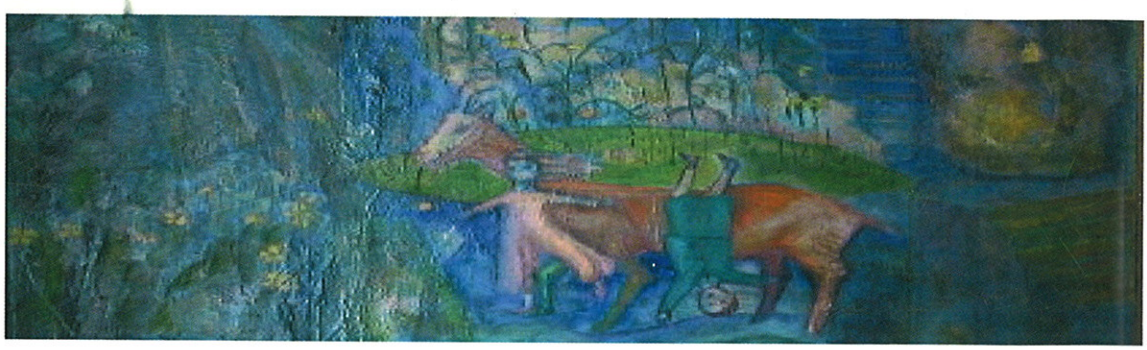
UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO



Medidas da complexidade da estrutura de algoritmos codificados em C

Ana Isabel Torres Garcia Portugal de Almada Cardoso (Mestre)

Dissertação para obtenção do grau de Doutor em Engenharia Informática e de Computadores

Orientador: Professor Doutor Rui Gustavo Nunes Pereira Crespo
Co-Orientador: Professora Doutora Tânya Vianna de Araújo

Júri:
Presidente: Reitor da Universidade Técnica de Lisboa
Vogais: Professor Doutor Hélder Manuel Ferreira Coelho
Professor Doutor Ludwig Paul Ary Evert Streit
Professor Doutor Arlindo Manuel Limede de Oliveira
Professora Doutora Tânya Vianna de Araújo
Professor Doutor Rui Gustavo Nunes Pereira Crespo
Professor Doutor António Manuel Ferreira Rito da Silva

Julho de 2004

UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Medidas da complexidade da estrutura de algoritmos codificados em C

Orientador: Professor Doutor Rui Gustavo Nunes Pereira Crespo

Co-Orientador: Professora Doutora Tânya Vianna de Araújo

Júri:

Presidente: Reitor da Universidade Técnica de Lisboa

Vogais: Professor Doutor Hélder Manuel Ferreira Coelho

Professor Doutor Ludwig Paul Ary Evert Streit

Professor Doutor Arlindo Manuel Limede de Oliveira

Professora Doutora Tânya Vianna de Araújo

Professor Doutor Rui Gustavo Nunes Pereira Crespo

Professor Doutor António Manuel Ferreira Rito da Silva

Julho de 2004

Ao meu pai e à tia Teresa que tanta falta me fazem.
Aos meus filhos, justificação primeira de todo o esforço de progresso da minha vida.

Titulo: Medidas da complexidade da estrutura de algoritmos codificados em C.

Nome: Ana Isabel Torres Garcia Portugal de Almada Cardoso

Doutoramento: em Engenharia Informática e de Computadores

Orientadores: Professores Doutores Rui Gustavo Pereira Nunes Crespo e Tanya Vianna de Araujo

Provas concluídas em:

Resumo

Nesta dissertação defendemos uma forma nova de medir o produto de software com base nas medidas usadas na teoria dos sistemas complexos. Consideramos o uso dessas medidas vantajoso em relação ao uso das medidas tradicionais da engenharia de software.

A inovação desta dissertação sintetiza-se em considerar o produto de software como um sistema complexo, dotado de uma estrutura que comporta vários níveis e na proposta da correlação de gama longa como medida da complexidade de estrutura de programas fontes. Essa medida, invariante para a escala de cada nível da estrutura, pode ser calculada automaticamente.

Na dissertação, primeiro descrevemos o processo de desenvolvimento do software e as medidas existentes para medir o referido processo e produto e introduzimos a teoria dos sistemas complexos. Concluímos que o processo tem características de sistema complexo e propomos que seja medido como tal.

Seguidamente, estudamos a estrutura do produto e a dinâmica do seu. processo de desenvolvimento. Apresentamos um estudo experimental sobre algoritmos codificados em C, que usamos para validar hipóteses sobre a complexidade da estrutura do produto. Propomos a correlação de gama longa como medida da complexidade da estrutura. Estendemos essa medida a uma amostra codificada em Java.

Concluímos, evidenciando as limitações e as potencialidades dessa medida e a sua aplicação em Engenharia de Software.

Palavras-Chave:

Processo de desenvolvimento do software, Produto de software, Métricas de software, Estrutura, Complexidade, Correlação de gama longa.

Title

Structure complexity metrics on algorithms codified in C language

Abstract

In this thesis we advocate an alternative way of controlling and measuring software product and development process.

In software engineering to measure product complexity, we believe that there is an advantage in the use of the metrics used in complex system theory.

We defend that the metric used to measure product structure complexity should be an invariable at all levels of the organisation of the structure of the product - long range correlation has this attribute and the has the advantage of being calculated automatically.

The innovation of this thesis is the use of free scale metrics to measure code source program structure complexity.

We organised this work in the following way:

First we describe the software development process, common metrics used in it and we concentrated on complex system theory.

We conclude that the software process development has the characteristics of complex systems and we therefore propose the use of a complex system theory to measure it.

Then, we focus the study on the product structure in order to understand, his dynamic. The hypotheses on the product complexity structure were validated on a sample of algorithms codified in C and Java languages

We conclude that long range correlation is a valid metrics for our purposes.

Key words

Software Development Process, Software Product, Software Metrics, Structure, Complexity, Long Range Correlation

Agradecimentos

Devo agradecimentos a muitas pessoas:

Ao Professor Doutor Rui Gustavo Crespo do Instituto Superior Técnico, meu orientador que me acompanhou sempre, incentivou nos momentos de desânimo e me avisou dos perigos nos momentos da euforia. Com a sua orientação sempre presente e atenta, aprendi o rigor científico que um trabalho de investigação necessita, a forma como devia prosseguir no estudo e na descoberta, apresentar os resultados e partilhá-los com outros. Muito desta dissertação é resultado das conversas longas que tivemos e das suas reflexões e trabalho que se traduziram em ensinamentos e em abertura de horizontes que me permitiram adquirir e estruturar os conhecimentos em que baseei esta dissertação. Contraí com ele uma grande dívida de gratidão e uma relação de respeito e amizade. Considero que a forma como me acompanhou me obrigará no futuro a tentar pelo menos a fazer igual com quem pretender obter a minha ajuda no desenvolvimento de trabalhos desta índole.

À Professora Doutora Tanya Vianna de Araújo do Instituto Superior de Economia e Gestão, minha co-orientadora por toda a disponibilidade do tempo que me dedicou e pela paciência que teve com a minha ignorância e teimosia. Teve um trabalho enorme, de correcção e modificação da escrita que foi decisivo para a aprovação da dissertação. Igualmente lhe devo os contactos com o seu grupo de investigação que possibilitaram muito do que fiz. Espero que a relação e amizade e de trabalho que estabelecemos continue ao longo das nossas vidas.

Aos Professores Doutores Ludwig Streit, Margarida Faria e Rui Vilela Mendes que estiveram no começo deste percurso fornecendo-me a perspectiva correcta do valor de um trabalho de investigação e o primeiro contacto com a Complexidade, pela amizade que sempre senti que me dedicavam, a confiança de que era capaz de realizar este trabalho que me incutiram e pela disponibilidade e resposta imediata e eficaz que sempre deram aos muitos pedidos de ajuda que formulei ao longo destes cinco anos.

Ao Professor Doutor Peter Kokol da Universidade de Maribor na Eslovénia que me inclui nos seus trabalhos de investigação, comigo partilha contactos e conhecimento e publica artigos.

Aos Professores Doutores Ricardo Lima e António Dente que responderam às minhas dúvidas e forneceram documentação de trabalho.

Aos meus colegas Dr. Leonel Domingos Telo Nóbrega que construiu os programas de filtro da amostra, num período de grande sobrecarga de trabalho durante a preparação das suas provas de aptidão pedagógica, Professoras Doutoras Rita Vasconcelos, Custódia Drumond e Sandra Mendonça que leram e corrigiram a parte estatística, Professor Doutor Amândio de Azevedo que me ensinou a usar o MatLab e apoiou na execução do primeiro programa, ao

Engenheiro Duarte Gomes que me ajudou a transferir e interpretar a mostra que foi objecto de análise nesta dissertação e ao Eng. Dionísio que instalou no meu portátil os programas que eu necessitava e ao meu velho amigo Eng. Alvaro Athayde que teve a paciência de me ajudar quando a concentração não era grande, e trabalho tinha de ser feito.

Aos meus colegas da Secção que foram sobrecarregados com o meu trabalho, durante o tempo de dispensa que gozei e que sempre me fizeram sentir que era mais importante o meu doutoramento do que o trabalho a mais que tinham de fazer. Principalmente ao Eng. Alberto Velez Grilo e Professor Doutor José do Carmo directores, durante a execução deste trabalho, do departamento a que pertenço na UMA, pela forma como sempre me apoiaram no meu trabalho e facilitaram a obtenção das condições de trabalho que necessitei e ao Sr. Nelson de Vasconcelos que me evitou grande parte do trabalho administrativo que me competia fazer e ajudou na impressão e revisão desta dissertação.

Aos funcionários da DTIM, meus colaboradores que me protegeram das interrupções que o meu dia a dia de trabalho na direcção de uma Associação causa. Aos Eng. Xavier Nunes e Paulo Nunes que me apoiaram na obtenção de alguns programas da amostra que analisei, à formadora Elizabete França que resolveu muitos dos problemas de formatação desta dissertação, à minha secretária Graciete Roseira que dactilografou as tabelas e à Drª Ana Cristina Viveiros e à Drª Teresa Leça que releram o texto.

Aos meus amigos que me fizeram sentir como era importante o esforço que estava a realizar nomeadamente o Drº Mário Passos Freitas que durante as consultas e tratamentos além de me ter mantido viva, tantas vezes teve tempo e paciência para discutir aspectos deste estudo, as Drªs Maria do Rosário Baptista, Rita Andrade e Lígia Figueira que leram, comentaram e corrigiram o estilo de escrita e me distraíram, divertiram e mantiveram a moral em alta.

Ao meu irmão António e à minha amiga Rosemary Blandy e ao meu sobrinho Andrew Zino que me ajudaram na escrita em inglês dos artigos que publiquei.

Ao meu marido António, às minhas filhas Margarida, Joana e aos meus filhos António e Manuel a quem roubei espaço e tempo.

À Universidade da Madeira pelo apoio logístico prestado nomeadamente na obtenção da bolsa de PRODEP. Ao Centro de Ciências Matemáticas da Universidade da Madeira pelo apoio prestado nas deslocações para apresentação de trabalhos.

1. Introdução

...Porque a Ciência avança inexoravelmente, nós aprendemos mais acerca dos problemas, e quanto mais aprendemos mais somos capazes de nos preocupar com o grau de complexidade que existe...
A.B. Çambel 1998

1.1. Resumo

Neste capítulo indicamos brevemente as motivações que nos conduziram à escrita desta dissertação, o enquadramento do problema que pretendemos desenvolver e as contribuições que propomos para a sua resolução. Apresentamos ainda a organização desta dissertação através de uma descrição sumária do conteúdo de cada capítulo.

