

Universidade do Minho

LEI 2º Ano 2º Semestre

LI3 - Projeto em JAVA

Gestauts

Uma aplicação de criação, gestão e consulta de uma Rede de Autores

Grupo 12

Tiago Cunha a67741, Daniel Caldas a67691, Xavier Rodrigues a67676

19 de Junho de 2014



Conteúdo

1	Introdução	3
2	Classes de Gestauts	3
2.1	Diagrama de Classes	3
2.2	Descrição	3
2.2.1	Documentação javadoc (ver anexo)	4
2.3	GestautsUI - Método main	5
2.3.1	Os métodos	5
3	Estrutura de dados	6
3.1	Encapsulamento	7
4	Interface do utilizador	8
4.1	Opções de Navegação	8
5	Medidas de Performance	9
5.1	Informações da máquina que correu os testes	9
5.2	I - Tempos de Leitura	9
5.3	II - Comparar Coleções do Java	11
6	Conclusão	16

1 Introdução

Na linha de desenvolvimento do projeto anterior onde na linguagem **C** criámos uma aplicação capaz de gerir ficheiros de publicações através da implementação de **módulos** (encapsulados). É nos colocado agora o desafio de se desenvolver uma aplicação muito semelhante à anterior mas, desta vez na linguagem **JAVA**, e na linha dos princípios da **Programação Orientada aos Objectos**. Veremos que o esforço de codificação para desenvolvimento de uma API *com tais propriedades* é reduzido quando a "ferramenta" passa a ser o JAVA, pois é uma linguagem que naturalmente nos providencia modularidade e encapsulamento. O objetivo principal deste projeto consiste sobretudo em **consolidar** os princípios do novo (para nós) **paradigma** e do uso das **coleções do JAVA** (JCF *Java Collections Framework*). Também é de salientar a parte deste projeto onde são feitos os testes de performance.

2 Classes de Gestauts

2.1 Diagrama de Classes

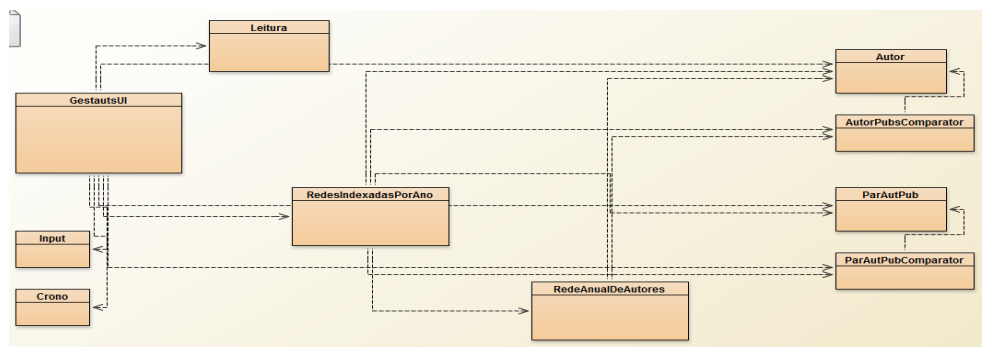


Figura 1: Diagrama de classes, o esqueleto da API.

2.2 Descrição

- **GestautsUI** - Única classe de *interface do utilizador (UI)*, é através desta que o utilizador **escolhe** o ficheiro *publicx* que quer **ler** ou **carregar de um ficheiro objeto**. É nesta classe que é posto um menu de queries interativas à disposição do utilizador em função de três parâmetros *Estatística*, *Consultas num intervalo de anos (ou ano)* e *Consultas globais especiais*. Ainda teremos oportunidade de ver com mais detalhe esta classe neste relatório.
- **Leitura** - Classe onde é feito o **processo de leitura** do ficheiro em si **simultaneamente é feita a população da estrutura de dados**.
- **RedesIndexadasPorAno** - Esta classe dá tipo ao mais abstrato objeto (*aquele que é guardado em ficheiro*) do nosso projeto. Nesta classe

