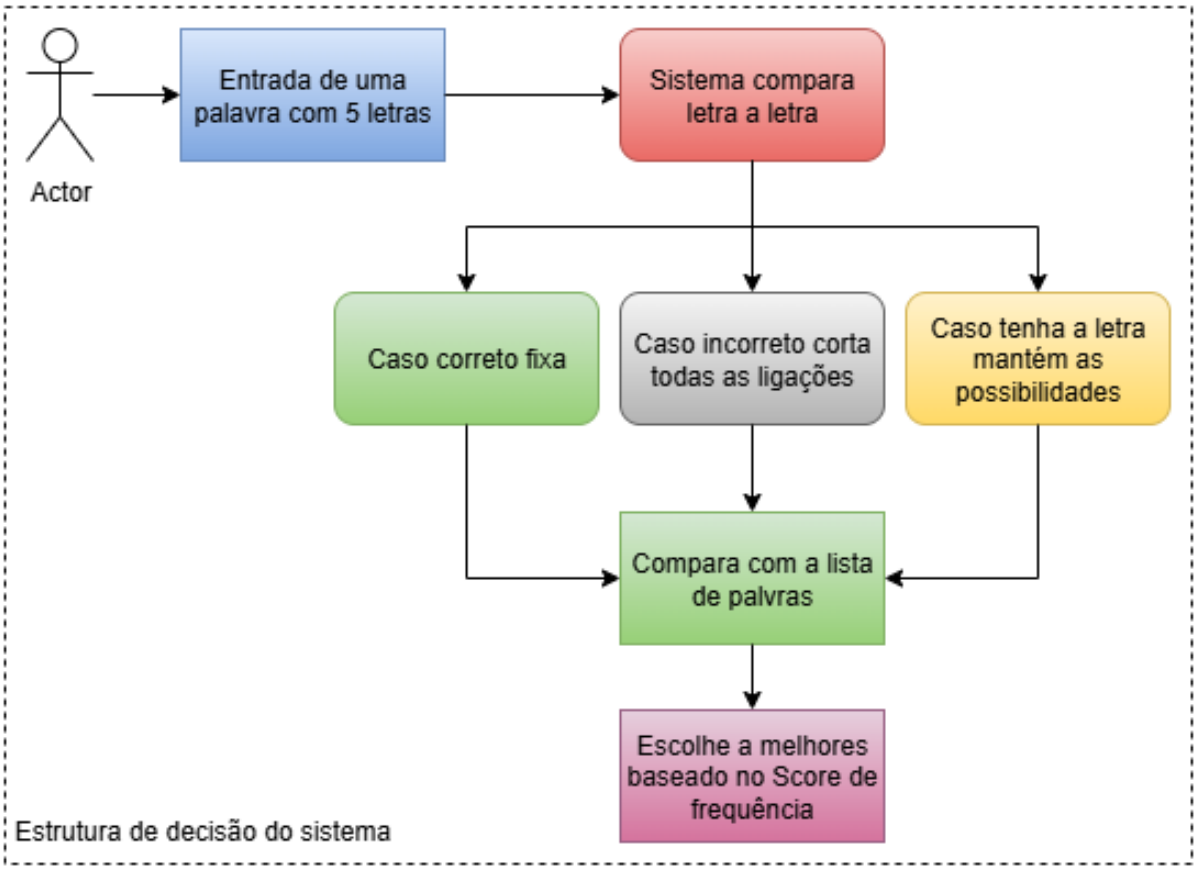

Figura 2: Fluxograma da estrutura de decisão do sistema, desde a entrada do usuário até a geração de sugestões.