1.2. Motivação

O controlo do desenvolvimento do projecto de software é um dos assuntos críticos da Engenharia do software [Naur,P.69]. Este problema foi largamente estudado e alvo de publicações científicas e comerciais. Os utilizadores conhecem e reportam com frequência custos adicionais aos seus orçamentos anuais de investimento e de pessoal, causados por entregas fora de prazo de sistemas e programas defeituosos. As empresas fornecedoras frequentemente vêem diminuídos os seus lucros ou até os projectos resultam em percas, porque, na impossibilidade de as empresas controlarem o processo de desenvolvimento do software, falham prazos de entrega e previsões de custo de sistemas e programas. Os cientistas e as Universidades são solicitados pela indústria a desenvolver investigação nesta área com o objectivo de desenvolver teorias, metodologias e técnicas que permitam aumentar a produtividade do processo.

A experiência pessoal, nos últimos vinte anos, como técnica de uma empresa de grande dimensão na área de comunicações, com responsabilidade na gestão de projectos baseados em computadores, forneceu a visão da dificuldade diária que o controlo do desenvolvimento do

projecto de software apresenta para o utilizador. Cerca de 90% do esforço de gestão e controlo de projectos de software é gasto em controlar a complexidade do produto ao longo do processo da sua evolução e a obter previsões precisas do tempo de desenvolvimento ou de manutenção e do custo real dos produtos.

Os estudos da disciplina de Engenharia de Software, especialmente dos seus ramos Processos de Produção de Software e Métricas de Software, sustentam teoricamente a forma de organizar e controlar o conjunto sucessivo das tarefas que o processo de desenvolvimento comporta.
O ramo das Métricas de software fornece a fundamentação para a construção de medidas do software e do seu processo de desenvolvimento. Como tal, permite melhorar o conhecimento que podemos ter do processo de desenvolvimento e dos seus resultados: os programas e as suas sucessivas versões.

Embora o estudo do processo de desenvolvimento do software, da sua medição e do seu controlo, tenha já sido objecto de estudos conjuntos e aprofundados de cientistas, fornecedores e clientes, a indústria necessita de um grau de precisão de estimativas (de custos e tempos de desenvolvimento, implementação e evolução) muito mais exigente do que actualmente é conseguido.

A teoria dos sistemas complexos estuda sistemas que, embora sejam determinísticos, manifestam comportamentos aparentemente aleatórios [Morowitz,H.88;Cohen,B.86;Gel-Mann,M.95;ZIF,00]. Não existe uma definição de sistema complexo aceite por toda a comunidade científica, mas nesta dissertação adoptamos a seguinte definição: um sistema complexo é um sistema formado por um elevado número de elementos, caracterizado por uma dinâmica não-linear, por interdependência entre o comportamento dos seus elementos e pelo aparecimento de propriedades colectivas, qualitativamente diferentes do comportamento individual.
A essência dos sistemas complexos reside na natureza e no efeito das interacções entre os seus elementos. A complexidade que lhes é característica decorre da dificuldade de obter, reunir e organizar informação suficiente para a sua representação.

M.M. Lehman e L.A.Belamy [Lehman,M.85], formularam leis qualitativas aplicáveis à generalidade dos sistemas de software, que podem sustentar a hipótese do seu determinismo (ver secção 1.4).
Os sistemas de software podem, de acordo com definição formulada no parágrafo anterior, ser considerados sistemas complexos porque:

- Consistem num grande número de entidades, tais como sub-rotinas e objectos que interagem entre si.
- Os atributos e comportamentos dessas entidades são reconhecidamente diferentes dos atributos globais do programa, embora se possa dizer que o comportamento do programa seja uma propriedade emergente da interacção entre os seus objectos constituintes.
- O seu processo de desenvolvimento é difícil de controlar e pode conduzir a tipos diferentes de soluções.
- Evoluem ao longo do tempo apresentando vários estados, os quais podem ser representados pelas versões dos programas disponíveis num determinado momento.

Consideramos nesta dissertação que os programas são sistemas complexos e que, por isso, o seu processo de desenvolvimento é caracterizado por uma dinâmica complexa.

As ideias de estrutura e organização são intuitivas e frequentemente usadas nas descrições de objectos. Informalmente, designamos a organização de um objecto como a disposição das suas partes e a estrutura como a descrição dessa organização.

1.3. A complexidade

Em geral um objecto é considerado complexo quando, por variadas razões, não somos capazes de compreender como actua e/ou de prever o seu comportamento.
O tamanho, a dificuldade de entendimento, a descrição mínima e a variedade contida num sistema, são apresentados como indicadores de complexidade de um sistema [Edmonds,B.00].
A razão de se considerar o tamanho como indicação da existência de complexidade é a intuição de que um sistema de grandes dimensões é potencialmente mais elaborado. Este raciocínio poderá ter alguma validade se estivermos a comparar entidades do mesmo tipo, cujo crescimento seja feito por criação de outros níveis de organização mas, mesmo nesse

caso, está-se a confundir tamanho com emergência de estrutura. Nada nos permite considerar que um conjunto grande de elementos não inter-relacionados seja (intuitivamente) mais complexo que um conjunto pequeno de elementos com grande número de relações entre eles. São exemplos as três situações seguintes: (i) um longo programa cuja maioria de linhas de código sejam comentários ou um programa escrito com recurso a chamadas a uma biblioteca de sub-rotinas, constituído por algumas chamadas estruturadas de sub-rotinas; (ii) o genoma humano mais pequeno mas muito mais difícil de decifrar e o genoma do trigo dez vezes maior que o humano mas construído por sequências de elementos idênticos; (iii) um contentor de maçãs e um ferro eléctrico. Nestes três exemplos, o tamanho por si só não indica complexidade.

É, também, intuitivo pensar-se que uma indicação da complexidade seja dada pela dificuldade de se entender a entidade. No entanto, esta avaliação da complexidade é subjectiva, visto incluir o grau de ignorância do observador sobre o assunto em análise. Veja-se o exemplo da electrónica do aparelho de vídeo, complexa para um programador de *visual basic*, [Campos,L.99] visto que não detém conhecimentos básicos que permitem compreender o seu funcionamento. Pelo contrário, os programadores consideram pouco complexo programar em *visual basic*, porque detêm o domínio experimentado dessa linguagem. A visão do técnico que repara o aparelho de vídeo é oposta à do programador: para o técnico, programar em *visual basic* é uma tarefa complexa e reparar aparelhos de vídeo é uma tarefa simples.
A complexidade pode, portanto, ser causa de dificuldade de entender um assunto mas a asserção inversa também é verdadeira. A dificuldade de entendimento, por si só, também não indica complexidade, tem pelo menos de ser eliminada a influência da ignorância do observador para poder ser utilizada, com alguma validade, como critério de avaliação da complexidade.

Analisemos então o uso do tamanho da descrição mínima que se pode fazer de um sistema como indicador da sua complexidade. Também aqui a complexidade fica dependente da linguagem usada para a descrição e, como tal, é de avaliação subjectiva. Por outro lado, pode-se provar matematicamente que uma linguagem permite a descrição mínima do sistema. No entanto, a linguagem seleccionada pode ser ilegível para os humanos, sendo assim gerados os problemas descritos no indicador baseado na dificuldade de entendimento.

Finalmente, o conceito de variedade está também ligado à complexidade. Um sistema complexo inclui necessariamente variedade, mas a afirmação contrária não é verdadeira. É exemplo desta situação um corpo humano que é complexo porque é formado por um conjunto variado de órgãos diferentes. No entanto, um quadro pode ser pintado com várias centenas de tons de cores diferentes e ser muito fácil, i.e., pouco complexo, de entender e de sentir.

A complexidade está ainda dependente do nível de detalhe da descrição da entidade. É exemplo desta situação o facto de uma sub-rotina poder ser considerada simples para um utilizador que pretende simplesmente usá-la. O referido utilizador terá unicamente de saber quais as variáveis que são incluídas na chamada e qual é o resultado dessa chamada durante o programa. Mas, a sub-rotina pode ser considerada muito complexa para o programador que a implementa e que terá necessariamente de compreender o procedimento que transforma as variáveis em solução.

Podemos concluir pelo que acima descrevemos, que a eficácia do contributo das qualidades referidas para a definição de complexidade de uma entidade (tamanho, dificuldade de compreensão, descrição mínima e variedade) se apresenta dependente do contexto e subjectiva em relação ao avaliador e à escala.

Actualmente, o conceito de complexidade refere o comprimento da descrição do conjunto, quer das suas regularidades (padrões) quer das suas irregularidades [GelMann,M.95]. A complexidade é uma propriedade de uma determinada descrição do sistema feita a um determinado nível de abstracção.
No mundo real, observa-se que uma qualquer sequência de dados expressos numa dada linguagem (binária, código ou descrição textual), se for aleatória, tem uma complexidade próxima do zero. Não existe nela nenhuma regularidade a observar.
Do mesmo modo, uma sequência formada só por zeros também não é complexa, porque tem estrutura mais simples: a repetição de *n* símbolos iguais. A complexidade é, portanto, qualquer organização entre a desordem total e a ordem total. Embora não exista nenhum processo para encontrar o conjunto de regularidades de uma entidade, é possível não só identificar classes de regularidades em entidades como verificar a sua persistência ao longo da estrutura da entidade. Esta dissertação debruça-se sobre a complexidade como um dos

aspectos centrais do processo de desenvolvimento do software.

Por fim consideramos que a complexidade de programas fonte, está associada a duas estruturas: a sintáctica e a funcional.
A estrutura sintáctica (da linguagem de programação) é a organização dos elementos do vocabulário, definida pela gramática da linguagem. Um algoritmo codificado numa linguagem obrigatoriamente respeita a estrutura sintáctica da linguagem na qual o algoritmo é expresso. Nesse caso, cada uma das instruções que o constituem estão construídas de acordo com as regras da gramática e encontram-se agrupadas (organizadas) de acordo com estruturas permitidas pela gramática.
A estrutura funcional do programa está relacionada com a quantidade e com a diversidade de funções e estruturas de dados que o programa processa a fim de cumprir um determinado objectivo.
Estas duas estruturas, embora diferentes, encontram-se contidas em qualquer algoritmo codificado que execute pelo menos uma função. Sendo compilavel, o algoritmo obedece à gramática da linguagem e, executando uma (ou mais) funções, tem uma funcionalidade.

1.4. A evolução e a complexidade do software

A evolução de software tem sido analisada por diversos estudiosos [Humphrey,W.01; Ahern,d.01;Yang,H.03;Pressman,R.94;Sommerville,I.96;Pfleeger,S.98].
Many Lehman [Lehman,M.85] propôs em 1980 leis sobre a evolução de software:
A primeira lei - lei da mudança contínua - diz-nos que um programa em uso ou incorpora continuamente mudanças ou torna-se continuamente menos útil (envelhece). A mudança e o processo de envelhecimento continuam, até ser mais económico substituir o sistema do que refazer uma nova versão.
A segunda lei - lei do aumento da complexidade - diz-nos que, como um programa em evolução está continuamente a ser modificado e a sua organização deteriora-se. Como reflexo disso a sua complexidade aumenta, a não ser que seja despendido esforço em a manter estável ou em reduzi-la.