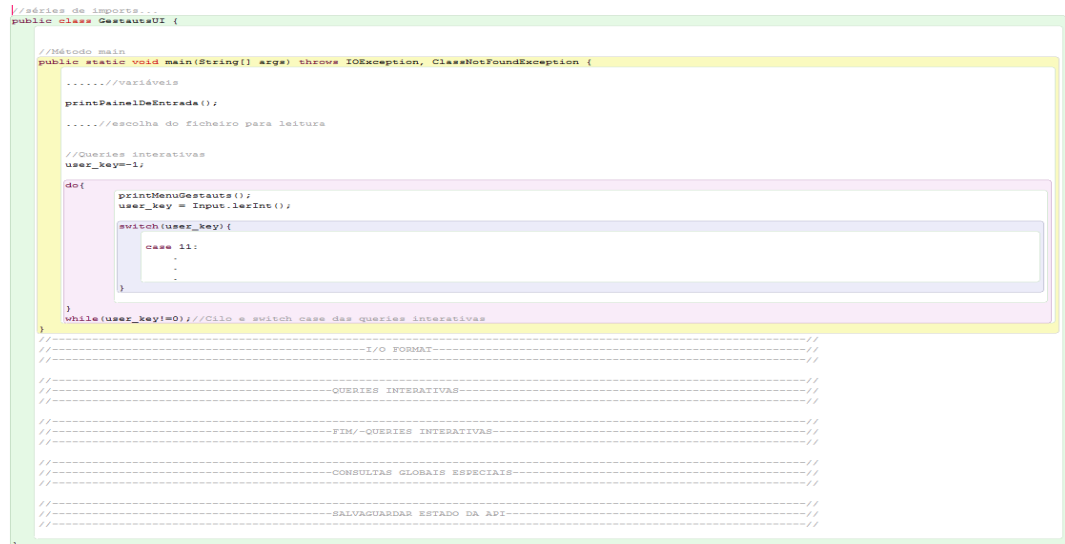
para além de tipos simples como `int minYear` (etc. ...) encontra-se um **TreeMap**<**Integer**,**RedeAnualDeAutores**> que a cada ano (chave do mapa) associa uma rede anual de autores que contém informações relevantes sobre as publicações desse ano (valor do mapa).

- **RedeAnualDeAutores** - Uma classe que contempla informação apenas de um ano. Um **TreeMap**<**String**,**Autor**> é aqui o centro das atenções que armazena num mapa **as instâncias** de autor associadas aos autores que publicaram nesse ano.
- **Autor** - Esta classe é uma base de dados de um autor, guarda para um determinado ano o nº de publicações desse autor, estas discriminadas como *a solo* ou *não a solo*, contém também informação acerca dos **coautores** do autor, e claro o nome do autor em questão.
- **AutorPubsComparator** - Compara dois autores pelo seu nº de publicações, caso o nº de publicações seja igual compara os nomes (*este comparator está implementado de modo que os autores fiquem ordenados por ordem decrescente do nº de publicações*).
- **ParAutPub** - Associa um par de autores e o seu nº de coautorias. Classe muito útil para **Querie 22** (X pares de autores).
- **ParAutPubComparator** - A mesma essência do comparator *AutorPubsComparator* mas desta vez fornece um algoritmo de comparação para pares de autores.
- **Crono** e **Input** - Duas classes muito úteis disponibilizadas pela equipa docente, a primeira simplifica as medidas do tempo de execução frequentes neste projeto, a segunda abstrai o uso da ferramenta *Scanner* fazendo um tratamento de erros de tipos interno à classe.

2.2.1 Documentação javadoc (ver anexo)

Por uma questão de organização e apresentação deste relatório, a **documentação javadoc** do projeto encontra-se em anexo para consulta.

2.3 GestautsUI - Método main



```
//séries de imports...
public class GestautsUI {

    //Método main
    public static void main(String[] args) throws IOException, ClassNotFoundException {

        .....//variáveis
        printPainelDeEntrada();
        .....//escolha do ficheiro para leitura

        //Queries interativas
        user_key=-1;

        do{
            printMenuGestauts();
            user_key = Input.lerInt();

            switch(user_key){

                case 11:
                    .....
                .....
            }

        } while(user_key!=0); //Ciclo e switch case das queries interativas

        .....//-----I/O FORMAT-----
        .....//-----QUERIES INTERATIVAS-----
        .....//-----FIM/-QUERIES INTERATIVAS-----
        .....//-----CONSULTAS GLOBAIS ESPECIAIS-----
        .....//-----SALVAGUARDAR ESTADO DA API-----
    }
}
```

Figura 2: Um bom resumo da estrutura e tipos de métodos que podemos encontrar na classe **GestautsUI**.

O nosso método main está bem estruturado pois encontra-se **seccionado por funcionalidades**, além disso dentro do próprio método main é **bem claro** quais as queries que queremos executar e em que contexto. O grupo decidiu dividir o controlo de I/O das diferentes queries por métodos auxiliares cujo nome é **standardizado** *Queriexx* (xx - número correspondente no switch case).

