TEMA 8 - ANAGRAMAS

Concepção de algoritmos 2013

Ivo Lima da Silva (201108006) e João Manuel Ferreira Trindade (201109221)

Índice

TEMA 8: ANAGRAMAS

Descrição

OBJECTIVO DO TRABALHO

O objetivo do trabalho é, dada uma palavra gerar todos os anagramas possíveis com as letras dessa palavra também contado com palavras em que só muda uma letra ou até palavras com todas as letras idênticas mais uma nova. Por exemplo: a palavra “amor” tem como anagrama a palavra “Roma”, a palavra “remo” e ainda “obra”.

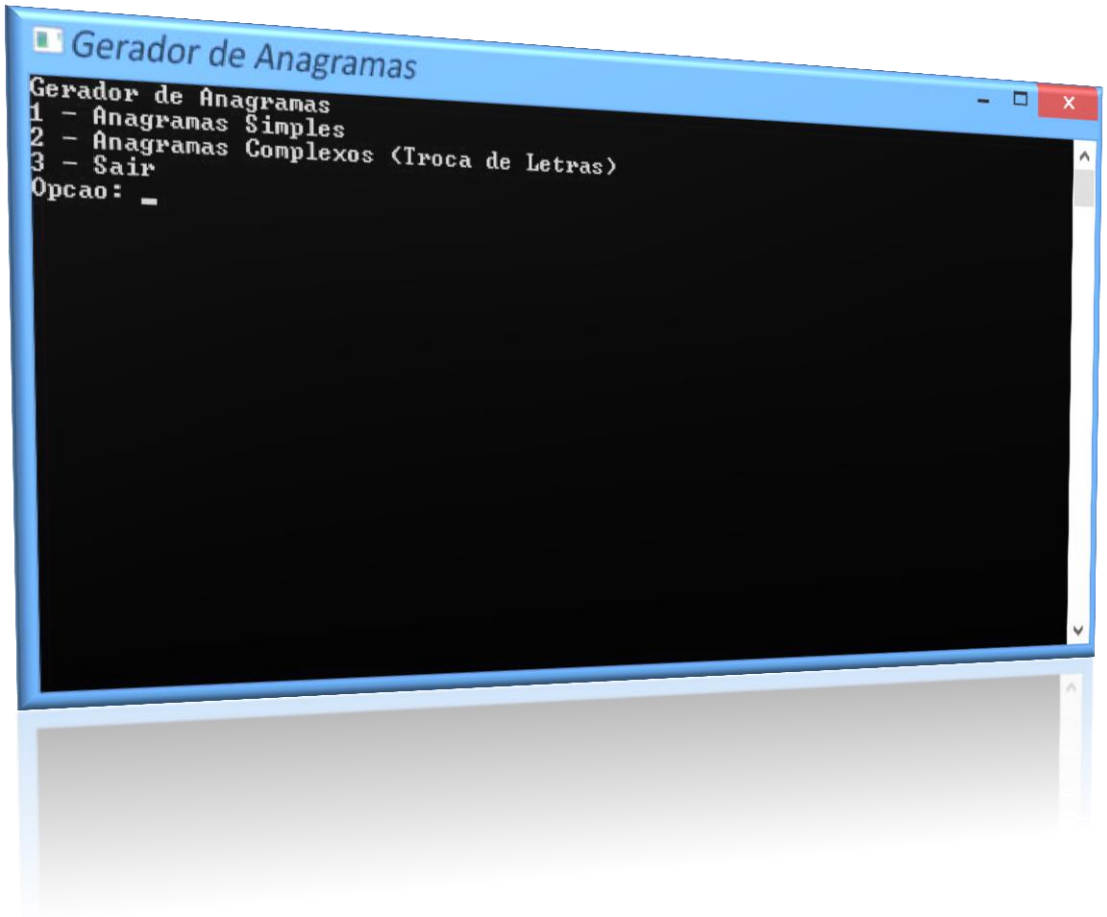
Poderemos dividir o trabalho em duas partes distintas: a primeira, como referido, consiste na geração de anagramas, a segunda, baseia-se na analise desses mesmo anagramas, sua comparação e apresentação ordenada.

Para alcançar o objetivos usamos os algoritmos *Levenshtein* para calcular a distancia dos anagramas à palavra original, de forma a determinar a ordem de impressao. Um algoritmo de pesquisa de anagramas num dicionário, e ainda o tradicional algoritmo de ordenação, quicksort.

TEMA 8: ANAGRAMAS

Manual do utilizador

Aberto o programa deparamos com um menu simples com apenas três funções. É bastante simples orientar-se mas mesmo assim deixamos aqui este manual.



Este é o primeiro ecrã a aparecer e funciona por mensagem, ou seja, escrevemos o número da função que queremos executar e aparecerá um novo ecrã.

Anagramas simples - aqueles anagramas cujas letras e tamanho da palavra se mantém, basta premir **1**.

Anagramas complexos – aqueles anagramas em que pode haver troca de uma letra e/ou adição de outra, basta premir **2**.

Para sair do programa, usa-se a opção **3**.

TEMA 8: ANAGRAMAS

Explicação dos algoritmos usados

IDENTIFICAÇÃO DE ANAGRAMAS

Para determinar um anagrama para uma determinada palavra, executamos uma pesquisa no dicionário por palavras cujo número de caracteres diferentes da palavra original seja igual, ou inferior a um.

Na pesquisa de anagramas simples, pesquisamos apenas as palavras do dicionário com o mesmo número de letras da palavra original. Ordenando alfabeticamente as duas palavras, se estas forem iguais, temos um anagrama.