Estas duas leis indicam que, durante o ciclo de vida do programa, a complexidade das várias versões têm de ser controlada. Se não o for, a dinâmica do processo acaba por produzir

versões que deixam de ser solução para as necessidades que o software pretende satisfazer. Estas leis são descritas informalmente. Não são conhecidas ferramentas de controlo da evolução do software que atinjam o nível de precisão exigido pela indústria.
As medidas tradicionais da complexidade do software, usadas no âmbito da engenharia de software, não permitem a análise global da complexidade do mesmo.
Consideramos que as medidas de complexidade empregues na caracterização de séries temporais (as quais, em geral, representam sistemas dinâmicos não lineares), consistem numa hipótese promissora para a formulação matemática rigorosa de leis de controlo do processo de desnvolvimento do software.
Pretendemos medir a complexidade do produto (software), que Lehman enunciou existir e ser necessário controlar (ao longo do processo de desenvolvimento), usando, como principal instrumento, o conceito de Correlação de Gama Longa (*"Longe Range Correlation")*.

1.5. Objectivos da dissertação

Esta dissertação tem início com a análise da forma como decorre o processo de desenvolvimento do software, quais as suas características e dificuldades em ser controlado. A motivação desta dissertação está em parte associada à necessidade de entender porque é que as previsões de esforço e calendário essenciais no controlo do processo de desenvolvimento do software, embora tenham vindo a aumentar a sua precisão, não conseguem atingir o nível exigido pela indústria. Propomos para a resolução deste problema, uma caracterização do produto de software enquanto um sistema complexo.
Sendo, o software, um sistema complexo, a sua sensibilidade à mudança de condições iniciais implica que a capacidade de previsão do seu comportamento seja limitada no tempo [Garnett,P.97]. Atendendo a esse facto, consideramos que a ênfase deve passar da previsão para o controlo do processo de desenvolvimento.

A contribuição inovadora desta dissertação consiste:

1. na caracterização de programas codificados em C como sistemas complexos. Verificamos, experimentalmente, a adequação de algumas medidas da complexidade e concentramos o nosso esforço no estudo das condições em que o uso da medida de correlação de gama longa pode caracterizar a complexidade da estrutura sintáctica e funcional do referido produto. Alargámos o âmbito de aplicação desta métrica á

linguagem Java.

2. na proposta da equação logística como modelo da dinâmica do processo de desenvolvimento do software e a utilização da correlação de gama longa como métrica de estrutura dos programas..

   Esta métrica é calculada automaticamente, para cada versão, e possibilita, através de da medida de uma série de versões, uma indicação do progresso ou retrocesso da complexidade dos programas e seu controlo.

1.6. Descrição da dissertação

Esta dissertação está organizada em oito capítulos. Cada capítulo, com a excepção do capítulo de introdução e do capítulo de conclusões, é iniciado por uma introdução e finaliza com um resumo do capítulo e, quando é o caso, com um resumo da contribuição de inovação que a sua escrita acrescentou nesta dissertação. Dentro de cada capítulo as tabelas, as figuras, as definições e os exemplos são numerados, usando o número da secção a que pertencem seguido de um número sequencial correspondente à sua ordem de aparecimento na secção. Os anexos são colocados no final e obedecem à mesma lógica de numeração. Utilizamos o símbolo "⊗" para indicar a terminação da definição e do exemplo.

O capítulo primeiro consiste numa introdução onde se descreve com brevidade o contexto do problema e as principais contribuições que se propõem como resolução.
Seguem-se os capítulos dois e três onde é feito o enquadramento teórico do problema.
No capítulo segundo começamos por definir produto (de software) e processo (de desenvolvimento do software). Descrevemos o referido processo indicando como vai dando origem, durante o desenvolvimento, a uma sequência de modelos da solução a níveis de abstracção diferentes e com fins diversos. Damos especial importância à descrição do processo ideal e às causas da evolução do produto. Argumentamos com as diferentes escalas temporais dos vários tipos de re-alimentação de informação ao longo do processo para demonstrar que um processo ideal (ortogonal e não incremental) não é realizável.
Descrevemos, seguidamente, de forma breve, os dois paradigmas utilizados para o processo real, analisando as suas limitações.
O capítulo termina com a hipótese de investigação de que o produto gerado no processo é complexo, no sentido da teoria dos sistemas complexos, e que a sua evolução deve ser

modelada com base na mesma teoria.

No capítulo terceiro abordamos a necessidade do uso das métricas em engenharia do software, e a forma como se constrói uma métrica. Descrevemos brevemente métricas do processo e métricas do produto. Focamos especialmente, no caso do produto, as várias métricas da complexidade mostrando as dificuldades que apresentam quando são usadas para controlar o processo.

Concluímos que outros tipos de métricas da complexidade - as métricas usadas na teoria dos sistemas complexos - podem ser usadas com vantagem no controlo do processo.

No capítulo quarto apresentamos as propriedades fundamentais dos sistemas complexos, algumas medidas da complexidade ou a esta associadas e procuramos desta forma estabelecer as condições necessárias à apresentação das hipóteses de investigação formuladas nesta dissertação.

No capítulo quinto descrevemos a amostra (o conjunto dos dados) sobre a qual trabalhamos, os 154 programas utilizados no estudo experimental que efectuamos, apresentamos e analisamos criticamente os resultados obtidos na experimentação.

Concluímos com a discussão das primeiras hipóteses formuladas nesta dissertação.

No capítulo sexto utilizamos as medidas da estrutura que usamos na parte experimental desta dissertação e que apresentámos no capítulo cinco, para evidenciar como a amostra dos programas com a mesma funcionalidade se segmenta em grupos internamente homogéneos e diferentes entre si. Para tal, primeiro efectuamos uma análise taxinómica e evidenciamos que a amostra é homogénea. Depois, usamos uma técnica de identificação de intrusão de vírus para caracterizar cada compilador. Construímos o chamado *self* de cada um. Verificamos que os "*selfs*" dos dois grupos mais afastados em termos de funcionalidade dentro da amostra não se podem considerar diferentes. As conclusões do capítulo reforçam as ilações do capítulo cinco.

No capítulo sétimo apresentamos uma aplicação do cálculo da correlação de gama longa para controlo do processo desenvolvimento de software.

Para tal, apresentamos brevemente os conceitos que caracterizam a dinâmica dos sistemas complexos e analisamos a equação logística, que modela a evolução de populações competindo por recursos limitados.

De seguida, aplicamos esta equação como modelo da dinâmica de um processo de desenvolvimento de software real e exemplificamos como é possível determinar o estado do

processo, calculando a correlação de gama longa para a sucessão de versões obtidas.
Finalmente apresentamos resultados obtidos sobre uma amostra de cerca de 43 programas e suas versões, onde se evidencia a utilidade do modelo referido para controlar o processo de desenvolvimento de software.
No capítulo oitavo descrevemos as conclusões globais da dissertação e o trabalho futuro a realizar no âmbito desta linha de investigação.

2. O Processo de software

...O bom funcionamento dos Sistemas políticos e económicos modernos depende da nossa habilidade para produzir software a custos controlados...
Ian Sommerville 1990

2.1. Introdução

Neste capítulo é apresentada uma análise dos processos de software em termos da sua organização intrínseca com vista à evolução do produto. Evita-se a mera descrição dos diferentes modelos adoptados pelas equipas projectistas, por se encontrarem amplamente divulgados em textos didácticos de disciplinas de engenharia de software [Pressman,R.94; Sommerville,I.96;Pfleeger,S.98].
Descrevemos os dois paradigmas utilizados para modelar o processo, analisando as suas limitações.
Fazemos uma breve referência às formas tradicionais do controlo do processo como forma de enquadramento comum da construção das métricas que propomos nos capítulos 4 e 5. Por fim, apresentamos a nossa justificação de que o processo obedece a leis gerais e listamos a nossa contribuição de que é possível formular um modelo matemático para essas leis.

2.2. Definições

No desenvolvimento da aplicação informática, o engenheiro de software modela parte do mundo real - o problema proposto - transformando-o em software: a solução [Lehman,M.85].

Definição 2.2.1.: O processo de desenvolvimento de software é a sequência de actividades que transformam um problema do mundo real numa solução informática. ⊗

A partir de agora, o processo de desenvolvimento do software é mencionado, de forma abreviada, como processo.

O processo envolve a escolha das actividades e sua sequência, a escolha de estratégias, metodologias e ferramentas, a avaliação de recursos e limitações necessários à obtenção da solução.
O processo, quando composto de subprocessos, constitui-se numa hierarquia de processos. Desde o início do processo, é prática corrente que as suas tarefas sejam executadas por equipas mistas de utilizadores e fornecedores: nesse contexto, a identificação do problema pode ser incluída no processo. Nesta dissertação não a incluímos por considerarmos que a identificação do problema é uma tarefa maioritariamente da responsabilidade do utilizador e, como tal, anterior e externa ao processo.

Definição 2.2.2.: Designa-se por produto de software, a partir de agora, simplesmente por produto, o resultado do processo, em qualquer das suas fases. ⊗

A designação de produto inclui:

- Programas que mapeiam os requisitos do problema em código executável,
- Estruturas dos dados, necessárias à execução dos programas,
- Documentação que compreende os sucessivos modelos da solução, as suas descrições, testes e simulações e guia o utilizador na instalação e uso do sistema e o engenheiro de software no controlo do processo e manutenção do produto.

As actividades de controlo e de manutenção permitem que o produto seja entregue obedecendo aos requisitos do problema e que, depois de entregue, evolua.

Definição 2.2.3.: Os ambientes de desenvolvimento do software são conjuntos de ferramentas, total ou parcialmente integradas, juntamente com a política da sua utilização, que apoiam a realização das tarefas do processo. ⊗

Os ambientes de desenvolvimento são mencionados neste estudo simplesmente como ambientes.
Ressalve-se que é impossível a automatização total do processo, atendendo a que se trata de uma actividade onde a criatividade é essencial.

A definição de um processo estrutura e confere consistência a um conjunto de tarefas diferentes definindo a sua dependência e sequência. Quando o processo é executado sem iteração, ou seja, sem repetição da mesmas fases, para redefinição ou acrescento de funcionalidade, é mais fácil controlar os produtos gerados de forma a assegurar o padrão de qualidade escolhido.
Consideramos que o modelo do processo e a forma como é gerido, influenciam a qualidade, o custo e o tempo de obtenção do produto [Dromey,R.96;Pfleeger,S.98a]. Logo, o processo de software é abordado com detalhe neste capítulo.

2.3. Produtos de software

Várias têm sido as propostas de classificação de produtos. Nesta secção referimos a divisão quanto à amplitude da aplicação: genéricos e feitos por medida [Pressman,R.94; Sommerville,I.96;Pfleeger,S.98]. A diferença entre os dois tipos consiste em que os primeiros são desenvolvidos para resolver um problema comum das organizações e vendidos a qualquer uma delas que seja capaz de os comprar e os queira utilizar; os segundos são desenvolvidos por encomenda de uma determinada organização para resolver um problema específico.
A avaliação do produto faz-se através da análise e medição dos seus atributos. Vários autores descrevem o conjunto de atributos que um software bem desenvolvido deve possuir [Abreu,F.94;Crespo,R.93;Pressman,R.94;Sommerville,I.96;Pfleeger,S.98].
De entre os múltiplos atributos do produto, descritos pelos autores referidos, seleccionamos como indispensáveis à classificação de um software bem desenvolvido os seguintes:

- Facilidade de manutenção - capacidade de o produto poder ser modificado, em resposta à necessidade de mudança do utilizador, por que garante maior tempo de validade para o produto.
- Eficiência - relação entre o nível de desempenho do produto e a quantidade de recursos utilizada, porque garante maior economia no desenvolvimento.
- Usabilidade - medida do esforço necessário para utilizar o produto, porque facilita a sua instalação e uso.
- Confiabilidade - capacidade de não causar estragos físicos ou económicos quando o produto entra em falha, porque diminui os riscos inerentes à utilização.

