Contagem Aproximada de Ocorrências – Palavras em Ficheiros de Texto

Pedro Xavier Leite Cavadas, Nº Mec. 85090, MEI

*Resumo* – Este relatório é realizado no âmbito da cadeira Algoritmos Avançados do Mestrado em Engenharia Informática da Universidade de Aveiro.

Neste relatório pretende-se analisar o desempenho de um contador aproximado com probabilidade fixa e de um contador aproximado com probabilidade decrescente logarítmica e compará-los entre si bem como a um contador exato.

Para isso será apresentado, detalhadamente, o problema, a estratégia de resolução utilizada, a solução obtida, bem como testes de *performance* e a análise aos mesmos. Isto é feito aplicando os contadores a palavras de uma dada obra.

*Abstract* – This report is written under the Advanced Algorithms course of the Master in Software Engineering, Universidade de Aveiro.

This is repots aims to analyse the performance of an approximate fixed-probability counter and a logarithmic decreasing-approximate counter and to compare them with an exact counter.

With this in mind, a detailed explanation of the problem will be given, as well as, the resolution strategy, the solution obtained, the results of the performance testing and their respective analysis. This is done applying the counters to words of a given book.

# Introdução

Este relatório é realizado no âmbito da cadeira Algoritmos Avançados onde é nos proposto um problema junstamente com um método de resolução. O objetivo do trabalho proposto é três contadores: um exato, um aproximado com probabilidade fixa e um aproximado com probabilidade decrescente logarítmica. O de probabilidade fixa deverá ter uma probabilidade igual a 1/64 e o de probabilidade decrescente logarítmica deverá ser de base 2. Além disto é necessário realizar-se testes de *performance* que comparem os vários contadores em termos de tempo gasto, memória gasta e exatidão dos valores obtidos. Além disto é necessário também comparar o *top-n* de palavras mais recorrentes para traduções diferentes do livro.

Tendo isto em conta foi escolhido o livro “Alice’s Adventures in Wonderland” de Lewis Carroll. Além da versão em inglês, foram utilizadas as versões em italiano, francês e alemão.

Este projeto divide-se então em duas partes: um s*script python* com a implementação dos contadores e outro que realiza os vários testes sobre os contadores.

Este relatório descrebe então alguns conceitos técnicos necessários para a resolução do problema, uma análise detalhada ao problema, à estratégia, ao algoritmo, à implementação, aos testes e à sua respetiva análise.

No final serão retiradas algumas conclusões acerca dos contadores.

# Conceitos técnicos

# Contador Exato

O tipo de contador mais simples existente. Sempre que se encontra uma ocorrência que se deseja contar incrementa-se o contador uma unidade.

# Contador Aproximado Com Probabilidade Fixa

Um contador que é também ainda relativamente simples. O seu funcionamento é idêntico ao contador exato com a exceção que se incrementa apenas com base numa dada probabilidade. No final para obter o número de ocorrências multiplica-se o valor do contador pelo inverso da probabilidade.

Caso se queira obter um valor mais aproximado do valor real é necessário realizar-se várias contagens e no final o número de eventos é a média do número de ocorrências obtido em cada uma das contagens.

# Contadr Aproximado Com Probabilidade Decrescente Logaritmica

Um contador já um pouco mais complexo, que tem um funcionamento idêntico ao de probabilidade fixa sendo que a principal diferença é o facto de que a probabilidade é dada em função de uma base e do valor atual do contador (probabilidade = ). No final para obter o número de ocorrências aplica-se a seguinte fórmula:

Tal como no contador anterior pode-se calcular a média para obter um valor mais próximo do real.

# Estratégia

Para a resolução do problema proposto, foram então implementados os três contadores indicados. Estes estão definidos em três classes:

* Classe Counter: implementa um contador exato;
* Classe LogarithmicCounter: implementa um contador aproximado com probabilidade decrescente logarítmica;
* Classe ProbabilisticCounter: implementa um contador aproximado com probabilidade fixa.

Estas classes implementam todas os mesmos métodos:

* get: retorna o número de ocorrências estimado;
* increment: incrementa o número de ocorrências;
* reset: reinicializa o contador a 0.

