Terceiro Trabalho Avaliativo

Estruturas de Dados Básicas I Instituto Metrópole Digital

2020.6

1 Introdução

Neste trabalho você terá que implementar um **corretor ortográfico** utilizando algumas das estruturas que foram vistas durante o semestre.

2 Descrição

Em computação, um corretor ortográfico é um programa que checa por palavras erradas e sugere uma ou várias palavras corretas para substituí-la. O uso de corretores ortográficos é bem comum em programas de edição de texto, clientes de e-mail, dicionários eletrônicos e motores de busca.

O funcionamento de um corretor ortográfico básico segue o seguinte procedimento:

1. Primeiro ele escaneia o texto e extrai as palavras dele;
2. Depois cada palavra é comparada com uma lista de palavras escritas corretamente (um dicionário, por exemplo);
3. Ao encontrar uma palavra errada, procura pela palavra mais **próxima** e a sugere como correção. Também é possível sugerir um conjunto de várias palavras ao invés de uma.

2.1 Distância entre palavras

Para podermos sugerir uma palavra precisamos de uma métrica para calcular a similaridade entre palavras. Uma das métricas mais utilizadas para sugestão de palavras utiliza o conceito de **distância de edição**. A distância de edição é um valor numérico inteiro que informa a quantidade de operações necessárias para transformar uma palavra em outra. Por exemplo, para transformar a palavra **casa** na palavra **carro**, são necessárias três operações:

1. ca**s**a $\rightarrow$ ca**r**a (substituição de **s** por **r**)
2. car**a** $\rightarrow$ car**r** (substituição de **a** por **r**)
3. carr $\rightarrow$ carr**o** (inserção de **o**)

Para criar um corretor, basta pegarmos as palavras mais próximas da palavra errada (com menor distância) e sugerirmos ao usuário.

2.1.1 Distância Levenshtein

A distância Levenshtein é o algoritmo mais comum quando estamos falando de distância de edição. O funcionamento deste algoritmo foge do escopo deste trabalho, mas sua implementação será fornecida para a implementação do programa[1]:

```
#include <algorithm>
#include <vector>

template<typename T>
typename T::size_type levenshtein(const T &source, const T &target) {
    if (source.size() > target.size()) {
        return levenshtein(target, source);
    }

    using TSizeType = typename T::size_type;
    const TSizeType min_size = source.size(), max_size = target.size();
    std::vector<TSizeType> lev_dist(min_size + 1);

    for (TSizeType i = 0; i <= min_size; ++i) {
        lev_dist[i] = i;
    }

    for (TSizeType j = 1; j <= max_size; ++j) {
        TSizeType previous_diagonal = lev_dist[0],
        ↪  previous_diagonal_save;
        ++lev_dist[0];

        for (TSizeType i = 1; i <= min_size; ++i) {
            previous_diagonal_save = lev_dist[i];
            if (source[i - 1] == target[j - 1]) {
                lev_dist[i] = previous_diagonal;
            } else {
                lev_dist[i] = std::min(std::min(lev_dist[i - 1],
                ↪  lev_dist[i]), previous_diagonal) + 1;
            }
            previous_diagonal = previous_diagonal_save;
        }
    }

    return lev_dist[min_size];
}
```

[1] Disponível em: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#C++

2.2 Detalhes de Implementação

Como este é o último trabalho da disciplina, não haverá restrições, nem detalhes de como o programa deve ser implementado, apenas a definição do problema que vocês devem criar um programa para resolver. Você pode utilizar qualquer estrutura de dados vista em sala de aula: vetor, pilha, fila, deque, tabela de dispersão ou dicionário. Também será possível utilizar as bibliotecas equivalentes disponíveis no STL ao invés das suas próprias implementações. No entanto, utilizar suas próprias estruturas renderá pontos extras.

2.2.1 Definição do problema

Crie um programa que recebe como entrada dois arquivos de texto: um arquivo contendo palavras ortograficamente corretas (dicionário) e outro arquivo que deve ser corrigido (arquivo alvo). O programa deve gerar como saída a lista de palavras escritas incorretamente no arquivo alvo seguida de até 5 sugestões de correção. Uma possível saída é:

```
./programa dicionario.txt teste.txt
izto:
  - isto

caza:
  - casa
  - vaza
  - cara
  - caia
  - coza
...
```

Há varias formas de implementar este programa, umas mais eficientes do que outras. Por isso, além de criar o programa, você deve justificar as escolhas das estruturas de dados utilizadas.

2.2.2 Arquivos de Suporte

Os seguintes arquivos de suporte serão fornecidos para a conclusão do trabalho:

- **dicionario.txt**: Um arquivo contendo várias (mais de 300.000) palavras escritas corretamente no português brasileiro. O arquivo foi adaptado do verificador ortográfico do **Libre Office** disponível em **https://pt-br.libreoffice.org/projetos/vero**.
- **Arquivos alvo**: Três arquivos que deve ser utilizados para validação se o programa funciona corretamente. Um dos três textos é o livro **Dom Casmurro**, que está em domínio público e foi adaptado a partir da versão do Projeto Gutemberg[2].

3 Avaliação

O trabalho possui uma pontuação máxima de 100 pontos e deve ser feito **individualmente**. Esses pontos estão distribuídos da seguinte forma:

- Implementação do programa como descrito — até **80 pontos**
- Justificativa das estruturas de dados utilizadas para resolver o problema — até **20 pontos**

[2] Disponível em: https://www.gutenberg.org/ebooks/55752

Também serão contabilizados pontos extras e descontos:

- **Pontos Extras**:
  - Implementação utilizando orientação à objetos — até **até 10 pontos**
  - Código organizado, bem estruturado e utilizando as boas práticas vistas em sala de aula — até **até 10 pontos**
  - Exibir a linha e a coluna das palavras erradas na saída — até **até 10 pontos**
  - Utilização das estruturas de dados implementadas por você — até **até 20 pontos**
- **Descontos**:
  - Vazamento de memória — até **-10% dos pontos**

4 Colaboração e Plágio

Ajudar o colega de classe é legal, mas copiar seu trabalho, não. Por isso, cópias de trabalhos **não** serão toleradas, resultando em nota **zero** para **todos** os envolvidos.

5 Entrega

O trabalho deve ser entregue em um único arquivo compactado contendo:

- O código-fonte do seu programa
- Um arquivo de texto contendo a justificativa das escolhas das estruturas de dados utilizadas no seu programa

O arquivo deve ser enviado **apenas** pelo *SIGAA* através da opção *Tarefas* até a data divulgada no sistema.