As quatro características descritas são, todavia, contraditórias e não podem ser maximizadas independentemente. Por exemplo, a usabilidade alta implica o uso de interfaces sofisticadas com o utilizador, o que reduz a eficiência. A sua maximização deve ser feita, atendendo ao uso específico do produto.

As várias possibilidades de validação de um produto dependem do seu tipo. Segundo Lehman há três tipos de produtos [Lehman,M.85]:

- Produtos S (“s*pecification”)* que representam a solução de um problema concreto e bem definido, independentemente do uso no mundo real que dele se possa fazer.
  Estes produtos são correctos se estiverem de acordo com a especificação que foi feita. A prova matemática da sua correcção é possível, visto que é derivável da especificação feita.

- Produtos P (“*problem”)* que resolvem problemas concretos do mundo real, mas cuja solução é construída através de uma abstracção do problema, incompleta, porque a dimensão do problema ou a sua complexidade não permitem a resolução de outra forma.
  A validação **da solução** não é feita por comparação com a especificação, uma vez que se considera que esta pode ser incompleta. Avalia-se o produto comparando o seu desempenho no mundo real, com o desempenho esperado para a solução desenhada. O critério de aceitação do produto é operativo. A solução, contrariamente à dos produtos S, não tem de estar de acordo com a especificação, mas ser uma das muitas soluções possíveis, que dê provas, durante a implementação, que é aceitável, porque funciona.

- Produtos E (“e*mbedded”*) em que o próprio produto faz parte do mundo real. A validação da solução é medida unicamente pela satisfação que ela traz ao utilizador. A especificação inicial só é relevante para o primeiro ciclo de desenvolvimento. Ela própria é afectada pela mudança e, como tal, vai sendo alterada ao longo do processo.

Exemplo 2.3.1.: Como exemplos de produtos S tem-se o cálculo de uma função num determinado domínio, a inversão de uma matriz.
Como exemplos de produtos P tem-se o problema do caminho óptimo da distribuição de um

bem ou serviço. Durante a implementação da solução, as condições iniciais do problema são modificadas pela solução. Uma urgência, surgida por um facto novo, pode obrigar a que o caminho óptimo (o melhor para a satisfação global dos clientes) seja mudado quando já se visitaram clientes.
Como exemplos de produtos E tem-se a criação dos horários de uma escola. Frequentemente, é durante essa organização que os responsáveis verificam que necessitam ou de mais salas, ou de mais professores, ou que há necessidade de modificar os turnos de funcionamento da escola. Durante a implementação e execução da solução, ela próprio modifica o mundo real**.** ⊗

Nesta dissertação abordamos só tipos de produto P e E, visto considerarmos que os produtos de tipo S não apresentam o grau de complexidade que conduz à necessidade de tratamento usando a analogia com os sistemas complexos.

2.4. Causas da evolução do produto

As causas conducentes à modificação do produto são-lhe intrínsecas e dependentes do processo. Sendo intrínsecas ao produto obrigam a que o processo inclua uma fase de evolução. As causas actuam durante o primeiro ciclo do processo e durante todo o ciclo de vida do produto.
As pressões para evolução têm sido descritas informalmente, salientando-se [Lehman,M.85]:

- Há pressões para a evolução feitas pelo aparecimento de novos ambientes de desenvolvimento ou novas plataformas tecnológicas. As modificações com esta origem, geralmente, tem intervalos quantificáveis em anos e são causadas pelo processo ser desenvolvido numa escala temporal diversa da escala da ocorrência da mudança tecnológica. Esta causas são intrínsecas ao produto.
  Exemplo 2.4.1.: São exemplos de mudanças, causadas por pressões deste tipo, a mudança, para o sistema operativo Windows NT, dos programas de imageologia radiológica, inicialmente desenvolvidos para um sistema operativo proprietário. ⊗

- Há pressão dos utilizadores que provêm da troca contínua de informação, entre os técnicos que construíram o produto e os clientes que o utilizam. Essa troca de informação conduz os aperfeiçoamentos, correcções a adições ao produto. As modificações com esta

origem geralmente têm intervalos quantificáveis em meses e são vulgarmente conhecidos por versões (“releases”). O aparecimento de novas versões é dependente do processo, que pode incluir a definição da metodologia para o seu desenvolvimento, e intrínseco ao produto visto ser causadas pelas necessidades de novas funcionalidades e adaptação a novos usos do produto.

Exemplo 2.4.2.: São exemplos de mudanças, causadas por pressões deste tipo, a construção de uma versão para a correcção de erros detectados durante um mês de utilização e para efectuar a modificação causada por mudanças do enquadramento legal do problema surgida durante esse intervalo de tempo. ⊗

- Há pressões para a mudança que se verificam durante a construção da primeira versão do produto, quando utilizadores e técnicos validam em conjunto as várias decisões do processo. As modificações, com esta origem, têm periodicidade mensurável em dias. São pressões intrínsecas do produto, uma vez que é a natureza do produto que faz que quer a sua validação, quer a sua implementação, vão modificando o ambiente onde opera e, como tal, induzam mudanças na própria definição do problema.

  Exemplo 2.4.3.: São exemplos de mudanças, causadas por pressões deste tipo, as modificações incorporadas por sugestão do cliente, enquanto este valida um protótipo da construção do interface com o utilizador. ⊗

Em resposta a estas pressões surgem acções, de sinal contrário, da parte das organizações que desenvolvem e usam os produtos, no sentido de conseguirem algum tempo para se familiarizarem com o que estão a construir ou a usar e equilibrarem a quantidade de esforço que lhes é exigida para acompanharem a evolução.

As causas listadas levam o processo a ser um sistema realimentado com pressões positivas para a mudança e pressões negativas para a conservação [Lehman,M.85]. Nestas condições, o produto evolui e o processo é iterativo e as modificações operam-se em diferentes escalas temporais ao longo da vida do produto. Podemos concluir que o produto sofre a necessidade de ser adaptado constantemente e o processo tem uma dinâmica que produz essa adaptabilidade.

Durante a construção da primeira versão do produto as fases de desenvolvimento de definição e de implementação alternam com a fase de evolução devido às modificações do ambiente do

produto e clarificação e mudança dos seus objectivos. Quando produto é entregue ao cliente, no fim do seu primeiro ciclo de vida, a fase de evolução determina quando é necessário definir e implementar novas funcionalidades do produto, iniciando um novo ciclo de vida do produto, que constrói uma nova versão do produto. A criação de novas versões repetem-se até que seja mais económico construir um novo produto do que adaptar a versão do produto em funcionamento.

2.5. O processo de software

O processo influencia os atributos do produto. Como tal, os atributos do processo e a decisão de escolha do tipo de processo a utilizar deve ser realizada na sequência da análise dos atributos que pretendemos para o produto [Abreu,F.94;Crespo,R.93;Pressman,R.94; Sommerville,I.96;Pfleeger,S.98].

Entre as diversas tarefas dos processos de software, destacamos:

- Definição de requisitos, que consiste no estabelecimento das limitações e da funcionalidade do sistema.
- Arquitectura, que consiste no tradução dos requisitos num modelo de componentes de software, respectivas estruturas de dados, procedimentos e caracterização de interligações.
- Implementação, que consiste na codificação das funções.
- Teste, que consiste em assegurar que cada componente desempenha a função para que foi criado e que em conjunto implementam os requisitos do problema.
- Manutenção (i.e., correcção, adaptação e aperfeiçoamentos no produto instalado), que consiste em manter o produto operacional e satisfatório ao longo do tempo.

Há várias sequências possíveis para estas tarefas.

Exemplo 2.5.1.: Nos modelos de cascata [Royce,W.87] as tarefas são executadas sequencialmente. No modelo espiral [Bohem,B.88] a seleção das tarefas depende do risco e a norma de qualidade ISO9000-3 menciona que as tarefas devem ser realizadas não impondo, no entanto, nenhuma ordem. ⊗

Com base na listagem das tarefas dos processos de software, as fases da construção de um

produto serão pelo menos três:

- definição, onde são estabelecidas as funcionalidades e as limitações requeridas ao funcionamento.
- desenvolvimento, onde é construído o software, de acordo com a definição, e se avalia se a solução encontrada funciona no mundo real de acordo com aquilo que o utilizador pretende.
- evolução, onde se vai modificando o software à medida que as necessidades dos utilizadores o requerem.

A última fase - a evolução do produto - é aquela que corresponde à maior percentagem do custo do produto. Embora seja variável de projecto para projecto, estudos indicam que corresponde em média de 60% a 70% do custo total [Dromey,R.96;Pressman,R.94; Sommerville,I.96;Pfleeger,S.98]. Sendo a evolução a tarefa de maior custo, é essencial que o processo seja, desde o seu início, orientado no sentido de facilitar a evolução/manutenção do produto.
De referir a existência de produtos, para os quais as preocupações de desenvolvimento se sobrepõem às da evolução, como, por exemplo, a escrita de produtos para a Web. Estes produtos estão em constante modificação para fidelizar a atenção do utilizador. Como tal, não é rendível apostar na sua manutenção, mas prefere-se ir produzindo novos programas de custos de concepção e implementação reduzidos. Esta classe de produtos sai fora do âmbito do trabalho desta dissertação.

2.5.1. O processo ideal

Idealmente, o processo para desenvolvimento do produto seria contínuo, concorrente e não iterativo, ou seja evoluiria ortogonalmente para uma solução. O processo ideal realizar-se-ia numa sequência de subprocessos ortogonais e de um conjunto de sub-transformações que levariam da proposta inicial do problema ao produto final, que uma vez em funcionamento não necessitaria nunca de ser modificado.

Na figura 2.5.1.1. representamos em diagrama os dois primeiros níveis de modelos criados no processo ideal. Os níveis representam graus de abstracção do processo ideal.
No primeiro nível, o de maior abstracção, o processo inicia-se com a criação dos modelos da

proposta do problema, a que se segue o modelo da identificação e o modelo da estruturação da solução, finalizando com o modelo operacional da implementação da solução. Esta é a sequência descrita na primeira coluna da figura 2.5.1.1..