4.2.1 GameEngine

O `GameEngine` é o coração do jogo, gerenciando o estado e implementando as regras:

```java
public class GameEngine {
    private final Dictionary dict;
    private String secret;            // Palavra secreta canônica
    private String secretDisplay;     // Palavra secreta com acentos
    private final Constraints constraints;

    public Feedback evaluate(String rawGuess) {
        // 1. Normaliza o palpite
        String guess = Dictionary.normalize(rawGuess);

        // 2. Conta letras na palavra secreta
        int[] remaining = new int[26];
        for (char c : secret.toCharArray()) {
            remaining[Constraints.idx(c)]++;
        }

```

```
17          // 3. Primeiro passe: marca GREEN (posição correta)
18          // 4. Segundo passe: marca YELLOW ou GRAY
19          // 5. Atualiza restrições
20
21          return new Feedback(guess, colors);
22      }
23  }
```

Listing 6: Estrutura Básica do GameEngine

O algoritmo de avaliação funciona em dois passes para tratar corretamente letras repetidas, garantindo que uma letra não seja marcada como amarela se já foi consumida por uma correspondência verde.

4.2.2 Atualização de Restrições

A lógica de atualização de restrições é crucial e complexa:

```
1  private void applyFeedbackToConstraints(String guess, FeedbackColor[]
       colors) {
2      // Restrições posicionais
3      for (int pos = 0; pos < L; pos++) {
4          int id = Constraints.idx(guess.charAt(pos));
5          if (colors[pos] == FeedbackColor.GREEN) {
6              constraints.fixed[pos] = id;
7          } else {
8              constraints.bannedPos[pos][id] = true;
9          }
10     }
11
12     // Restrições de contagem global
13     int[] greenYellow = new int[26];
14     int[] total = new int[26];
15
16     for (int pos = 0; pos < L; pos++) {
17         int id = Constraints.idx(guess.charAt(pos));
18         total[id]++;
19         if (colors[pos] != FeedbackColor.GRAY) {
20             greenYellow[id]++;
21         }
22     }
23
24     for (int id = 0; id < 26; id++) {
25         // Atualiza mínimo
26         constraints.minCount[id] = Math.max(
27             constraints.minCount[id], greenYellow[id]);
28
29         // Atualiza máximo se houver GRAY
```

```
30          if (total[id] > greenYellow[id]) {
31              int ceiling = greenYellow[id];
32              if (constraints.maxCount[id] == -1) {
33                  constraints.maxCount[id] = ceiling;
34              } else {
35                  constraints.maxCount[id] = Math.min(
36                      constraints.maxCount[id], ceiling);
37              }
38          }
39      }
40  }
```

Listing 7: Atualização de Restrições (Simplificado)

Este algoritmo garante que letras repetidas sejam tratadas corretamente. (Ver exemplos nas seções 6.2.1 e B.2).

4.3 Camada de Grafo

4.3.1 GraphModel

O `GraphModel` implementa o grafo em camadas usando uma matriz tridimensional para armazenar pesos:

```
1  public class GraphModel {
2      private final int L;
3      private final int[][][] wBase;     // [L-1][26][26] pesos base
4      private final boolean[][] active;   // [L][26] máscara de nós
           ativos
5
6      public static GraphModel fromDictionary(Dictionary dict) {
7          GraphModel g = new GraphModel(dict.L());
8          for (String w : dict.words()) {
9              for (int pos = 0; pos < L - 1; pos++) {
10                 int a = w.charAt(pos) - 'A';
11                 int b = w.charAt(pos + 1) - 'A';
12                 g.wBase[pos][a][b]++;
13             }
14         }
15         return g;
16     }
17
18     public void applyConstraints(Constraints c) {
19         // Atualiza máscara de nós ativos baseado nas restrições
20     }
21 }
```

Listing 8: Estrutura do GraphModel

A construção do grafo tem complexidade $O(N \cdot L)$, onde N é o número de palavras no dicionário e L é o comprimento das palavras. O espaço ocupado é $O(L \cdot 26^2)$, sendo independente do tamanho do dicionário.