class Counter:

    def \_\_init\_\_(self):

        self.reset()

    def get(self):

        return self.count

    def increment(self):

        self.count += 1

    def reset(self):

        self.count = 0

class ProbabilisticCounter(Counter):

    def \_\_init\_\_(self, a = 2):

        super().\_\_init\_\_()

        self.\_probability = 1 / a

        self.\_a = a

    def get(self):

        return super().get() \* self.\_a

    def increment(self):

        if random.random() < self.\_probability:

            self.count += 1

class LogarithmicCounter(ProbabilisticCounter):

    def get(self):

        b = self.\_a - 1

        return (self.\_a \*\* self.count - b) / b

    def increment(self):

        if random.random() < self.\_probability \*\* self.count:

            self.count += 1

Posto isto, o livro nas suas várias versões é carregado para memória. Neste processo toda a pontuação é removida, as palavras passam a ser todas em minúsculas e são então colocadas todas numa lista pelo ordem em que aparecem no livro. Finalmente algumas palavras como artigos definidos e indefinidos, entre outros, são removidas. Este processo consiste em carregar para memória um ficheiro com as palavras a serem removidas e então removê-las verificando a sua existência nesse ficheiro (este tipo de lista é comumente denominado de *stopwords list*).

def load\_book(path, stopwords, encoding = 'ascii'):

    with open(path, 'r', encoding = encoding) as fin:

        return [ word for word in re.split(r' +', fin.read().translate(translator).strip().lower()) if len(word) > 2 and word not in stopwords ]

    return [ ]

def load\_stopwords(path):

    with open(path, 'r', encoding = 'utf-8') as fin:

        return [ line.strip() for line in fin.readlines() if len(line.strip()) != 0 ]

    return [ ]

def main():

    languages = [ 'french', 'german', 'english', 'italian' ]

    encodings = [ 'utf-8', 'iso-8859-1', 'iso-8859-1', 'iso-8859-1' ]

    stopwords = { language : load\_stopwords('stopwords\\{}.txt'.format(language)) for language in languages }

    books = { language : load\_book('books\\stripped\_{}.txt'.format(language), stopwords[language], encoding = encoding) for language, encoding in zip(languages, encodings) }

Finalmente realiza-se então os testes de *performance*. Nesta fase, cada contador faz a contagem das palavras para cada uma das versões do livro 10, 100, 1000 e 10000 vezes. Para cada uma destas instâncias e para cada uma das versões é calculado o tempo gasto a realizar a contagem, o número de *bits* máximo necessário para o contador (valor real e utilizado, onde o real é um “arredondamento para cima”), além disto para os contadores aproximados são calculadas várias métricas para cada uma das palavras, nomeadamente:

* Errro relativo máximo, erro relativo mínimo, error relativo médio, erro absoluto médio, valor esperado, valor máximo contado, valor mínimo contado, valor médio contado, desvio médio absoluto, desvio máximo, desvio padrão, variância, número esperado de ocorrências e número de ocorrências registado.

De seguida são apresentados os resultados para às 10 primeiras palavras com maior ocurrência, uma média de todas as palavras e os resultados de tempo gasto e de memória máxima necessária por contador.

# Implementação

A implementação do algoritmo e da resolução do problema é feita na linguagem *Python3*. A seguir estão apresentados os módulos implementados e as bibliotecas utilizadas na sua implementação.

# Módulos

1. *counters.py* – módulo que contém as três classes de contadores;
2. *main.py* – este módulo contém as funções para carregar os livros e as listas de palavras a filtrar, bem como funções para realizar os testes de *performance*. Além disto, contém uma função main quando o módulo é utilizado como *main script*.

# Bibliotecas Utilizadas

1. *Itertools* – módulo nativo ao *Python3* que fornece ferramentas para a geração. bem como manipulação de iteradores. Neste trabalho em particular foi utilizada para selecionar as 10 primeiras palavras em conjunto com a função “sorted”;
2. *Sys* – módulo nativo ao *Python3* que fornece ferramentas de interação com o sistema nativo da máquina. Neste projeto foi utilizado para imprimir para o *standard output*;
3. *Math* – Também um módulo nativo ao *Python3*, que fornece ferramentas matemáticas. Utilizado para arredondamentos e cálculos de logaritmos e raízes quadradadas;
4. *Random* – Outro módulo nativo ao *Python3*. Contém ferramentas de geração de valores aleatórios. Neste trabalho utilizado para a geração de números aleatórios utilizados nos contadores aproximados;
5. *Time* – Último módulo nativo ao *Python3* utilizado. Disponibiliza ferramentas que permitem trabalhar com o tempo, tais como obter o *timestamp* atual, entre outros;
6. *Re*  – módulo nativo do *Python3*. Este módulo permite a utilização de expressões regulares e foi utilizado para dividir a *string* com o conteúdo do livro numa lista de *strings*;
7. *String* – módulo nativo do *Python3*. Este módulo contém algumas funcionalidades relacionadas com *strings*. Foi utilizado neste trabalho para remover sinais de pontuação;
8. *Argparse* – módulo nativo do *Python3*. Este módulo permite de forma simples realizar o *parsing* dos argumentos do programa e foi utilizado para detetar se o utilizar quer escrever os resultados em disco e para que ficheiro o quer fazer;
9. *Venv* – módulo nativo do *Python3* (a partir da versão 3.3). Este módulo permite tornar o código *python* portátil e sem necessidade de instalar bibliotecas no ambiente *default* do *python*, o que por outro lado permite remover todas as bibliotecas e módulos não nativos do *python* apagando uma pasta. Neste trabalho é utilizado para que não haja a necessidade de instalar os módulos *prettytable* e *progressbar2*;
10. *Prettytable* – módulo não nativo do *Python3*. Este módulo permite de forma simples imprimir tabelas em ficheiros ou no *standard output* e neste trabalho foi utilizado para mostrar os resultados dos testes;
11. *Progressbar2* – módulo não nativo do *Python3*. Este módulo permite de forma simples imprimir uma *progresso bar* na consola para mostrar ao utilizador o estado do processo em execução. Neste trabalho é utilizado para mostrar o estado dos testes para as diferentes versões do livro.