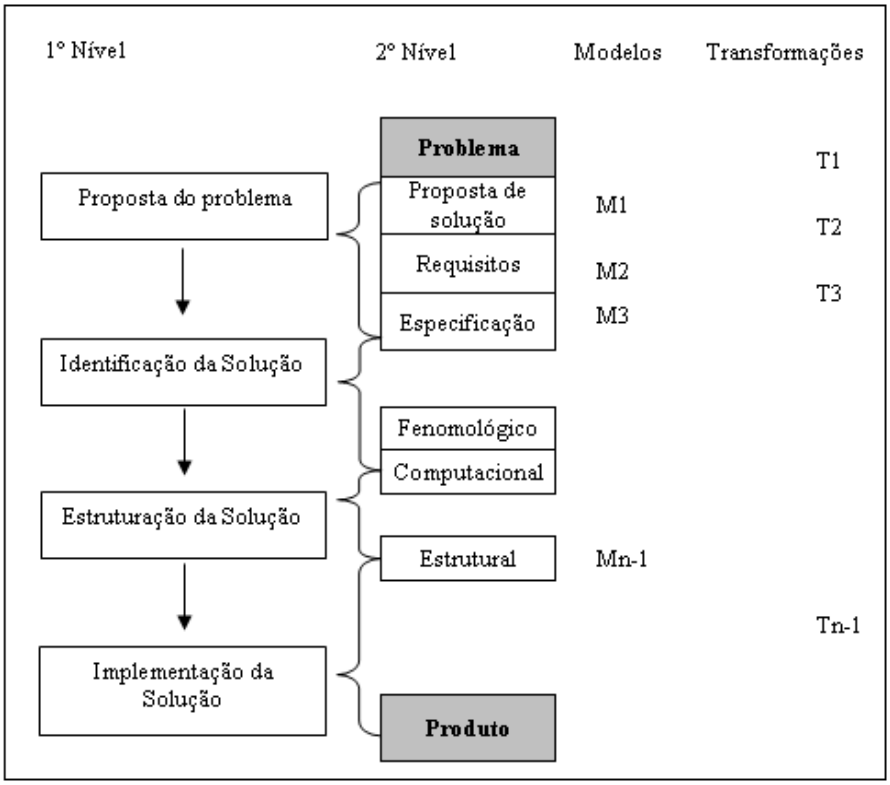


Figura 2.5.1.1. O Processo Ideal - Adaptado de [Leahman, M. 85]

No segundo nível, indicado na coluna 2 da figura, estão pormenorizados os sub-modelos que constituem os modelos descritos na coluna 1 da figura 2.5.1.1. As chavetas indicam a sua inclusão nos modelos do primeiro nível.

Cada modelo, numerado na coluna três da figura, é obtido do modelo anterior, a partir de uma transformação indicada na coluna quatro da figura.

Exemplo 2.5.1.1.: É exemplo da sequência da figura 2.5.1.1. a construção para uma empresa de um programa de gestão de viaturas. A proposta do problema, o primeiro modelo indicado, consiste na comunicação do cliente para o fornecedor do pedido de um programa de gestão de viaturas incluindo um primeiro entendimento do domínio do problema e das suas condicionantes.

A seguir, o analista (o fornecedor) constrói a sua perspectiva sobre o domínio do problema que lhe foi proposto e obtém consenso sobre a sua perspectiva e aquela que o cliente tem sobre o problema. Esta harmonização origina uma identificação de solução, que é um modelo que inclui representação das entidades, relações, estrutura, acontecimento e processos importantes para a solução. O analista necessita adicionalmente de descrever o domínio do problema e as regras que vigoram nesse domínio.
Com este conhecimento, é possível estruturar a solução, criando um modelo do sistema a construir. Esse modelo inclui as entidades que o formam e para cada entidade do modelo são estabelecidas as interligações, os dados que processa, os dados que produz. Finalmente, a solução é implementada com base nessa arquitectura. ⊗

Cada um destes modelos é formado por submodelos, que pormenorizamos na coluna do segundo nível da figura 2.5.1.1., e que descrevemos a seguir.
A proposta do problema inclui a construção dos seguintes modelos do produto:

- Um modelo de um primeiro consenso entre o pedido do cliente e a interpretação que o fornecedor faz do pedido - a proposta da solução.
- Um modelo de requisitos onde está definida e estruturada a totalidade dos objectivos do produto, em termos do que ele deve fazer, excluindo como o vai fazer. Esta descrição é feita em termos do domínio do discurso.
- Um modelo de especificação onde os requisitos são expressos como componentes correspondentes implementáveis.

Com o submodelo de especificação inicia-se a identificação da solução que incluirá ainda:

- Um modelo fenomenológico, que se fundamenta nos objectivos estabelecidos, nos requisitos e nos componentes dos produtos escolhidos para os atingir combinando-os com a teoria e leis que definem o comportamento esperado do produto. Este modelo inclui a descrição do produto, as limitações dos atributos do modelo em relação aos fenómenos do mundo real e refina objectivos operacionais do produto.
- Um modelo computacional, que a partir das limitações descritas no modelo fenomenológico, define os mecanismos e os procedimentos a ser implementados no

computador. Novas limitações são acrescentadas, causadas pelas utilizações da plataforma tecnológica, onde o produto é implementado pelos recursos computacionais disponíveis.

Exemplo 2.5.1.2.: Como exemplo de um modelo fenomenológico temos o POC (Plano oficial de contas) que define a forma de contabilização das receitas e despesa.
A máquina de estados, que é implementada por um programa de contabilização, codificado em C, é um exemplo de modelo computacional. ⊗

Com base no modelo computacional, o passo seguinte consiste na estruturação da solução. Esta estruturação comporta a construção do modelo das primitivas que constituirão o produto e a forma como elas estão ligadas. É neste modelo que se define como vão ser implementados os requisitos.
A implementação, utilizando o modelo estrutural, constrói o último modelo do problema que é o produto final.

Pela descrição anterior, a forma de progressão para a solução, efectua-se através de desenho de modelos, sucessivamente menos abstractos, que se constroem pela selecção gradual das variadas opções que o engenheiro de software toma. Para se obter uma solução válida, cada um dos modelos terá necessariamente que ser internamente coerente e manter a consistência com todos os restantes.

2.5.2. A construção de cada modelo do processo

Interessa analisar como idealmente se constrói cada um dos modelos mencionados na figura 2.5.1. e como se passa de um modelo para o seguinte.
Na figura 2.5.2.1. representamos diagramaticamente o que acontece quando se passa de um modelo para outro, construído num determinado nível de abstracção. Os modelos são referidos por letras e as transformações por números.

As transformações efectuadas numa fase do processo ideal de desenvolvimento implementam modificações no modelo de entrada, criando à saída um novo modelo construído no nível diferente de abstracção. Idealmente, as transformações aproximam cada modelo de saída da solução pretendida. As transformações incluem mudanças de representação, reestruturação, refinamento e adição de funcionalidades. O engenheiro do software desenvolve a arquitectura

do novo modelo, aplicando o processo humano de decisão, fundamentado no conhecimento dos objectivos imediatos do modelo e do que pretende construir como solução. Como consequência, ao fim de cada passo, o modelo da solução criada necessita de ser validado.
Em cada fase do processo é construído um modelo que, se for satisfatório, é a entrada válida para a actividade seguinte.

Como está representado na figura 2.5.2.1., a construção de cada modelo implica, no início, o uso de uma metodologia que suporte a transformação de notação compreensível para o interlocutor que validou a fase anterior, para a notação usada pelo projectista na metodologia de desenvolvimento dessa fase. A construção do modelo termina com uma transformação de sentido inverso. É necessário realizar uma transformação da notação usada pelo projectista, para uma notação compreensível pelo cliente que validará o modelo construído. Estas transformações são indicadas como 1 e 3 na figura 2.5.2.1.. As acções 1 e 3 são transformações puramente de representação, não contêm em si nenhuma decisão de refinamento ou reestruturação. Com facilidade se verificará a equivalência entre os modelos de partida e chegada. No primeiro caso, entre os modelos A e B, no segundo caso entre os modelos C e D.
Exemplo 2.5.1.2.: É exemplo do esquema 2.5.2.1. a obtenção do modelo de especificação dos requisitos para a construção de um programa de gestão de alunos numa escola a seguir descrito.
O modelo A será a descrição fornecida pelo utilizador, em linguagem natural, dos serviços a implementar; a transformação 1 será realizada usando a linguagem natural estruturada e separando os requisitos dos serviços em funcionais e não funcionais. Obteremos um modelo B que contém a mesma informação que a descrição anterior, traduzida de forma a permitir que o engenheiro que constrói a especificação de requisitos a entenda sem ambiguidades.
A transformação 2 é realizada com a metodologia de definição axiomática que usa as visualizações das propriedades e o comportamento do produto para construir o modelo de axiomas e teoremas a que as propriedades e comportamento do produto têm de obedecer.
O resultado é o modelo C, um dos possíveis modelos de especificação de requisitos do sistema. Por exemplo, o modelo C pode ser descrito na linguagem OCL (*Object Constraint Language*). Nesta transformação o engenheiro analista fez escolhas e acrescentou informação, quando considerou o modelo B incompleto; o modelo C é traduzido para um modelo de Uses

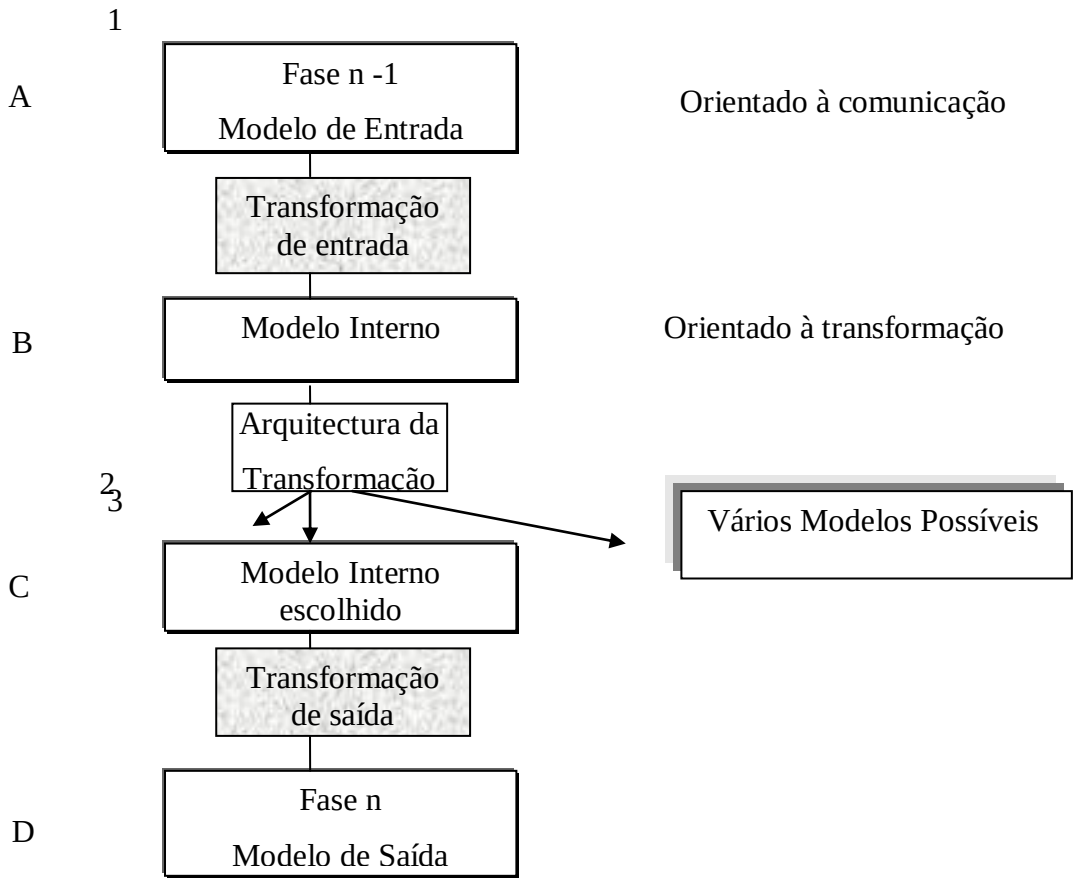


Fig 2.5.2.1. Modelo de criação de cada transformação do processo real

Cases [Jacobson,I.99], para o modelo D, uma vez que a arquitectura e a implementação vão ser executadas em UML numa fase seguinte e por outro técnico. ⊗

A evolução do modelo é realizada pela transformação que se designou por arquitectura na fig. 2.5.2.1. Esta transformação consiste em refinamentos e /ou em reestruturações e é um processo de obtenção dos detalhes necessários acrescentar, da forma de que se deve revestir o seu desenvolvimento futuro, das implicações que vai ter nos modelos anteriormente construídos e das funcionalidades ainda necessárias juntar para os modelos futuros. No fim da arquitectura terá de ser provado que o modelo obtido é uma aproximação válida à solução que pretendemos para o problema. Esta prova consiste na avaliação desta fase do processo no contexto geral de todo o processo.