2.3.1 Os métodos

Vamos explicitar com mais detalhe alguns dos métodos mais relevantes desta classe...

public static void printMenuGestauts() - Este método sempre que invocado imprime no terminal o menu principal do Gestauts.

public static void Querie11 (RedesIndexadasPorAno ripa) - O método *Querie11* (assim como todos os outros) recebe como parâmetro uma instância das redes anuais de autores indexadas, para que seja consultada no contexto da query.

public static void lerLinhasDuplicado(String filename) throws FileNotFoundException - este método faz parte do conjunto de queries pedidos no enunciado deste projeto, tendo este uma particularidade. *lerLinhasEmDuplicado* implica que se releia o ficheiro, por tal motivo este método não é um método de instância da classe *RedesIndexadasPorAno*, mas sim um método static auxiliar da main.

public static RedesIndexadasPorAno open() throws IOException, ClassNotFoundException - método para carregar estado anteriormente gravado do programa num objecto *RedesIndexadasPorAno*.

public static void save(RedesIndexadasPorAno ripa) throws IOException - gravar o estado da nossa aplicação num ficheiro .obj.

3 Estrutura de dados

Para explicarmos a solução que implementámos para a resolução do problema vamos explicar detalhadamente três classes em simultâneo, isto porque elas são **criadas por composição simples de classes** entre si.

São as classes **RedesIndexadasPorAno**, **RedeAnualDeAutores** e **Autor** que formam na verdade esta *rede de autores*. No seguinte esquema da estrutura de dados que apresentámos encontra-se toda a informação relevante de cada uma das classes e como estas interagem entre si. O nível **mais abstracto** encontra-se à esquerda e o esquema vai mostrando como é composta a nossa rede de autores desde uma instância de **Autor** até à rede em si.

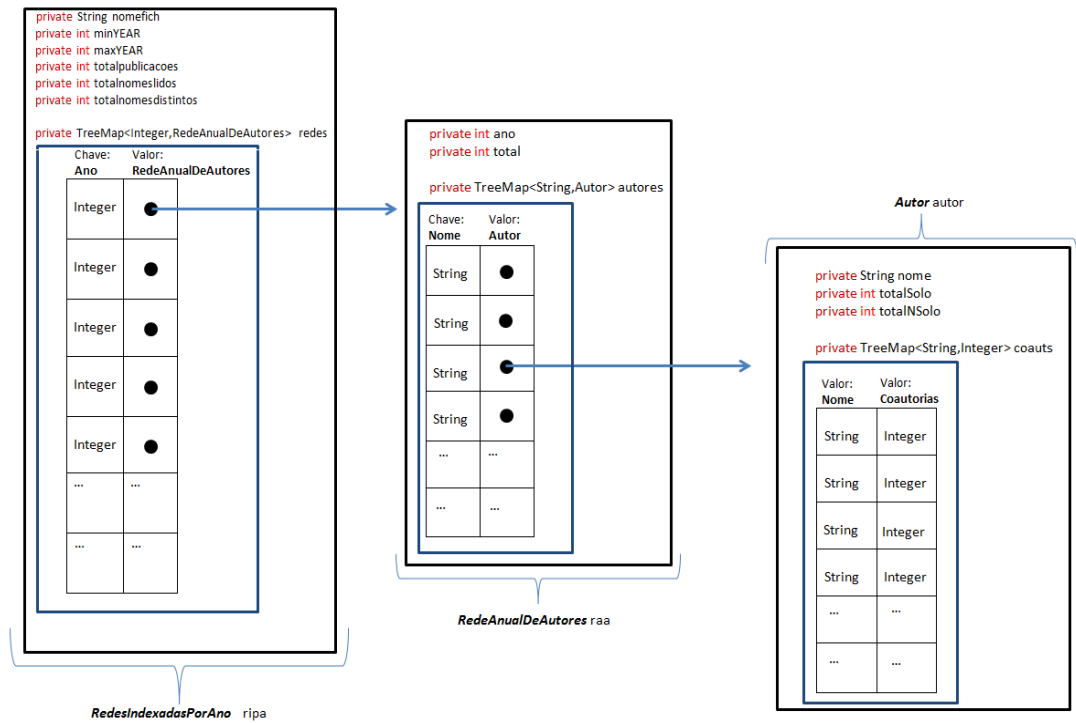


Figura 3: A estrutura de dados de Gestauts.

3.1 Encapsulamento

Toda esta "comunicação" entre classes tem de ser construído com base no princípio do encapsulamento dos dados, *i.e* nunca esqueçamos que cada classe tens as suas variáveis (**privadas**) de instância, pelo que apenas instâncias da classe podem alterar essas variáveis!