4.3.2 Cálculo de Scores Posicionais

O método `positionLetterScores()` calcula frequências aproximadas baseadas no grafo:

```
public double[][] positionLetterScores() {
    double[][] score = new double[L][26];

    // Posição 0: usa saídas
    for (int a = 0; a < 26; a++) {
        if (!active[0][a]) continue;
        int sum = 0;
        for (int b = 0; b < 26; b++)
            sum += edgeWeight(0, a, b);
        score[0][a] = sum;
    }

    // Posições intermediárias: média de entradas e saídas
    for (int pos = 1; pos < L - 1; pos++) {
        for (int x = 0; x < 26; x++) {
            if (!active[pos][x]) continue;
            int in = 0, out = 0;
            for (int a = 0; a < 26; a++)
                in += edgeWeight(pos - 1, a, x);
            for (int b = 0; b < 26; b++)
                out += edgeWeight(pos, x, b);
            score[pos][x] = (in + out) / 2.0;
        }
    }

    // Posição L-1: usa entradas
    // ... (código similar)

    // Normalização
    for (int pos = 0; pos < L; pos++) {
        double tot = 0;
        for (int l = 0; l < 26; l++)
            tot += score[pos][l];
        if (tot > 0)
            for (int l = 0; l < 26; l++)
                score[pos][l] /= tot;
    }

```

```
39      return score;
40  }
```

Listing 9: Cálculo de Scores do Grafo

4.4 Camada de Resolução (Solver)

4.4.1 Filtragem de Candidatos

O `Solver` fornece métodos para filtrar palavras candidatas com base nas restrições:

```
1  public static List<String> filterCandidates(
2          List<String> dict, Constraints c) {
3      return dict.stream()
4                  .filter(w -> isValid(w, c))
5                  .collect(Collectors.toList());
6  }
7
8  private static boolean isValid(String w, Constraints c) {
9      int L = c.L();
10     int[] cnt = new int[26];
11
12     // Verifica restrições posicionais
13     for (int i = 0; i < L; i++) {
14         int id = Constraints.idx(w.charAt(i));
15         if (c.fixed[i] != -1 && c.fixed[i] != id)
16             return false;
17         if (c.bannedPos[i][id])
18             return false;
19         cnt[id]++;
20     }
21
22     // Verifica restrições de contagem
23     for (int id = 0; id < 26; id++) {
24         if (cnt[id] < c.minCount[id])
25             return false;
26         if (c.maxCount[id] >= 0 && cnt[id] > c.maxCount[id])
27             return false;
28     }
29
30     return true;
31 }
```

Listing 10: Filtragem de Candidatos

A complexidade da filtragem é $O(N \cdot L)$ para processar todos os candidatos.

4.4.2 Sistema de Sugestões

O sistema de sugestões calcula um score para cada palavra candidata:

```
public static double scoreWord(String w, double[][] freq) {
    double s = 0.0;
    boolean[] seen = new boolean[26];

    for (int i = 0; i < w.length(); i++) {
        int id = Constraints.idx(w.charAt(i));
        s += freq[i][id];  // Score de frequência

        // Bônus por diversidade (letras únicas)
        if (!seen[id]) {
            s += 0.02; // Bônus pequeno por letra nova
            seen[id] = true;
        }
    }

    return s;
}

public static List<String> suggestTop(
        List<String> candidates, int L, int k) {
    double[][] freq = positionFrequencies(candidates, L);
    return candidates.stream()
            .sorted((a,b) -> Double.compare(
                scoreWord(b, freq), scoreWord(a, freq)))
            .limit(k)
            .collect(Collectors.toList());
}
```

Listing 11: Sistema de Scoring

O algoritmo favorece palavras que:

- Contêm letras com alta frequência nas posições corretas.
- Têm maior diversidade de letras (menos repetições).