Em contraste, a verificação da validade da transformação de notação será uma demonstração

na vertical que a transformação é correcta e o que o modelo de saída é idêntico ao de entrada. No fim da fase, novamente ter-se-á de demonstrar que o modelo obtido, neste caso o modelo D, é consistente e completo em relação à funcionalidade que se descrever para essa fase.

2.5.3.O processo real

Como o descrevemos, o processo ideal nunca é executado nos projectos reais de desenvolvimento. No mundo real, o produto engloba vário ciclos de vida e o processo executa modificações, ou acrescentos, nos modelos que vão sendo desenvolvidos.
Existem no entanto tentativas, não conseguidas, de mecanização do processo que pretendem criar um modelo estruturado que seja a decomposição do modelo inicial em sub-modelos do tipo S.

Mesmo que fosse possível mecanizar o processo, através da sua decomposição em sub modelos do tipo S, a selecção dos submodelos admite várias hipóteses e materializa várias soluções possíveis que em diversas fases do modelo podem ser abandonadas em favor de outras mais promissoras. Como tal, a escolha da solução a desenvolver, na prática, é incremental. O desenvolvimento dos submodelos escolhidos é igualmente incremental e, nesse caso, obriga a iterações.
Exemplo 2.5.3.1.: O modelo de transformações de Balzer [Balzer,R.81] usa uma especificação muito precisa, para sobre ela aplicar uma sequência de transformações e obter a solução. O modelo inclui uma actividade não mecanizada de execução e revisão permanente da especificação que permite o desenvolvimento incremental. ⊗

No processo real ocorrem vários tipos de iterações causadas por vários tipos de pressões para a mudança, já referidas.

- A pressão do utilizador que encontra lacunas e erros, quando usa o sistema, origina o aparecimento de várias versões.
- A pressão causada pela necessidade de o cliente validar constantemente a interpretação do problema, realizada pelo técnico, durante o desenvolvimento.
- A pressão surgida, no fim de cada fase, quando se realiza a validação do modelo resultante.

Não sendo o processo ideal praticável na maioria dos casos, podem delinear-se dois tipos de aproximação alternativos para o processo real, que sendo padrões, podem ser considerados paradigmas do processo.
No primeiro paradigma, orientado à completude, considera-se que não se opera nenhuma transformação enquanto o modelo sobre a qual se trabalha não estiver completamente definido. A definição completa quer dizer que todas as questões relativas a esse modelo estão completamente levantadas e estão respondidas de forma única, especifica, ou são consideradas irrelevantes para o desenvolvimento. O modelo não é ideal, na medida que pode haver incrementos dentro de cada fase. O desenvolvimento incremental surge quando é necessário, numa fase, obter a resposta para incorrecções internas ao modelo e, neste paradigma, obrigará, a redefinir os modelos das fases anteriores, para ser obtido o modelo completo inicial.

Exemplo 2.5.3.2.: É exemplo da aproximação de completude o modelo da espiral de Boehm [Bohem,B.88], em que a constante avaliação do risco existente no desenvolvimento indica as áreas onde as tarefas de definição devem ser concentradas. Outro exemplo é o modelo V [German Ministry of Defense92], onde as actividades de teste são sucessivamente usadas para verificar se o sistema está correctamente definido e implementado e se recomeça desde a definição quando se encontra uma incorrecção. ⊗

No segunda paradigma, orientado à evolução, define-se que o desenvolvimento é incremental e engloba mais do que um modelo e o processo desenvolve-se em sequências de vários modelos, sempre que houver necessidade de redefinir ou acrescentar algum dos modelos já definidos, continuando a desenvolver-se sequencialmente depois dessas modificações.
A dificuldade, neste caso, é saber até que modelo retroceder no processo incremental e quando se considera que um dos modelos está completo e se deve prosseguir com as transformações para se ir aproximando da solução.
Exemplo 2.5.3.3.: São exemplos da aproximação alternativa o modelo de programação exploratória, onde se começa por desenvolver a parte conhecida do problema como forma de obter conhecimento sobre as partes não conhecidas que serão então desenvolvidas. ⊗

Concluímos que o desenvolvimento incremental é dependente do processo e intrínseco ao

produto e incluído em qualquer dos paradigmas. Uma vez que a necessidade de iterações aumenta a complexidade do processo, este tem de ser controlado.

Com excepção dos processos de obtenção dos produtos de tipo S, o controlo do processo consiste sobretudo no controlo do tamanho da iteração, seja do controlo do número de modelos ou fases abrangidos, seja do controlo do número de ciclos de cada iteração.

2.5.4. A necessidade do uso de um paradigma de desenvolvimento do processo real

Sendo a totalidade do processo uma sucessão de transformações aplicada a um problema do mundo real a fim de se obter um modelo, de um modelo, de um modelo,... de um modelo final que será a solução, a escolha quer da sequência dos modelos quer dos próprios modelos confere complexidade ao processo.

A complexidade do processo gera a necessidade do uso de um paradigma de desenvolvimento.

A utilização de um paradigma orienta o profissional para a escolha dos modelos e sua sequência e permite, atendendo a que existe uma estratégia de trabalho, dividir tarefas, entre pessoas que conhecem a definição global do sistema, e facilitar o controlo do processo. Como se está a lidar com o processo de criação intelectual, o paradigma utilizado tem de ser dinamicamente ajustado, quer aos profissionais que o usam, quer ao tipo de solução que se vai procurando.

Durante o desenvolvimento, é forçoso estabelecer uma relação de equilíbrio entre o profissional que necessita de flexibilidade para desenvolver um processo criativo e a empresa para quem não só as normas de procedimento internas devem ser respeitadas, como o resultado do trabalho tem de ser consistente e analisável numa perspectiva global por qualquer técnico. O uso de um paradigma de desenvolvimento auxilia a conseguir esse equilíbrio porque fornece a linguagem de notação e a organização dos procedimentos comuns que tornam a dinâmica do processo inteligível para todos os participantes.

Existem vários modelos descritos para o processo.

Em 1970, Royce descreveu o modelo do processo, designado por cascata ("*waterfall*"), em que as fases de um sistema (definição de requisitos, arquitectura, teste e implementação e manutenção) se realizam sequencialmente. A vantagem principal do modelo de cascata traduziu-se na visibilidade que trouxe às fases de desenvolvimento. O modelo de cascata tem

também a vantagem de ser consistente com a programação estruturada descendente ("*top-down*"), desenvolvida por Mills [Mills,H.86] e como representava uma analogia de um modelo usado para desenvolvimento de projectos de Engenharia Civil, muito conhecido e testados, teve grande aceitação.
Tal como indicado na secção 2.5.1., um processo ideal não é seguido no mundo real: a condição de só se passar de uma fase para a seguinte quando não existe indefinição levaria à impossibilidade do desenrolar o processo.
Fundamentando-se na visibilidade que o modelo de cascata conferiu às várias fases do processo, surgiram outros modelos, baseados igualmente na ideia da separação das fases, mas permitindo a incrementação. São exemplos, o paradigma incremental de Mills [Mills,H.86] e o modelo V [German Ministry of Defense92]. Classificamos estes dois modelos como orientados à completude.

Com o objectivo de não fazer depender a construção da solução do conhecimento completo da especificação de um problema, surgiram outros modelos orientados à evolução.
O modelo evolutivo ou incremental, consiste em desenvolver uma implementação inicial incompleta, validá-la junto dos utilizadores, obter com essa validação mais informação sobre o produto e refiná-lo de seguida, juntando mais funcionalidade ao sistema.
Essa ideia foi também seguida por Barry W. Boehm [Boehm,B.88], no modelo evolutivo do processo que usa a noção de avaliação do risco como elemento que define a estratégia do prosseguimento do processo. A equipa de desenvolvimento não segue um único modelo para cada ciclo da espiral ou mesmo para todo o processo. O modelo toma a forma de uma espiral em que cada volta representa uma fase desenvolvida do processo, usando o modelo mais apropriado para eliminar o risco da solução identificado no início da fase. Esta classe de modelos apresenta também desvantagens. Mesmo sendo desenvolvido por pessoas altamente preparadas e motivadas, que conseguem ter uma visão global do problema, o produto não deixa de ser construído por acrescentos tornando-se cada vez mais complexo e menos estruturado.
Também é difícil implementar um sistema de métricas do processo, atendendo a que não será rendível produzir documentação formal de todas as versões, por serem em grande número.

Do que foi dito pode parecer que todos os modelos são evolutivos. Embora consideremos que

todo o processo terá de ser incremental, os dois paradigmas apresentam estratégias diferentes. Assim, nos paradigmas orientados à completude o produto é construído a partir de uma estratégia de compreensão global. Nos paradigmas orientados à evolução é privilegiada a resolução parcial (da parte conhecida), o que conduz a produtos complexos porque construídos com acrescentos. O pressuposto é, pois, completamente diverso e o resultado, em relação ao controlo da complexidade do produto resultante, também.

Os dois paradigmas brevemente descritos aplicam igualmente à construção de novas versões. Cada novo ciclo usa um paradigma e dá origem a uma nova versão. Quando são usados paradigmas orientados à completude, os múltiplos ciclos de desenvolvimento executam-se sempre iniciando toda a sequência das fases do processo, uma vez que a especificação global do problema tem de estar clarificada antes de qualquer fase de desenvolvimento. Quando são usados paradigmas orientados à evolução, os diversos ciclos de desenvolvimento vão incorporando as fases que o gestor do processo determina como necessárias para obter a nova funcionalidade do produto.

2.5.5. O desenvolvimento e controlo do processo real

Os paradigmas do desenvolvimento do processo ideal, descrito na secção 2.5.5., são aplicados com omissões e adaptações no desenvolvimento real do produto. Os pontos de vista, quanto ao processo, dos investigadores académicos e dos gestores do desenvolvimento nas empresas comerciais são diferentes.

Os académicos preconizam que o processo deve ter as suas fases bem definidas: uma fase de definição e outra de implementação. Embora considerando válido que dentro de cada fase seja usada iteração e prototipagem, indicam que os requisitos e a arquitectura têm de estar bem compreendidos e bem documentados antes de se iniciar a implementação. Os gestores encaram as propostas dos académicos como princípios ideais, que são uma base sólida para a construção do processo individual a seguir por uma organização comercial e não como normas absolutas a seguir.

A experiência da autora na participação em desenvolvimentos comerciais, em vinte anos, levou à conclusão que o processo real deve incluir o rigor das metodologias e a aprendizagem com os erros, o que obriga à definição de medidas e à escrita de documentação.

