

Corretor Ortográfico

Concepção e Análise de Algoritmos

01/06/2015

André Sousa Lago - up201303313@fe.up.pt

Gustavo Rocha da Silva - up201304143@fe.up.pt

Ricardo Dantas Cerqueira - [up201304000@fe.up.pt](mailto:up201304000@fe.up.pt)

Índice

[Formalização do problema 3](#_Toc420905620)

[Solução 4](#_Toc420905621)

[Análise 4](#_Toc420905622)

[Casos de utilização 6](#_Toc420905623)

[Observações 7](#_Toc420905624)

[Divisão do trabalho 7](#_Toc420905625)

[Referências 8](#_Toc420905626)

# Formalização do problema

Dado um texto T pretende-se, para cada palavra P deste, determinar se P pertence a um conjunto de palavras válidas D. Caso tal não se verifique, pretende-se ainda determinar uma palavra ou conjunto de palavras tal que seja minimizada a distância de edição entre elas.

Seja *T* o texto de entrada, *P* uma sua palavra e *D* um dado conjunto de palavras válidas, pretende-se determinar se:

Caso tal não se verifique, pretende-se encontrar um subconjunto , tal que:

Posteriormente, pretende-se ordenar o conjunto:

De tal maneira que:

# Solução

De  modo a corrigir um dado texto prosseguimos da seguinte forma:

* Para cada palavra do texto, verificar se esta se encontra no dicionário. Sendo este implementado com uma tabela de dispersão, esta operação ocorre com complexidade temporal O(1).
* Se a palavra não se encontrar no dicionário, são procuradas as palavras do dicionário cuja distância de edição à palavra mal escrita é mínima. Para tal foram implementados três algoritmos:

1. Iterar sobre todas as palavras do dicionário e calcular a distância de edição de cada uma à palavra a ser corrigida, não são consideradas palavras cujo tamanho difere significativamente da palavra alvo.
2. Pesquisar todas as palavras com uma distância de edição menor ou igual a um certo máximo pretendido num árvore BK.
3. Pesquisar todas as palavras com uma distância de edição menor ou igual a um certo máximo pretendido num árvore Trie.

* Ordenar os resultados por critérios que visam aumentar a probabilidade de que a palavra efetivamente pretendida apareça como um dos primeiros resultados. Para isso considerou-se primeiramente a menor distância de edição e, em caso de empate, a maior frequência em textos pré-processados da língua a ser analisada.

## Análise

1. **Naive:**

Visto que é efetuado o algoritmo de Levenshtein com programação dinâmica (que tem complexidade temporal O(|T| |P|), sendo T e P as palavras a serem comparadas), para cada palavra do dicionário a complexidade temporal deste método é O(|M| |P| n), sendo n o número de palavras do dicionário e |M| o comprimento médio das palavras do dicionário. Em espaço, a versão do algoritmo utilizada é O(|T| |P|), no entanto, esta pode ser melhorada utilizando um dos próximos métodos.

Não implementamos esta técnica na nossa solução uma vez que dado o tamanho do dicionário e dos textos a corrigir o tempo de execução seria de vários minutos, pelo que consideramos desnecessária a sua aplicação. Tanto pela “teoria” como pela prática observada nos restantes algoritmos, é claro que esta técnica é bastante menos eficiente.

**2 - BK-tree:**

* complexidade temporal: O( log(n) )

Construção:

Pesquisa:

* complexidade espacial: O(n)

Existe um nó por cada palavra do dicionário

**3 - Trie:**

Como todas as palavras da árvore com o mesmo prefixo de tamanho 1 se encontram num mesmo ramo, pode-se reutilizar a matriz do cálculo da distância de edição, acrescentando-lhe uma linha ao visitar um filho de um nó e removendo-a ao voltar para esse nó, sendo apenas necessário calcular uma linha da matriz por letra visitada, sendo cada linha do tamanho da palavra a pesquisar. Para além disso a pesquisa de um ramo é terminada quando a distância de edição entre a palavra atual na árvore e a substring da palavra alvo de igual tamanho é superior a um determinado limite, o que permite evitar visitar todos os nós da árvore,

* complexidade temporal:

Construção: O(|P|.n)  A inserção de cada palavra dá-se em tempo proporcional ao seu comprimento ( O( |P| ) ).