4.5 Camada de Interface (UI)

4.5.1 WordGraphleFrame

A interface gráfica é construída usando Swing com design moderno e minimalista:

```
public class WordGraphleFrame extends JFrame {
    private GameEngine engine;
    private GraphModel graph;
```

```
    private JLabel[][] grid;       // Grade de letras
    private JTextField input;      // Campo de entrada
    private DefaultListModel<String> suggestionsModel;

    // Cores do tema escuro
    private static final Color BG_MAIN = new Color(18, 18, 19);
    private static final Color BG_CELL_EMPTY = new Color(18, 18, 19);
    private static final Color BORDER_CELL_EMPTY = new Color(58, 58,
        60);
}
```

Listing 12: Estrutura da Interface Principal

A interface é organizada em:

- **Cabeçalho**: Título e botão "Novo Jogo".
- **Grade central**: Matriz de células para exibir tentativas.
- **Campo de entrada**: Para digitar palpites.
- **Painel de sugestões**: Lista de palavras sugeridas.
- **Visualização do grafo**: Representação gráfica das transições.

4.5.2 GraphPanel

O `GraphPanel` renderiza o grafo visualmente usando Java 2D:

```
protected void paintComponent(Graphics g0) {
    Graphics2D g = (Graphics2D) g0;
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

    // Seleciona letras mais relevantes por posição
    List<int[]> lettersPerPos = selectTopLetters();

    // Calcula posições X,Y dos nós
    Map<Node, Point> positions = calculateNodePositions();

    // Desenha arestas com espessura proporcional ao peso
    for (Edge e : edges) {
        float thickness = calculateThickness(e.weight);
        g.setStroke(new BasicStroke(thickness));
        g.drawLine(e.from, e.to);
    }

    // Desenha nós
    for (Node n : nodes) {
```

```
21          g.fillOval(n.x, n.y, NODE_RADIUS, NODE_RADIUS);
22          g.drawString(n.label, n.x, n.y);
23      }
24  }
```

Listing 13: Renderização do Grafo (Simplificado)

A visualização usa:

- **Antialiasing** para suavização de bordas.
- **Espessura variável** nas arestas, proporcional ao peso.
- **Transparência** para evitar sobrecarga visual.
- **Limitação** do número de nós exibidos por posição (máximo 12).

5 Análise de Complexidade

5.1 Complexidade Temporal

5.1.1 Construção do Grafo

A construção do grafo a partir do dicionário tem complexidade:

$$T_{construct} = O(N \cdot L) \tag{9}$$

onde N é o número de palavras e L é o comprimento das palavras. Para cada palavra, percorremos $L - 1$ transições.

5.1.2 Avaliação de Palpite

A avaliação de um palpite envolve:

- Normalização: $O(L)$
- Contagem de letras: $O(L)$
- Primeiro passe (GREEN): $O(L)$
- Segundo passe (YELLOW/GRAY): $O(L)$
- Atualização de restrições: $O(L)$

Complexidade total: $T_{evaluate} = O(L)$

5.1.3 Filtragem de Candidatos

Para filtrar candidatos:

$$T_{filter} = O(N \cdot L) \tag{10}$$

No pior caso, verificamos todas as N palavras, cada uma com comprimento L.

5.1.4 Geração de Sugestões

O processo de sugestão envolve:

- Cálculo de frequências: $O(N \cdot L)$
- Scoring de candidatos: $O(N \cdot L)$
- Ordenação: $O(N \log N)$

Complexidade total: $T_{suggest} = O(N \cdot L + N \log N)$

5.1.5 Cálculo de Scores do Grafo

O cálculo de scores posicionais do grafo:

$$T_{graph\_scores} = O(L \cdot 26^2) = O(L) \tag{11}$$

já que o alfabeto é fixo (26 letras).

5.2 Complexidade Espacial

5.2.1 Dicionário

O espaço para armazenar o dicionário:

$$S_{dict} = O(N \cdot L) \tag{12}$$

5.2.2 Grafo

O grafo ocupa:

$$S_{graph} = O(L \cdot 26^2) = O(L) \tag{13}$$

Note que o espaço é independente do tamanho do dicionário.

5.2.3 Restrições

As estruturas de restrições ocupam:

$$S_{constraints} = O(L \cdot 26 + 26) = O(L) \tag{14}$$

5.3 Otimizações Implementadas

Diversas otimizações foram aplicadas:

1. **Uso de arrays primitivos**: Evita o *overhead* de objetos.
2. **LinkedHashMap**: Preserva a ordem de inserção (para o dicionário de exibição) sem *overhead* de performance significativo.
3. **Streams com operações lazy**: Evita a criação de coleções intermediárias desnecessárias.
4. **Normalização única**: Cada palavra é normalizada apenas uma vez, no carregamento.