Um exemplo de encapsulamento explícito dá-se por exemplo a quando da leitura de uma linha do ficheiro e a inserção dos autores no *TreeMap* de autores da rede de publicações do ano em questão, pois vejamos:

Shufeng Wang, Hong Zhu, 2012, ao lermos esta linha do ficheiro eis o que se sucede:

```
ripa.addPublicacao(autores,ano), estamos a adicionar à nossa rede os au-  
tores lidos (autores é um HashSet<String>) - classe Leitura;  
rededoano = this.redes.get(ano);  
rededoano.incTotalPublicacoesDoAno();  
rededoano.addAutoresRedeDoAno(autores);
```

Em cima estamos a adicionar à rede do ano os autores - classe **RedesIndexadasPorAno**;

this.autores.put(a.getNome(),a) adicionamos ao *TreeMap* de autores desta rede um dos autores passados como parâmetro pelo método acima...

Basicamente, o encapsulamento atua como uma camada de proteção do objeto.



Figura 4: Pequena esquematização do conceito de encapsulamento, **abstração** & **especialização** de classes.

4 Interface do utilizador

A interface Gestauts mantém-se praticamente inalterada desde 1º projeto. O utilizador tem acesso a um menu indexado de acordo com as queries do enunciado do projeto.

```
Tempo leitura do ficheiro e população da estrutura de dados: 1.432469741 segundos

#####RELATORIO#####
Nome do ficheiro lido: publica.txt
Total de artigos publicados: 136127
Total de nomes lidos: 368014
Total de nomes distintos: 164250
Intervalo de anos das publicações: [1969,2013]

#####
##          GESTAUTS      Menu          ##
#####

-I-Consultas Estatisticas-
11 - Total de artigos publicados por um dado autor
12 - Total de autores que apenas publicaram a solo
13 - Total de autores que nunca publicaram a solo
14 - Total de autores que publicaram mais de X artigos no intervalo de anos
15 - Consultar tabela de publicações

-II-Consultas indexadas por ano ou ano-
21 - Nomes dos X autores que mais publicaram
22 - X pares de autores com mais coautorias
23 - Coautores comuns a todos os autores de uma dada lista
24 - Listar todos os autores que publicaram num dado intervalo de anos

-III-Consultas globais especiais-
31 - Número de linhas em duplicado no ficheiro
32 - Nomes de todos os autores começados por uma determinada letra
33 - Determinar coautorias dado um autor e um ano
34 - Todos os coautores de um dado autor

4 - Gravar em ficheiro o estado da aplicação
0 - Sair
>
```

Figura 5: Menu principal da API.

4.1 Opções de Navegação

Tal como no primeiro projeto dá-mos a possibilidade ao utilizador de navegar através de páginas para que seja mais legível o resultado de queries que listam intermináveis quantidades de nomes ou até mesmo pares de autores.

```
Tamanho da lista: 100

- Autores: Haridimos T. Vergos & Dimitris Nikolas Coautorias: 14
- Autores: Haridimos T. Vergos & Costas Efstatthiou Coautorias: 14
- Autores: Hermann Mey & Daniel Kayzers Coautorias: 14
- Autores: Hyun-Chul Kim & Daijin Kim Coautorias: 14
- Autores: Isaac Cohen & GSheard G. Medioni Coautorias: 14
- Autores: James R. Wilson & Emily K. Lada Coautorias: 14
- Autores: Jiang-Yu Yang & David Zhang Coautorias: 14
- Autores: Jonathan J. Hull & Dar-Rhyang Lee Coautorias: 14
- Autores: Kae-Young Yoo & Eun-Won Yoon Coautorias: 14
- Autores: Kae-Young Yoo & Eun-Kyung Ryu Coautorias: 14
- Autores: Keng Pang Lim & Feng Pan Coautorias: 14
- Autores: Lile Lu & Hong-Tiang Zhang Coautorias: 14
- Autores: Marco Ajmons Marsan & Fabio Neri Coautorias: 14
- Autores: Matthew R. Boutell & Niebo Luo Coautorias: 14
- Autores: Mehdi Dehghan & M. Marjed-Jamei Coautorias: 14
- Autores: Michael G. McComas & Arvill M. Lav Coautorias: 14
- Autores: Michraeg Potkonjak & Jennifer L. Wong Coautorias: 14
- Autores: Mohamed Ould-Thaous & Lewis M. Mackenzie Coautorias: 14
- Autores: Naokazu Yokoya & Haruo Takemura Coautorias: 14
- Autores: Narayanan Vijaykrishnan & Mahmut T. Kandemir Coautorias: 14
- Autores: Natalie M. Steiger & James R. Wilson Coautorias: 14
- Autores: Nicu Sebe & Michael S. Lev Coautorias: 14
- Autores: Niraj K. Jha & Anand Raghunathan Coautorias: 14
- Autores: Philip L. Worthington & Edwin R. Hancock Coautorias: 14

Página: 4/5

1 - Página seguinte 0 - Sair: |
```

Figura 6: Navegação por páginas.