Pesquisa: [1]

* complexidade espacial: O( p ) sendo p o numero de prefixos possíveis do dicionário, no pior caso, em que todas as palavras do dicionário tem prefixos diferentes, será

# Casos de utilização

1. **Escolher o dicionário a utilizar na correção das palavras inseridas**. Os ficheiros de dicionário (ficheiros de texto com as palavras do dicionário, linha a linha) devem ser colocados na pasta “dictionaries” com o nome da língua que representam e sem o caracter “\_” (por exemplo, “eng.txt”). Os ficheiros de dicionários pré processados (que incluem a contagem de ocorrências de cada palavra) devem estar na mesma pasta e com um nome igual ao anterior mas com “\_p” no final (por exemplo, “eng\_p.txt”). Em ambos os casos a extensão deve ser “.txt”. Para a escolha do dicionário, o programa apresenta apenas os dicionários não processados, optando quando possível pelo processado equivalente à escolha do utilizador.
2. **Escolher o ficheiro que contém o texto a ser corrigido**. Os textos a corrigir devem ser colocados na pasta “texts” e devem ter extensão .txt para que o programa os reconheça e apresente.
3. **Para cada erro encontrado, escolher uma das alternativas apresentadas ou adicionar o erro ao dicionário e continuar**. Para cada erro, o programa apresenta uma série de palavras sugeridas e a possibilidade de adicionar a palavra errada ao dicionário, permitindo que esta palavra seja aceite noutras ocorrências posteriores no texto. As palavras sugeridas estão ordenadas segundo:
   1. Distância de edição;
   2. Frequência de utilização da palavra na língua escolhida (obtida através de texto exemplo).
4. **Analisar texto corrigido**. Quando tiverem sido percorridos todos os erros encontrados no texto selecionado, é possível visualizar o resultado obtido no ficheiro “corrected\_text.txt” que se encontra na pasta do executável. Este texto é produzido partindo do original mas tendo em conta as opções do utilizador. O texto é produzido à medida que os erros vão sendo tratados pelo utilizador, pelo que uma interrupção súbita do programa não “perde” todas as correções feitas até ao momento.

Para além disto, no arranque do programa é permitido que o utilizador opte por um modo de benchmark, no qual consegue comparar a duração dos diferentes tipos de algoritmos e estruturas de dados, para um número constante de execuções para cada um.

# Observações

Um dos principais problemas com que nos deparamos no desenvolvimento deste corretor ortográfico foi a necessidade de adaptar o algoritmo de cálculo da distância de edição entre duas strings para considerar trocas de letras adjacentes como tendo peso de 1 e não de 2, por se tratar de um erro frequente ao escrever texto.

Além disso, para garantir que as sugestões são apresentadas por ordem decrescente de probabilidade, além da distância de edição foi tida em conta a frequência com que cada palavra sugerida aparece em textos na língua escolhida. Para isso, foi necessário “treinar” o programa com textos-exemplo.

O terceiro grande problema foi a nível de eficiência (no que diz respeito ao tempo mas também ao espaço), tendo sido utilizadas estruturas de dados adequadas, conforme descrito anteriormente neste relatório.

# Divisão do trabalho

No geral, a divisão das tarefas foi aproximadamente igual para todos os membros do grupo, pelo que procuramos que todo o grupo estivesse envolvido em todas as partes do trabalho.

# Referências

[1] H. Shang and T.H. Merrett. Tries for Approximate String Matching. PhD Dissertation, September 8, 1995

http://www.uio.no/studier/emner/matnat/ifi/INF3800/v12/undervisningsmateriale/tries\_for\_approximate\_string\_matching.pdf