5. **Limitação de nós no grafo visual**: Exibe apenas os nós mais relevantes para manter a performance de renderização.

6 Resultados e Discussão

6.1 Funcionalidades Implementadas

O sistema final implementa com sucesso todas as funcionalidades planejadas:

1. **Mecânica completa do jogo**: Sistema de palpites com feedback colorido funcionando corretamente.
2. **Suporte ao português**: Normalização adequada de acentos e cedilha.
3. **Sistema de restrições**: Tratamento correto de letras repetidas.
4. **Sugestões inteligentes**: Algoritmo de *scoring* baseado em frequências.
5. **Visualização de grafo**: Representação gráfica das transições entre letras.
6. **Interface moderna**: Design inspirado no Wordle com tema escuro.

6.2 Tratamento de Casos Especiais

6.2.1 Letras Repetidas (Exemplo Corrigido)

O sistema trata corretamente palavras com letras repetidas, incluindo a definição de contagens mínimas e máximas. Exemplo:

- Palavra secreta: BANCO
- Palpite: CANOA
- Resultado: C A N O A
- Análise:
  - C (0): Amarelo (existe, mas na posição errada).
  - A (1): Verde (posição correta).
  - N (2): Verde (posição correta).
  - O (3): Amarelo (existe, mas na posição errada).
  - A (4): Cinza (o palpite 'CANOA' tem dois 'A's, mas a secreta 'BANCO' só tem um, que já foi marcado como Verde).
- Restrições estabelecidas: $minCount[A] = 1$, $maxCount[A] = 1$.

6.2.2 Acentuação

Palavras acentuadas são tratadas transparentemente:

- Entrada do usuário: "ÁGUA"
- Forma canônica interna: "AGUA"
- Exibição preservada: "ÁGUA"

6.3 Performance

Testes com um dicionário de aproximadamente 5.000 palavras de 5 letras demonstraram:

- **Tempo de inicialização**: $< 200ms$ (carregamento do dicionário e construção do grafo).
- **Tempo de avaliação**: $< 5ms$ por palpite.
- **Tempo de filtragem**: $< 50ms$ com milhares de candidatos.
- **Tempo de renderização**: $< 16ms$ (60 FPS) para a visualização do grafo.

6.4 Limitações e Trabalhos Futuros

Algumas limitações foram identificadas:

6.4.1 Limitações Atuais

1. **Alfabeto fixo**: O sistema suporta apenas o alfabeto A-Z (26 letras).
2. **Comprimento fixo**: Requer recompilação para mudar o tamanho das palavras (variável ‘L‘).
3. **Dicionário estático**: Não permite a adição dinâmica de palavras.
4. **Visualização do grafo**: A visualização pode ficar confusa com muitas arestas ativas.

6.4.2 Melhorias Propostas

1. **Modo multijogador**: Implementação de um modo de competição entre jogadores.
2. **Estatísticas**: Rastreamento de histórico de vitórias e tentativas.
3. **Dificuldade variável**: Permitir ao usuário escolher diferentes tamanhos de palavra.

4. **Temas personalizados**: Permitir customização de cores (tema claro/escuro).
5. **Análise de estratégia**: Avaliar a eficácia de diferentes palavras iniciais.
6. **Modo de treinamento**: Exibir o número de candidatos restantes em tempo real.

6.5 Aspectos Educacionais

O projeto demonstrou ser uma excelente ferramenta educacional, ilustrando:

- **Estruturas de dados**: Arrays, listas, mapas, grafos.
- **Algoritmos**: Ordenação, filtragem, busca.
- **Teoria de grafos**: DAGs, pesos em arestas, análise de caminhos.
- **Engenharia de software**: Arquitetura em camadas, separação de responsabilidades.
- **Interface gráfica**: Swing, Java 2D, manipulação de eventos (*event handling*).
- **Processamento de texto**: Normalização Unicode, análise de frequências.