# Análise de Complexidade

Com base no que foi dito acima, chega-se à conclusão que a complexidade do algoritmo está na ordem de O(*nn*), segundo a notação big-O, isto porque gera-se o produto cartesiano de [ 0 .. *n* – 1] com *n* repetições, o que gera *nn* combinações de cores que dps deverão ser testadas.

# Testes de *Performance*

Para análise de *performance* foi executado um programa de teste que cria vários gráficos aleatórios e calcula o número de operações básicas, o tempo gasto (média de 10 execuções) e as soluções percorridas quando submetido ao algoritmo. Estes gráficos têm um número de vértices entre 1 e 8, sendo que para cada número de vértices existe um grafo com , , e do máximo possível de arestas para o dado número de vértices (no caso desse número ser inferior a 4, o número de arestas é arredondado para baixo).

Por fim, os resultados são guardados num ficheiros e apresentados em dois gráficos, um que mostra a relação entre o número de vértices e o o tempo gasto, e outro que mostra a relação entre o número de vértices e o número de operações básicas (ambos com o número máximo de arestas).

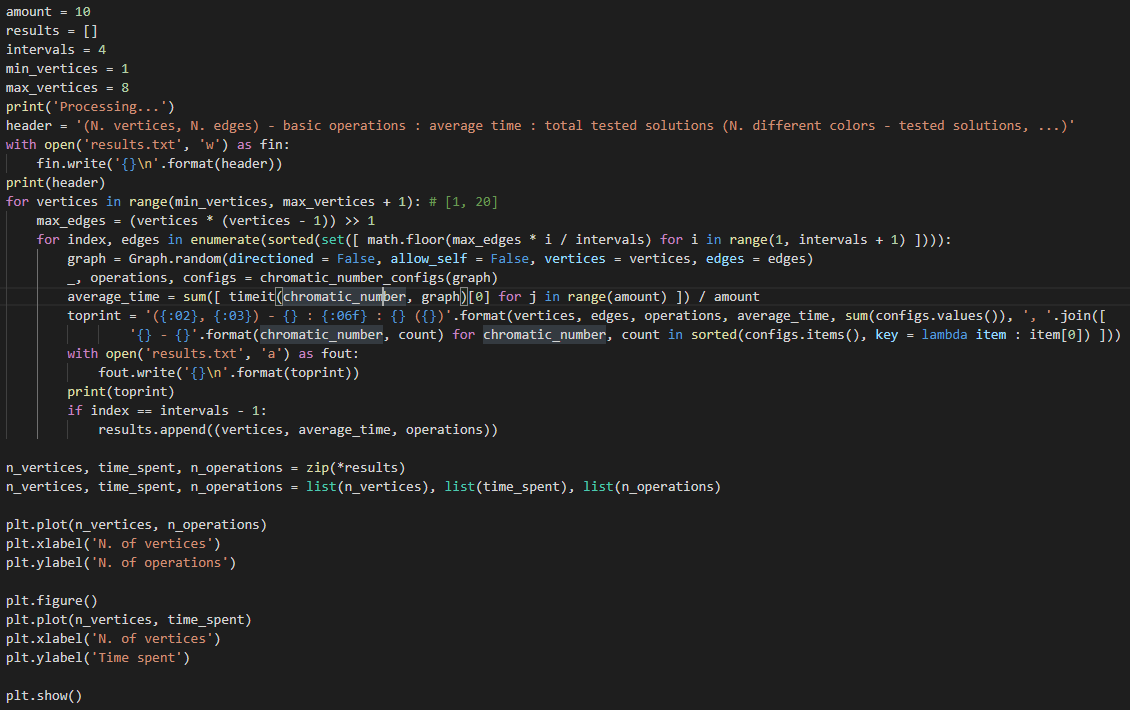


Figura - código de teste de performance

# resultados

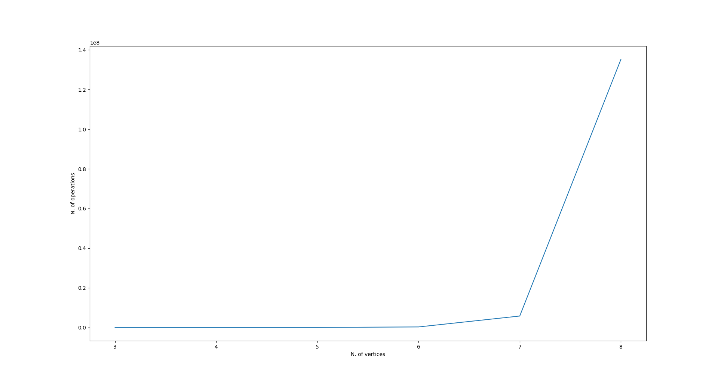


Figura - Relação entre o número de vértices e o número de operações básicas

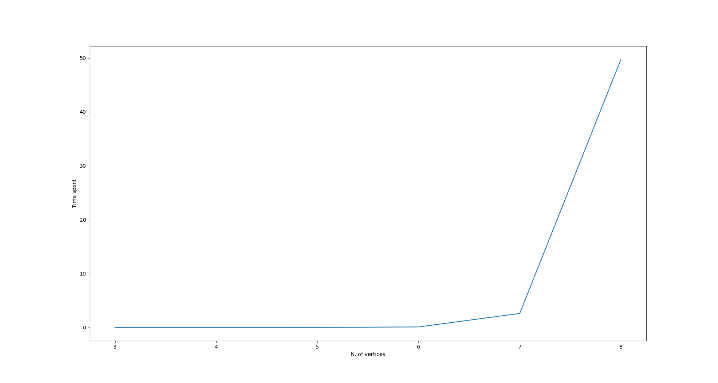


Figura - Relação entre o número de vértices e o número de operações básicas

# Análise dos Resultados

Pelo gráfico da relação entre o número de vértices e o número de operações básicas, pudemos observar que a relação entre os dois é dada por f(*n*) 1.15 \* (*n* – 1) \* *nn* (isto para o número máxismo de arestas, que é dado por .