5 Medidas de Performance

5.1 Informações da máquina que correu os testes

ASUS K55V - CPU: intel core i7 3630QM 2.4GHz

Memória: 6GB HDD: 500GB OS: Windows 8.1 Pro IDE: BlueJ 3.1.1

5.2 I - Tempos de Leitura

No primeiro ponto das medidas de performance é nos pedido que seja feita a leitura dos três ficheiros teste com **BufferedReader()** e **Scanner()**, ainda para ambos, com parsing e sem parsing.

BufferedReader()						
Ficheiro	Sem parsing		Com parsing		Com parsing + população da estrutura de dados	
publicx.txt	0.040248062	0.042228008	0.209023677	0.295709804	0.73274967	1.13245024
publicx_x4.txt	0.082707847	0.08280492	0.511456316	0.518362606	2.406002186	2.706278267
publicx_x6.txt	0.129712099	0.137026352	0.847695565	0.838761862	4.095924305	4.115271325

Figura 7: Tabela de tempos (tempo em segundos).

Scanner()						
Ficheiro	Sem parsing		Com parsing		Com parsing + população da estrutura de dados	
publicx.txt	0.204545066	0.21638497	0.379254959	0.390706571	1.157656275	1.873255253
publicx_x4.txt	0.753793542	0.763308827	1.211989747	1.225496996	3.301344738	3.467578068
publicx_x6.txt	1.087170636	1.089618414	1.924016271	1.943543752	5.735464291	5.969123539

Figura 8: Tabela de tempos (tempo em segundos).

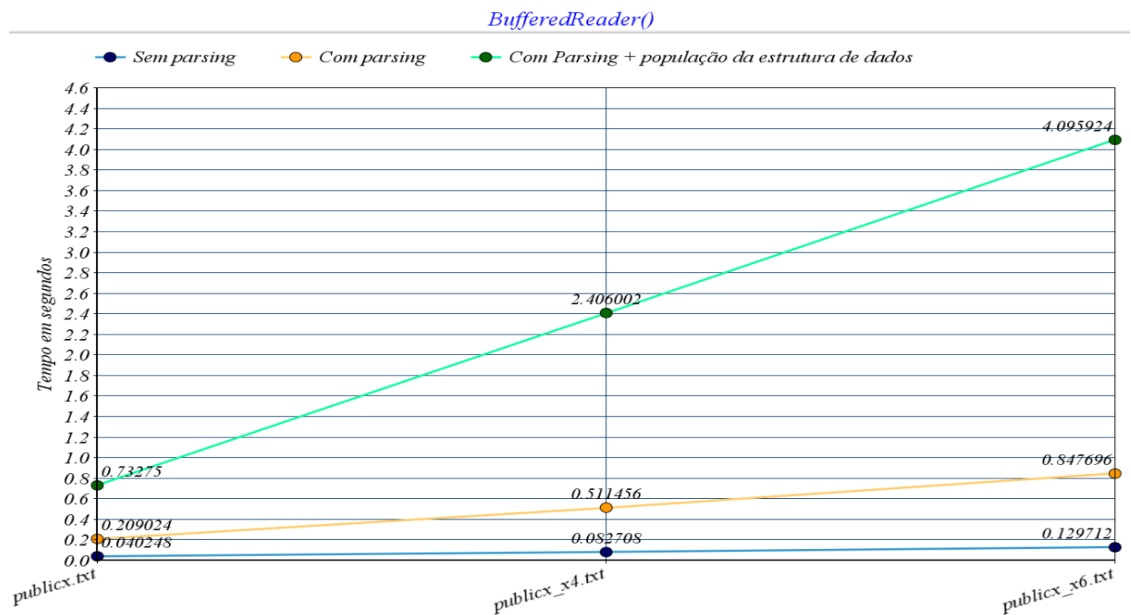


Figura 9: No gráfico podemos ver as variações do **BufferedReader()** em função dos parâmetros pretendidos.