Consideramos que:

- A incerteza em relação a uma série de condicionantes do processo e a mudança constante

do ambiente onde ele será instalado são uma realidade com que as equipas do processo têm de lidar.

- A gestão do risco inerente a estes dois factores é fundamental.
- Uma aproximação eficiente é aquela que, baseada em princípios gerais e regras robustas, permita adaptabilidade e controlo da complexidade do produto sem impedir a criatividade da equipa de desenvolvimento. Deve também incluir formas de antecipação das excepções que surjam ao longo do processo.

O controlo de qualquer projecto implica a existência de um plano, contendo as previsões do desenvolvimento do trabalho e o processo de medição a usar para obter indicadores que permitam verificar se o desenvolvimento está de acordo com o estabelecido ou necessita de correcções. As estimativas de tempo e de recursos são usadas, habitualmente, como referências, nas tarefas de controlo do processo.

Normalmente, nos processos reais é exercido controlo. Mesmo quando estamos perante um problema mal definido e, para o resolver, foi escolhido um modelo evolucionário, ou até quando não existe processo definido, mas existem condicionantes de tempo, recursos e exigência de atributos do produto que obrigam a estabelecer uma estratégia de controlo do processo [Booch,G.95;Reifer,D.98]. A medida proposta nesta dissertação será adequada também para estes casos. A sua justificação baseia-se nas considerações seguintes:

Tradicionalmente, as equipas projectistas têm usado os seguintes atributos para controlar os processos:

- calendário estabelecido para a entrega de produtos e de documentação de suporte à avaliação (baseado nas estimativas),
- nível de cumprimento dos requisitos,
- nível de qualidade da arquitectura do produto,
- nível de qualidade do produto,
- documentação necessária.

Existem várias dificuldades a ter em conta durante o controlo do processo.

Primeiro, não é realizável controlar um processo, controlando a totalidade dos atributos mencionados na definição 2.4.1.2., porque são conflituosos e a sua maximização conjunta não é possível.
Exemplo 2.5.6.1.: Um processo em que a principal preocupação do gestor é o calendário leva a que a equipa de desenvolvimento esteja espartilhada no tempo de que dispõe para apresentar resultados e abrevie as tarefas de inspecção de código, não controlando o número de erros existente no produto e consequentemente a sua qualidade. Pelo contrário, um processo em que é maximizada a importância dos requisitos tem tendência a não cumprir o calendário, por gastar muito tempo a verificar, se foram construídas todas as funcionalidades expressas nos requisitos. ⊗

Segundo, para basear as suas decisões de controlo, o gestor do processo necessita de documentação da descrição do estado do projecto, em tempos que não são coincidentes com os fins de cada etapa. Isto acontece por que toma decisões de refinamentos e acrescento de funcionalidade como se viu na secção 2.4.1., durante cada etapa. Esta necessidade pode obrigar, ou a ter um registo permanente ou a elaborar documentação de descrição intermediária, o que torna o processo de controlo mais custoso.
A validação em cada actividade do processo leva algum tempo, durante o qual, normalmente o processo abranda a velocidade de desenvolvimento. As pressões para cumprimento de prazos levam à continuação do processo sem aprovação, obrigando a retroceder para repor resultados da validação. Uma pequena modificação num dos modelos pode causar grande necessidade de reestruturação dos modelos anteriores.
A visibilidade do processo, através da sua documentação, não é igual em todos os modelos. Para o modelo da catarata e para o modelo da espiral cada actividade dá origem a um documento, que pode ser validado, e ao mesmo tempo permitir avaliar do estado do processo e da possibilidade de cumprimento da calendário e custos planeados. Nos desenvolvimentos de software, com base no modelo evolucionário, é claramente pouco económico para o gestor do processo produzir documentação para cada fase e o esforço da execução da documentação ultrapassaria o esforço de desenvolvimento. Em alternativa, normalmente, o gestor mantém registos na operacionalidade, que no fim do processo serão transformados em documentação.

Terceiro, os pontos de vista do director de projectos, do director do projecto em

desenvolvimento, do técnico que trabalha no projecto, do utilizador do produto e do cliente que o encomendou são diferentes e conflituosos em relação a objectivos atingir com a actividade de controlo. O director dos projectos quer os prazos cumpridos e os recursos poupados. O director do projecto em desenvolvimento tem de gerir a pressão do director para o cumprimento de prazos, conjuntamente com a capacidade de trabalho da equipa que dispõe para o desenvolvimento. O técnico quer cumprir o seu trabalho técnico e considera geralmente que os prazos são da responsabilidade do gestor e se não são cumpridos é por que foram mal estabelecidos. O utilizador pede funcionalidades que não necessita, enquanto que o cliente que as terá de pagar vai tentar que seja implementada a funcionalidade mínima que permita a resolução do problema.

Consideramos, por todas estas razões, que uma outra alternativa de controlo do processo baseada numa estratégia diferente tem de ser considerada. A alternativa será dispor de uma avaliação global e sucessiva do produto, quer durante a fase de construção, quer durante a fase de evolução.

2.5.6. A dinâmica não linear do processo real

Na secção 2.5.3. foram analisadas as causas da evolução do produto, tendo-se constatado que as referidas causas obrigam a que o processo seja incremental.

Um sistema dinâmico muda ou evolui no tempo [Garnett,P.97]. O processo é considerado dinâmico quando:

- ao longo do tempo apresenta-se em vários estados diferentes,
- ao longo do tempo de desenvolvimento, emprega diferentes metodologias de construção e obtém como resultado modelos de tipos diferentes de representação da solução,
- depois da primeira operacionalidade, ao longo da evolução, os produtos aparecem numa sucessão de versões, uma vez que o objectivo inicial da resolução do problema proposto, mesmo depois de atingido, se altera devido a necessidades intrínsecas do produto criado ou a modificações do ambiente onde opera.

Exemplo 2.5.6.1.: São exemplos dos vários estados do processo a escrita, a partir da definição do problema dos requisitos, a execução de testes dos módulos que constituem o produto, a

construção de uma nova versão de um sistema já entregue, etc.
São exemplos das modificações em metodologia usada e modelos construídos ao longo do processo: a transformação do modelo de requisitos em modelo de arquitectura, a codificação dos módulos da arquitectura, etc.
São exemplos de evolução por mudança de objectivos as modificações que ocorrem quando uma publicação de uma lei muda o enquadramento jurídico do problema que preexiste ao produto e obriga a reiniciar o processo como forma de adaptar o produto a esta nova realidade. ⊗

A dinâmica dos processos reais revela características de não linearidade e de adaptabilidade [Kitchenham,B.98].
Vejamos porque é que o processo é não linear. Uma tarefa, realizada em qualquer das fases, depende significativamente das anteriormente realizadas e influencia também da mesma forma as fases seguintes. Se não existisse esta interdependência de tarefas, teríamos o equivalente a uma linha de montagem e o processo seria uma combinação linear da sequência das tarefas.
A não linearidade do processo confere-lhe uma dependência sensível de várias condicionantes que, quando actuam, podem alterar drasticamente o estado de desenvolvimento do processo.
Exemplo 2.5.6.2.: É exemplo da não linearidade a propagação exponencial de erros ao longo das fases do sistema. Um erro não corrigido inicialmente causa um número de erros com crescimento exponencial nas fases seguintes. Este exemplo também é indicativo da dependência sensível do número de erros de uma fase no número de erros das fases seguintes.
É exemplo da dependência sensível de certas condicionantes o aparecimento de uma nova tecnologia que obrigue à modificação total do projecto. Um exemplo concreto desta situação é a experiência da autora na reformulação completa do software, na Portugal Telecom, que estava em fase de testes dos telemóveis de segunda geração face à disponibilidade dos telemóveis de terceira geração. ⊗
A dinâmica do processo é adaptativa porque as modificações do domínio do discurso do produto e da envolvente tecnológica do processo obrigam a que o processo vá, em cada estado, mudando, de forma a responder a essas mudanças.
Exemplo 2.5.7.3.: É exemplo da adaptabilidade do processo a mudança da plataforma de desenvolvimento de Windows NT para Windows 2000, porque a rede onde o produto ficará

operacional mudou de Windows NT para Windows 2000 ou a entrega a uma equipa exterior subcontratada a execução de uma fase, porque da análise do risco dessa fase se concluiu que a equipa de desenvolvimento não tinha capacidade de a realizar. ⊗

Tal como é descrito na secção 2.4.1., a dinâmica do processo é organizada de duas maneiras diferentes:

- Sequencial, quando uma das tarefas mencionada da definição 2.4.1. termina e se inicia outra diferente.
- Incremental, quando a informação, as ferramentas e as metodologias relevantes a uma única tarefa são aplicadas sucessivamente aos vários níveis de estruturação dessa tarefa até à sua completa realização e quando as actividades de verificação e de validação obrigam a mudar ou acrescentar funcionalidades ao produto ou quando o problema de que o produto é solução ou o seu ambiente mudam e causam a evolução do produto.

As duas formas reveladas pela dinâmica do processo, são fundamentais para o considerar como sistema complexo, tal será detalhado no capítulo 4.

Aparentemente seria de pensar que o carácter dinâmico dos processos não conduziria à identificação de leis gerais e o processo seria impossível de controlar. Tal como essa afirmação é inválida em domínios físicos como, por exemplo, as leis físicas relacionando a pressão e temperatura dos gases são observadas apesar do comportamento dinâmico dos átomos, também Lehman [Lehman,M.85] enunciou, qualitativamente, leis gerais que regulam a evolução do software:

- *Lei da mudança contínua: um produto em uso ou está em mudança constante ou se torna progressivamente menos útil.*
- *Lei do aumento da complexidade: a mudança contínua introduz continuamente complexidade no produto deteriorando a estrutura do produto. Se não for desenvolvida nenhuma actividade explícita do controle da complexidade a manutenção do produto deixa de ser possível e este torna-se inútil.*
- *Lei fundamental da evolução do produto: o processo de evolução do produto é uma*

dinâmica auto regulada com tendências estatísticas determináveis e invariâncias.

- *Lei da conservação da estabilidade organizacional: durante a vida activa do produto a taxa de actividade global é invariante.*
- *Lei da conservação da familiaridade: durante a vida activa do produto o conteúdo de cada versão é estatisticamente invariante.*

Nota-se que a complexidade usada na segunda lei é informal e deve ser interpretada como a noção intuitiva de complexidade que indica a dificuldade de redução do tamanho da especificação do produto.

As cinco leis, fundamentadas nas causas da evolução do produto, justificam e enquadram a tentativa de obter um modelo matemático que permita controlar, a partir da evolução da complexidade do produto, a sua convergência para a solução na fase de desenvolvimento e o controlo da sua complexidade na fase de evolução.

No capítulo 4, abordam-se as propriedades dos sistemas ditos complexos que apresentam características análogas às descritas para o produto. No capítulo 5 analisam-se programas com medidas usadas na área dos sistemas complexos.

2.6. Síntese do capítulo

Este capítulo enquadra o processo segundo os itens seguintes:

- A interpretação do processo de software, como um processo adaptativo, que engloba vários tipos de dinâmica. Assim, em geral, o processo é não linear.
- A verificação que o processo obedece a leis gerais, embora empregando uma variada gama de metodologias para construir o produto e como tal sendo expectável que apresentasse grande diversidade de comportamentos**.**
- A necessidade de controlar o processo, nas versões sucessivas do produto.