7 Conclusão

Este trabalho apresentou o desenvolvimento completo do *WordGraphle*, um jogo de adivinhação de palavras que incorpora técnicas avançadas de ciência da computação. O sistema implementado demonstra com sucesso a aplicação prática de conceitos teóricos, incluindo teoria de grafos, sistemas de restrições, análise de frequências e engenharia de software.

A arquitetura modular em camadas facilitou o desenvolvimento e permite extensões futuras. O uso de grafos direcionados acíclicos para modelar transições entre letras mostrou-se uma abordagem elegante e eficiente para a análise linguística. O sistema de restrições implementado garante a corretude mesmo em casos complexos, como o de letras repetidas.

O suporte completo à língua portuguesa, incluindo o tratamento adequado de acentuação, diferencia este projeto de implementações similares e demonstra atenção aos detalhes de internacionalização. A interface gráfica moderna e responsiva proporciona uma experiência de usuário agradável.

Do ponto de vista educacional, o projeto alcançou seu objetivo de ilustrar conceitos fundamentais de forma prática e tangível. A combinação de teoria de grafos, algoritmos, estruturas de dados e interface gráfica em um único projeto coeso oferece uma visão integrada do desenvolvimento de software.

As análises de complexidade demonstram que as soluções implementadas são eficientes, com tempos de resposta adequados mesmo para dicionários grandes. As otimizações aplicadas, como o uso de arrays primitivos e a limitação de elementos visuais, contribuem para a performance geral do sistema.

Os resultados obtidos validam a abordagem proposta e demonstram a viabilidade de aplicar técnicas avançadas de ciência da computação em contextos lúdicos e educacionais. O projeto serve como uma base sólida para extensões futuras, incluindo funcionalidades de multijogador, análise estatística e modos de jogo alternativos.

Em suma, o *WordGraphle* representa uma contribuição significativa tanto como ferramenta educacional quanto como um exemplo de aplicação prática de conceitos teóricos da ciência da computação.

Referências

[1] WARDLE, Josh. **Wordle**. Disponível em: `https://www.nytimes.com/games/wordle`. Acesso em: 5 nov. 2025.

[2] GEE, James Paul. **What video games have to teach us about learning and literacy**. Palgrave Macmillan, 2003.

[3] DIESTEL, Reinhard. **Graph Theory**. 5th ed. Springer, 2017.

[4] CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3rd ed. MIT Press, 2009.

[5] ROSSI, Francesca; VAN BEEK, Peter; WALSH, Toby (Eds.). **Handbook of Constraint Programming**. Elsevier, 2006.

[6] UNICODE CONSORTIUM. **Unicode Standard Annex #15: Unicode Normalization Forms**. Disponível em: `https://unicode.org/reports/tr15/`. Acesso em: 5 nov. 2025.

[7] ORACLE. **Creating a GUI With Swing**. Disponível em: `https://docs.oracle.com/javase/tutorial/uiswing/`. Acesso em: 5 nov. 2025.

[8] BLOCH, Joshua. **Effective Java**. 3rd ed. Addison-Wesley, 2018.

[9] GOODRICH, Michael T.; TAMASSIA, Roberto; GOLDWASSER, Michael H. **Data Structures and Algorithms in Java**. 6th ed. Wiley, 2014.

[10] PRESSMAN, Roger S.; MAXIM, Bruce R. **Software Engineering: A Practitioner's Approach**. 9th ed. McGraw-Hill, 2020.

A Instalação e Execução

A.1 Requisitos do Sistema

- Java Development Kit (JDK) 17 ou superior.
- Sistema operacional: Windows, Linux ou macOS.
- Memória RAM: Mínimo 512 MB.
- Resolução de tela: Mínimo 1024x768.