Para conseguir prever o tempo para um dado número de vértices calculamos precisams primeiro de descobrir a relação entre o número de operações e o tempo gasto, para isto pudemos definir uma relação t(*n*) = *m* \* f(*n*), onde f(*n*) é o número de operações básicas para um dado número de vértices, t(*n*) o tempo gasto para esse mesmo número de vértices e *a* descrebe a relação entre o tempo gasto e o número de operações. Tendo 4 como o número de vértices, temos que:

0.0004 = *m* \* 820 ⬄ m = 0.0004 / 820 ⬄ m 4.88e-0.7

Com *m* calculado e a função f, pudemos finalmente calcular o tempo gasto para completar o algoritmo um dado grafp *n* número de vértices e com o número máximo de arestas para esse mesmo vértice. Por exemplo, vamos supor que *n* = 100, então o tempo gasto é dado por:

t(100) = 4.88e-0.7 \* f(*100*) 5.55e+195 segundos

Ou seja, o tempo necessário para calcular o número cromático de um grafo com 100 vértices e número máximo de arestas (4950), utilizando pesquisa exaustiva, é de aproximadamente 6.43e+190 dias.

# Conclusão

Com este trabalho pudemos concluir que utilizar algoritmos de pesquisa exaustiva para é inviável para instâncias de um problema com uma dimensão elevada. Por vezes, mesmo para uma instância com uma dimensão razoável, utilizar algoritmos de pesquisa exaustiva é impensável. Para verificar isto basta olhar o exemplo dado neste trabalho: para calcular o número cromático de um grafo com 100 vértices e número máximo de arestas (4950), utilizando pesquisa exaustiva, é de aproximadamente 6.43e+190 dias. Isto são cerca de 1.76e+188 anos; para se ter uma ideia, estima-se que o nosso universo tenha 13.7e+09 anos. Olhando para estes números torna-se bem claro que de facto o tempo que estes algoritmos demoram a executar tornam-nos inviáveis para resolver uma grande parte dos problemas.

# Referências

[1] <https://pt.wikipedia.org/wiki/Teoria_dos_grafos>

[2] <https://pt.wikipedia.org/wiki/Colora%C3%A7%C3%A3o_de_grafos>

[3] <http://www.educ.fc.ul.pt/icm/icm2001/icm33/grafosnaoorientados.htm>

[4] <https://en.wikipedia.org/wiki/Big_O_notation>

[5] <https://matplotlib.org/contents.html>