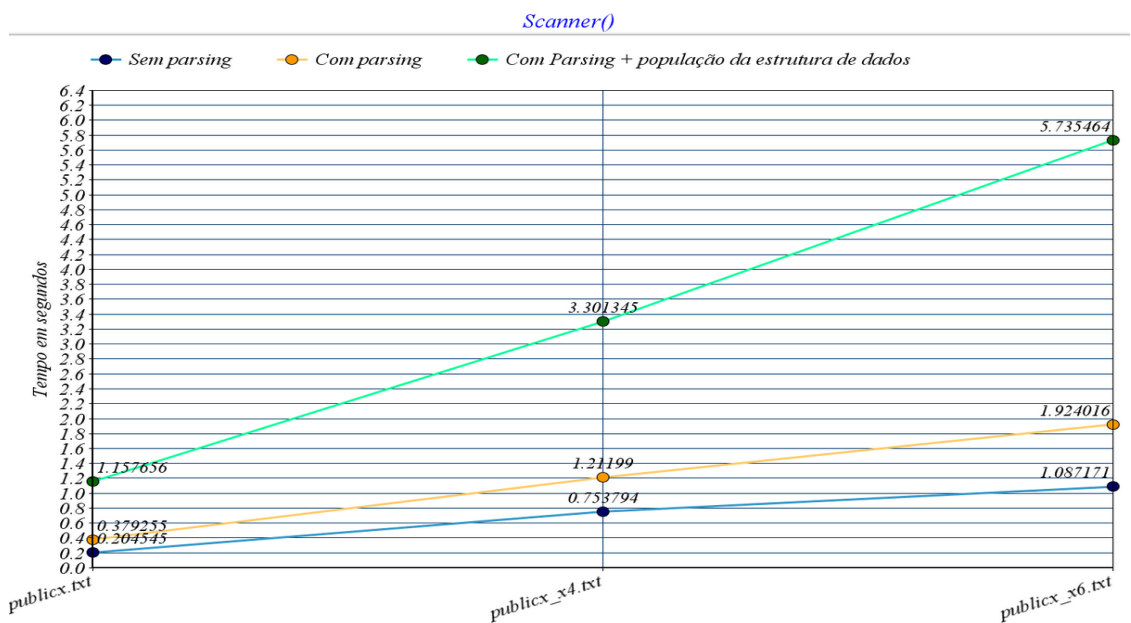


Figura 10: No gráfico podemos ver as variações do **Scanner()** em função dos parâmetros pretendidos.

Comentário: Após efetuados os testes de leitura podemos afirmar o **BufferedReader()** é mais eficiente que o **Scanner()** pois para todos os testes o

primeiro superou o segundo. O grupo ficou satisfeito com o resultado (já esperado) pois no código entregue à equipa docente para avaliação é feita a leitura do ficheiro teste com `BufferedReader()`. Optamos pela ferramenta de leitura mais eficiente. Apesar de tudo fica a nota de que o **`Scanner()`** oferece um maior leque de métodos no que toca a fazer *parse* de um ficheiro, por outro lado o `BufferedReader()` é mais eficiente na leitura linha a linha (que é este caso).

5.3 II - Comparar Coleções do Java

Para estes testes o grupo de trabalho fez medidas de tempo com a API no seu estado original, podemos ver esse estado pela fig.3. Fizemos medições de tempos para **carregar a informação de memória secundária para memória central** e ainda das seguintes Queries:

- **22** - "X pares de autores com mais coautorias" - ***Anos**[1997,2010] X:100;*
- **24** - "Listar todos os autores que publicaram num dado intervalo de anos"
- ***Anos**[1997,2010] ;*
- **32** - "Nomes de todos os autores começados por uma determinada letra"
- ***Letra:** 'a';*

Numa primeira fase de testes (a mais importante) vamos substituir as implementações *TreeMap* por *HashMap* nas classes: **RedesIndexadasPorAno** temos *private TreeMap<Integer,RedeAnualDeAutores> redes* passa a *private HashMap<Integer,RedeAnualDeAutores> redes*, repetimos este passo para as classes **RedeAnualDeAutores** e **Autor**.

Testes com API no estado original (TreeMap<K,V>)								
Ficheiros	Carregar ficheiro em memória central		Querie 22		Querie 24		Querie 32	
publicx.txt	0.6783	0.7214	0.7066	0.8122	0.0312	0.0363	0.2060	0.2283
publicx_x4.txt	2.5916	2.6599	0.8422	0.8493	0.0350	0.0398	0.2153	0.2243
publicx_x6.txt	4.1969	4.2988	0.8508	0.8987	0.0364	0.0411	0.2229	0.2359

Figura 11: Tabela de tempos (tempo em segundos).

Testes com API com colecções modificadas (HashMap<K,V>)								
Ficheiros	Carregar ficheiro em memória central		Querie 22		Querie 24		Querie 32	
publicx.txt	0.4691	0.4914	0.6978	0.8514	0.0306	0.0427	0.2508	0.2649
publicx_x4.txt	1.8141	2.1097	0.8418	0.9151	0.0361	0.0399	0.2355	0.2477
publicx_x6.txt	2.4168	2.5548	0.8782	0.8883	0.0343	0.0389	0.2405	0.2626

Figura 12: Tabela de tempos (tempo em segundos).