A.2 Compilação

Para compilar o projeto, execute no diretório raiz (assumindo que o arquivo '.txt' do dicionário esteja presente):

```
# Cria os diretórios de saída se não existirem
mkdir -p bin
# Compila todos os .java para o diretório bin
javac -d bin wordgraphle/*.java wordgraphle/ui/*.java \
    wordgraphle/engine/*.java wordgraphle/graph/*.java \
    wordgraphle/model/*.java wordgraphle/solver/*.java
```

Listing 14: Comando de Compilação (Portável)

A.3 Execução

Para executar o jogo (a partir do diretório raiz):

```
# Adiciona o diretório 'bin' ao classpath
java -cp bin wordgraphle.WordGraphleApp
```

Listing 15: Comando de Execução

A.4 Arquivo de Dicionário

O arquivo `palavras.txt` (ou similar) deve conter uma palavra por linha, em UTF-8. Exemplo:

```
ÁGUA
ÁRVORE
AÇÃO
BAIÃO
CAFÉ
...
```

Listing 16: Exemplo de palavras.txt

B Exemplos de Uso

B.1 Exemplo de Partida Completa

Suponha que a palavra secreta seja **GRAFO**:

1. Tentativa 1: TEMAS → 

2. Tentativa 2: BARCO → 

3. Tentativa 3: GRAMO → 

4. Tentativa 4: GRAFO → 

B.2 Exemplo de Restrições Complexas (Corrigido)

Este exemplo demonstra o tratamento de letras repetidas no palpite.

- Palavra secreta: **BANCO** (Contém: A:1, B:1, C:1, N:1, O:1)
- Tentativa: **CANOA** (Contém: A:2, C:1, N:1, O:1)
- Feedback: 

- Restrições estabelecidas:
  - $fixed[1] = A$ (Verde)
  - $fixed[2] = N$ (Verde)
  - $bannedPos[0][C] = true$ (Amarelo)
  - $bannedPos[3][O] = true$ (Amarelo)
  - $bannedPos[4][A] = true$ (Cinza)
  - $minCount[C] = 1$
  - $minCount[O] = 1$
  - $minCount[A] = 1$
  - $maxCount[A] = 1$ (Definido porque a segunda ocorrência 'A' no palpite foi Cinza, indicando que 'A' não aparece mais que 1 vez).

C Código-Fonte Selecionado

C.1 Algoritmo Principal de Avaliação

```java
public Feedback evaluate(String rawGuess) {
    String guess = Dictionary.normalize(rawGuess);
    if (guess.length() != L()) {
        throw new IllegalArgumentException(
            "Digite uma palavra de " + L() + " letras.");
    }

    int L = L();
    FeedbackColor[] colors = new FeedbackColor[L];

    // Conta letras na palavra secreta
    int[] remaining = new int[26];
    for (int i = 0; i < L; i++) {
        int id = Constraints.idx(secret.charAt(i));
        if (id >= 0) remaining[id]++;
    }

    // Primeiro passe: GREEN
    for (int i = 0; i < L; i++) {
        char g = guess.charAt(i);
        if (g == secret.charAt(i)) {
            colors[i] = FeedbackColor.GREEN;
            int id = Constraints.idx(g);
            if(id >= 0) remaining[id]--;
        }
    }

    // Segundo passe: YELLOW ou GRAY
    for (int i = 0; i < L; i++) {
        // Pula se já foi marcado como GREEN
        if (colors[i] != null) continue;

        int id = Constraints.idx(guess.charAt(i));
        if (id < 0) { // Caractere inválido
             colors[i] = FeedbackColor.GRAY;
             continue;
        }

        if (remaining[id] > 0) {
            colors[i] = FeedbackColor.YELLOW;
            remaining[id]--;
        } else {
            colors[i] = FeedbackColor.GRAY;
```

```
44          }
45      }
46
47      // Atualiza o sistema de restrições com base no feedback
48      applyFeedbackToConstraints(guess, colors);
49      return new Feedback(guess, colors);
50  }
```

Listing 17: GameEngine.evaluate() - Versão Completa