No caso de anagramas complexos, a estratégia é semelhante, permite-se que, na pesquisa de anagramas simples nem todas as letras tenham que ser iguais. Se existir até uma letra diferente, então a palavra é um anagrama complexo. Se existirem, por exemplo, duas letras diferentes, então essa palavra "candidata" não é um anagrama da palavra original.
Repetimos esta mesma estratégia não só para palavras do mesmo comprimento mas também para palavras com mais, e com menos um caracter.

DISTÂNCIA DE *LEVENSHTEIN*

De forma a calcular a distância de edição entre a palavra indicada pelo utilizador e cada anagrama gerado utilizamos o algoritmo de *Levenshtein*. A distância de *Levenshtein* é a quantidade de caracteres que necessitam de ser editados – adicionados, removidos ou alterados – para que uma dada palavra, ou frase, se transforme noutra.
A implementação deste algoritmo foi feita em programaçao dinâmica, inicialmente utilizando a versão de matriz completa, e posteriormente passamos para a forma mais eficiente em que apenas é necessario um vector.
A técnica deste algoritmo consiste em comparar a distância entre todas as *strings* mais pequenas, que vêm a construir a string final. O valor da comparação de cada uma das strings, mais pequenas, é facilmente derivável a partir do valor de anteriores comparações.
Na implementação em matriz, o resultado de cada comparação entre as *strings* é colocada numa posição da matriz, a primeira linha e primeira coluna dessa matriz, terão valores entre 0 e o tamanho de cada uma das *strings*. Iteramos depois pela matriz, preenchendo cada célula com um, mais o menor valor entre o valor na posição imediatamente acima, à esquerda ou à esquerda e acima (diagonal superior esquerda), que representam, respectivamente, adicionar um caracter, remover um caracter, e alterar um caracter, isto caso os caracteres a comparar sejam diferentes, caso contrário, continuamos a calcular com o valor anterior.
No final da matriz estar completamente preenchida, o resultado da distância estará na ultima célula da matriz.

Na implementação utilizando apenas um vector. A estratégia é a mesma, mas utilizando apenas o correspondente à ultima linha da matriz.

TEMA 8: ANAGRAMAS

Analise de Complexidade

PESQUISA E IDENTIFICAÇAO DE ANAGRAMAS

Os grandes custos computacionais da execução deste programa encontram-se na pesquisa obrigatória ao dicionário de palavras. É importante por isso referir a grande influência do tamanho deste, na eficiência de execução do programa.

De forma a tornar mais eficientes as pesquisas de palavras, importamos todas as palavras do ficheiro "dicionário" para um vector de vectores de palavras. Explicando, temos um vector para palavras de cada tamanho. Ou seja, todas as palavras de 3 caracteres, encontram-se num vector, enquanto as de 4 caracteres encontram-se num outro vector. Todos estes vectores encontram-se por sua vez num outro vector que tem, em cada índice, um vector de palavras com palavras de tamanho igual a esse indice.

Ou seja, na posição 5 do nosso vetor de vectores de palavras, temos o vector com todas as palavras de tamanho 5.

Este consumo de memória permite obter grandes benefícios em termos de eficiência temporal, uma vez que limita a pesquisa de palavras às essenciais.
Supondo que o dicionario consiste em 1000 palavras, distribuidas equitativamente entre palavras de 3,4, 5, 6, e 7 unidades – este exemplo não é totalmente representativo da realidade, mas suficientemente preciso para demonstrar o ganho da pesquisa feita desta forma. Temos portanto, 200 palavras de cada um dos tamanhos.

Se a palavra geraradora de anagrama for de 5 letras, na pesquisa de anagramas simples, temos apenas que pesquisar 200 palavras em 1000. Poupando assim 800 pesquisas face a uma implementação sem divisão das palavras por tamanho.

Para a mesma palavra, na pesquisa de um anagrama complexo, temos a analisar 600 palavras, uma poupança de 400 face a implementação trivial.

A complexidade temporal do algoritmo de identificação de anagramas é, em rigor:

$$\boldsymbol{O(N + N * \log_2 N)}$$

Em que N é o comprimento da palavra. (No caso de anagramas complexos, N é o tamanho da palavra maior.)
A componente $N * \log_2 N$ advém do *quicksort* utilizado para ordenar a palavra. A componente N, linear, da comparação entre as duas palavras.

Uma implementação trivial, em que se comparasse caracter a caracter das duas palavras, não ordenadas teria, no minimo complexidade temporal quadrática, pois seria necessario iterar cada caracter da segunda palavra, para cada caracter da primeira.

TEMA 8: ANAGRAMAS

DISTÂNCIA DE LEVENSHTEIN

Como já referimos anteriormente, o cálculo da distância de edição de Levenshtein foi implementado usando programação dinámica.

Uma implementação nesta estratégia obtem uma complexidade temporal $O(|P| * |T|)$ em que |P| representa o comprimento da primeira palavra e |T|, da segunda.

Em termos de complexidade espacial, implementamos a versão para optimizar o espaço apresentada nas aulas teóricas, tendo conseguido assim uma complexidade $O(|T|)$.

ORDENAÇÃO DE ANAGRAMAS

De forma a ordenar os anagramas gerados, voltamos a recorrer ao algoritmo de *quicksort*, em que o critério de comparação foi a distância das palavras anagramas, à palavra original. A complexidade do algoritmo de *quicksort* é:

$$\boldsymbol{O(N * log_2 N)}$$

Referências

*"**Algoritmos em Strings**" - R. Rossetti, A.P. Rocha, N. Flores*

*"**Introduction to algorithms."** - Thomas H. Cormen, et al.*

http://en.wikipedia.org/wiki/Anagram

http://en.wikipedia.org/wiki/Levenshtein_distance