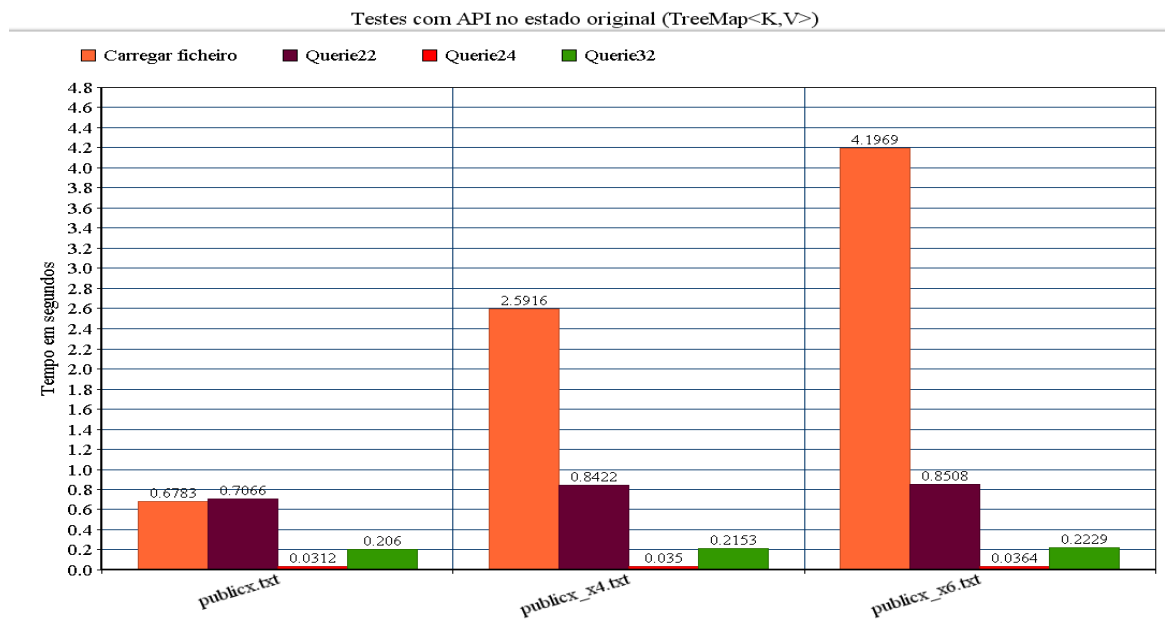


Figura 13: Gráfico produzido através da tabela da fig.11.

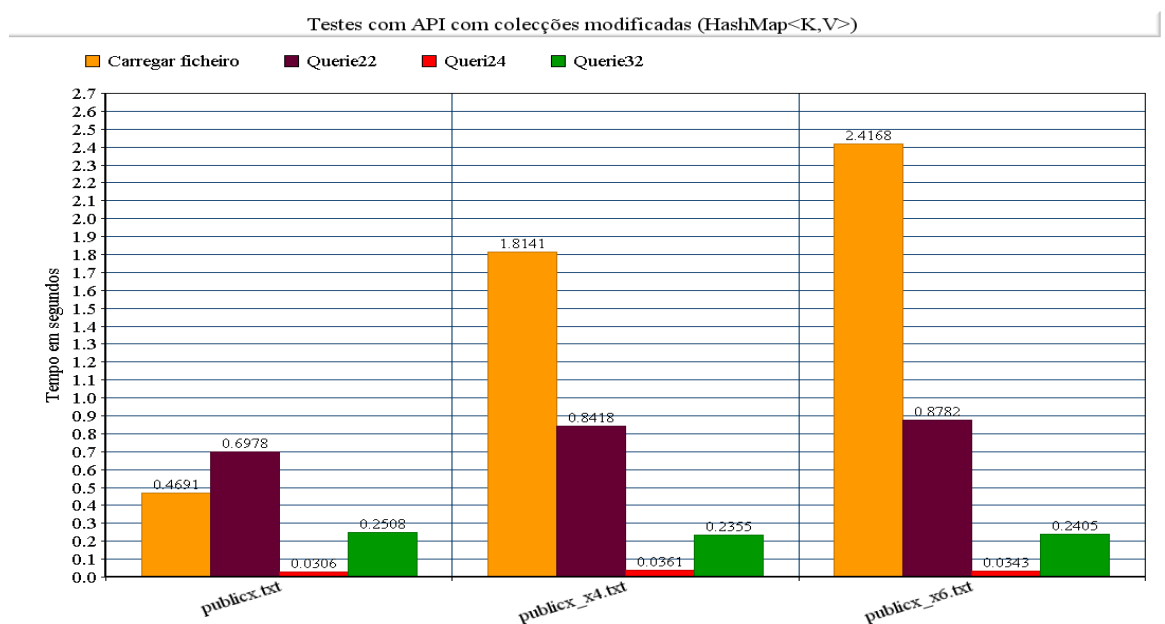


Figura 14: Gráfico produzido através da tabela da fig.12.

Comentário: como podemos comprovar através de medições de tempo da nossa API a coleção **HashMap** é potencialmente rápida que **TreeMap**. Pelos gráficos das fig. 13 e 14 vemos claramente que o gráfico de correspondente ao HashMap efetua o carregamento do ficheiro em memória central muito

mais rápido que o TreeMap! Mas nem tudo joga a favor do HashMap pois em muitas das vezes que codificamos algo queremos que os nossos objectos estejam ordenados, como é o caso da nossa rede de autores. Portanto fica a nota de que devemos aproveitar o potencial da coleção HashMap mas nunca desprezando a utilidade de um mapa ordenado (TreeMap).

O segundo teste feito pelo grupo de trabalho não tem o mesmo impacto no tempo de execução que o que vimos anteriormente. Vamos confrontar em duas Queries, Querie22 e Querie32 o uso do HashSet<E> com o TreeSet<E>, este último o qual utilizávamos devido à necessidade de ordem.

Ficheiros	Testes com API com colecções modificadas (HashSet<E>)			
	Querie 22		Querie 32	
publicx.txt	0.5741	0.6484	0.2027	0.2056
publicx_x4.txt	0.6515	0.6831	0.2088	0.2171
publicx_x6.txt	0.6277	0.6558	0.2056	0.2218

Figura 15: Tabela de tempos (**tempo em segundos**).

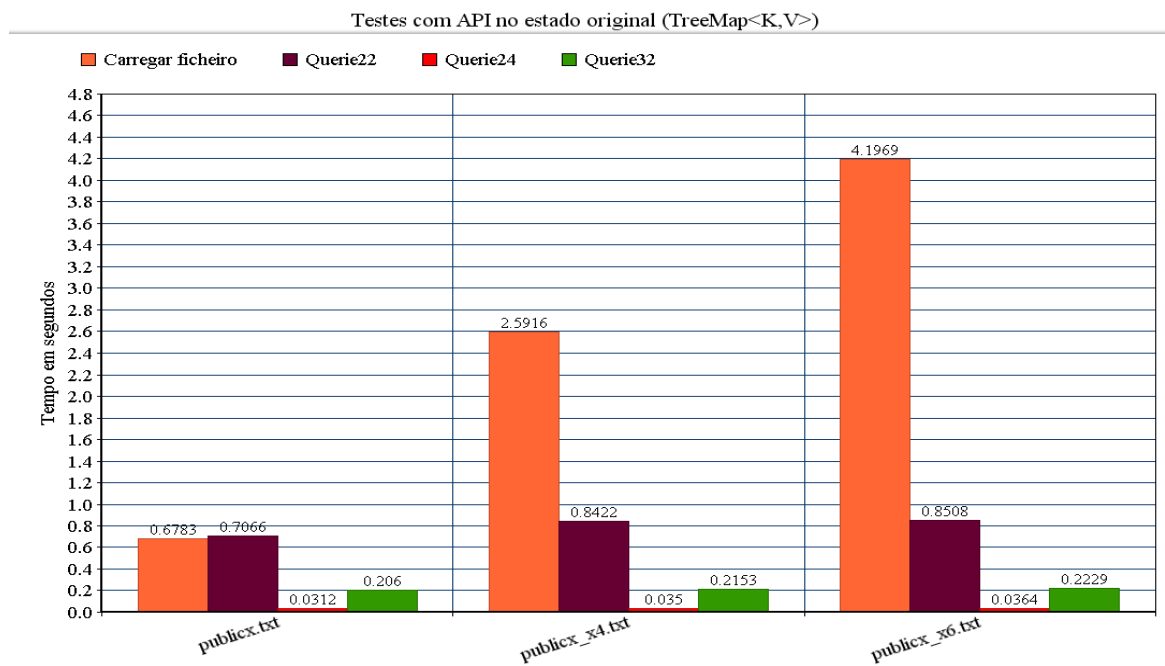


Figura 16: Gráfico produzido através da tabela da fig.13.

Comentário: Mais uma vez somos confrontados com o dilema de necessidade de ordem (TreeSet<E>) *versus* velocidade (HashSet<E>).

6 Conclusão

Mais uma vez tivemos a experiência de lidar com **Abstracção de Dados, Encapsulamento e Modularidade**, características do paradigma da programação por Objectos. Foi feita a consolidação do uso das coleções do JAVA (JCF) genéricas: Iteradores, Tipos, List, Map e Set. Foi também com este projeto que fomos introduzidos à medição de tempos nestes termos, e à comparação do desempenho das diferentes coleções o que será certamente uma arma poderosa num futuro muito próximo, sabermos **o que é** que temos de usar e **e quando** a altura mais apropriada.