A contribuição desta dissertação é resumida na seguinte **hipótese de investigação**:

O produto de software é modelado por um sistema complexo e é possível identificar um modelo matemático para medir a complexidade da sua estrutura.

O processo de desenvolvimento de software tem uma dinâmica complexa possível de modelar por um modelo matemático.

3. As Métricas do Software

...A maior diferença entre uma área científica "bem desenvolvida" como a Física ou outra menos "bem desenvolvida" como a Psicologia ou a Sociologia é a capacidade de medição disponível...
Fred Roberts 1990

3.1. Introdução

Neste capítulo, com o objectivo de evidenciarmos as suas limitações, apresentamos um levantamento das métricas usadas tradicionalmente em Engenharia de Software.
Enunciamos a teoria representacional da medida como base teórica da construção das métricas de software.
Descrevemos brevemente os tipos de métricas usadas na medição do processo e do produto. Visto serem o assunto principal desta dissertação, prestamos especial atenção às métricas de controlabilidade do processo, e às métricas da complexidade do produto. Verificamos que, embora a controlabilidade forneça uma indicação do grau global deste atributo, o seu uso para controlar o processo, durante o seu desenvolvimento, só é eficaz quando o grau de visibilidade de todas as fases do processo é idêntico, o que na prática se verifica com pouca frequência.
Concluímos que as métricas tradicionais da complexidade do produto permitem apenas uma caracterização parcial do mesmo, e, como tal, não são as mais adequadas ao controlo do processo de desenvolvimento.

3.2. Necessidade do uso das métricas

A Engenharia de Software tem sido definida por diversos autores [IEEE90;Pressman,R.94; Conte,S.86;Sommerville,I.96;Fenton,N.97;Naur,P.69] podendo ser designada como a aplicação de uma aproximação sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção do software.

O carácter quantificável torna fundamental a utilização de métricas no controlo do processo.

Definição 3.2.1.: Métrica é uma função que permite a obtenção de valores de um ou mais atributos duma entidade. Cada valor recolhido de uma entidade é designado por medida da entidade. ⊗

Com base nas medidas, pode-se comparar processos distintos de desenvolvimento, e prever custos e duração de projectos (por exemplo, COCOMO). As medidas são igualmente necessários para determinação da qualidade dos produtos (ISO9000).

Como se definiu no capítulo 2, o processo consiste numa sequência de transformações em modelos do problema proposto. O processo termina quando é obtido um modelo final que configura uma solução aceitável [Pressman,R.94;Sommerville,I.96; Lehman,M.85].
Em cada uma das fases do processo é obtido um modelo do problema, cujo tipo depende da metodologia usada para construir esse modelo. As métricas têm o seu domínio nos atributos específicos de cada um desses modelos do problema.
Para cada tipo de modelo podem ser construídas variadas métricas. O engenheiro de software identifica as métricas adequadas a cada caso, seleccionando as que respondem às suas necessidades de informação na gestão e controlo do processo.
Em consequência, os técnicos responsáveis pelo controlo do processo identificam métricas diferentes, não só para os vários tipos de processo e diversas fases/modelos do processo, mas também para as diferentes perspectivas de análise desse processo.
Exemplo 3.2.1.: Consideremos um engenheiro de software que pretenda controlar a legibilidade dum projecto definido segundo o modelo da cascata. Encontrando-se na fase de obtenção do modelo do diagramas de fluxo de dados, ele mede entre outros atributos possíveis o número de processos desenhados em cada nível sempre que os analistas produzem uma versão. O referido técnico não medirá número de linhas de código porque, neste modelo e nesta fase, não dispõe ainda deste produto. Se se encontrasse na fase de codificação, usaria, para o mesmo fim, por exemplo, métricas de acoplamento e coesão entre módulos construídos, porque nesta fase a medida do número de processos desenhados que servem de base ao algoritmo do programa é menos eficaz para evidenciar adequadamente a legibilidade. Arquitecturas distintas, com o mesmo número de módulos diferentemente coesos e acoplados,

terão compreensibilidade diferentes. ⊗

Uma vez que as métricas quantificam apenas alguns atributos do processo, ou do produto, coloca-se naturalmente a questão de saber se, com as medidas recolhidas, é possível obter uma ideia global do processo e do produto.

Para responder a esta questão, na secção 3.3 descrevemos como se identificam métricas e nas secções 3.4 a 3.6 listamos as métricas mais usadas em Engenharia de Software.

3.3. A construção de uma métrica

Nesta secção adoptamos a teoria representacional da medida como modelo a usar na definição de métricas. A razão desta escolha prende-se com o objectivo que as métricas, objecto desta dissertação, meçam atributos a que corresponda, sem ambiguidades, uma entidade física do domínio do discurso.

3.3.1.A teoria representacional da medida

A teoria representacional da medida [Zuse,H.98;Fenton,N.97] propõe o fundamento teórico para a construção de uma métrica, baseado na existência de três entidades, a seguir indicadas:

- um mundo real e as relações empíricas observadas entre os atributos das entidades que lá existem.
- um espaço formal de símbolos, por exemplo números, onde também se verificam relações.
- um mapeamento entre o mundo real e o espaço de símbolos que obedece a uma condição de representação.

A condição de representação é assim definida:

Definição 3.3.1.1.: A condição de representação é um mapeamento que preserva as relações empíricas, fazendo corresponder aos objectos do mundo real símbolos do mundo formal. ⊗

O mapeamento deve verificar a relação de monotonicidade. i.e., a ordem entre dois programas do mundo real P_i e P_i deve ser mantida nas medidas dos mesmos programas, respectivamente $M(P_i)$ e $M(P_i)$

$P_i > P_j \quad \Rightarrow \quad M(P_i) \geq M(P_j)$

Esta condição de representação traduz a coerência da métrica.

Obtêm-se relações empíricas entre entidades, por comparação dos seus atributos, observando como o mundo real funciona. No caso do processo de Software, observa-se o que acontece nas várias fases e estabelecem-se relações entre os produtos e documentação que vão sendo sucessivamente construídos.

Exemplo 3.3.3.1.: Se se considerar que o tamanho de um programa desenvolvido num determinado ambiente de programação indica que esse programa é complexo, uma métrica da complexidade pode ser construída com base na relação de ordem dos inteiros.

Sendo (P1, P2, P3...P4) o conjunto de programas a medir, a relação empírica será Pi >Pj se a listagem do programa Pi é maior ou igual do que a listagem de Pj.

Medir, ou usar a métrica, é obter uma instância de uma função que mapeia o mundo empírico real num mundo relacional e formal, sendo obrigatoriamente satisfeita uma condição de representação.

A medida é o número ou símbolo atribuído a uma entidade, caracterizando um seu atributo.

Assim, é possível construir várias métricas válidas para cada atributo. Qualquer mapeamento que verifique a condição de representação é considerado como uma métrica válida.

Diz-se que uma relação empírica, verificada no mundo real, é tanto mais completa quanto maior for número de entidades que conseguir relacionar.

Quanto mais completa for uma relação empírica, menos métricas válidas são possíveis construir a partir dela, visto a sua construção exigir um maior número de condições de representação a que uma métrica deve obedecer.

É possível existirem métricas distintas para um único atributo de uma entidade.

O conjunto do mapeamento do sistema relacional empírico e o sistema relacional numérico, o domínio e o contradomínio do relacionamento, são referidos como a escala da métrica.

Habitualmente, as várias métricas construídas para um atributo têm o mesmo domínio e contradomínio, nesse caso simplifica-se a noção de escala, dizendo que o mapeamento é a escala.

Embora existam outros tipos de escala, há cinco classes de escalas que são as mais utilizadas: Nominal, Ordinal, Intervalar, Por Razão e Absoluta [Briand,L.95].
A escala Nominal separa entidades em categorias, não havendo qualquer relação entre as categorias estabelecidas. A escala Ordinal é uma escala nominal, onde se verifica, adicionalmente, uma relação de ordem entre as categorias. A escala Intervalar é uma escala ordinal, mas que captura, além da ordem, o tamanho dos intervalos que separa as classes. Uma escala Por Razão é uma escala intervalar que preserva, adicionalmente, as razões entre as entidades. A escala Absoluta é a escala que atribui a cada categoria o número de elementos que contem.

3.3.2. Construção das métricas de Engenharia de Software

Como se referiu na secção 2.3., as métricas usadas em engenharia de software são construídas, quer sobre os atributos dos modelos do problema que sucessivamente são obtidos ao longo do processo, quer sobre os atributos dos tipos do processo que se usam para o desenvolvimento.
O uso de modelos em substituição da realidade obriga à satisfação dos seguintes requisitos:

- Conhecer bem a realidade, a fim de saber o que é relevante para a caracterizar e estabelecer um modelo útil da mesma.
- Conhecer bem o modelo do mapeamento das entidades em números ou símbolos, para garantir que não só a condição de representação é sempre satisfeita, como também para entender o que as medidas obtidas explicam da realidade.

O primeiro requisito justifica que existam, em Engenharia de Software, mais métricas do produto do que do processo, visto este último ser menos compreendido [Pressman,R.94; Sommerville,I.96;Lehman, M.85; Fenton,N.97].

Conhecendo-se os modelos de entidades e um conjunto dos seus atributos, podem estabelecer-se medidas de duas formas:

- Medidas directas, atribuindo directamente a um atributo um número ou um símbolo. Na obtenção dos valores de uma medida directa só estão envolvidos a entidade e um único atributo que é o objecto da medida.

Exemplo 3.3.2.1.: São exemplos, o número de erros encontrado na fase de teste de um programa e o tipo de modificações executadas num programa. ⊗

- Medidas indirectas, formadas por uma relação entre vários dos atributos quantificados. Na sua obtenção podem estar envolvidos vários atributos de uma entidade ou várias entidades.

  Exemplo 3.3.2.2.: São exemplos de medidas indirectas, a densidade de erros encontrada por linha de código. ⊗

As medidas indirectas podem ser complementares às medidas directas, na medida que evidenciam as interacções que se verificam entre medidas directas e lhes acrescentam significado.

Por vezes, um atributo de uma entidade só pode ser medido a partir de um conjunto de sub-atributos. Nesse caso, a métrica constrói-se de forma análoga à já descrita para a medição directa. O mundo formal será, neste caso, o conjunto dos n-tuplos formados pelos produtos cartesianos dos espaços de números ou símbolos em que cada sub-atributo é mapeado. Uma descrição matemática deste modelo e a da forma com que é assegurada a condição de representação pode ser encontrada em [Fenton,N.97].
Exemplo 3.3.2.3.: Um exemplo de uma métrica baseada em sub-atributos é a métrica da qualidade do produto estabelecida na norma ISO9126. Os seis sub-atributos são a funcionalidade, a confiabilidade, a eficiência, a usabilidade, a manutibilidade e a portabilidade. Aqui o mapeamento far-se-á entre qualquer um dos modelos construídos durante o processo e R^6. ⊗

Outra classificação utilizada [Fenton,N.97;Conte,S.86; Pressman,R.94] separa as métricas de software em dois tipos, internas e externas, de acordo com o tipo de atributos medidos.

Definição 3.3.2.1.: Um atributo diz-se interno quando pode ser medido por observação do processo, do produto, ou dos recursos do processo, independentemente do comportamento dos mesmos em relação ao ambiente onde existem.

Um atributo diz-se externo quando mede a forma como o produto, o processo ou os recursos do processo se relacionam com o ambiente. ⊗

Exemplo 3.3.2.4.: São exemplo de métrica interna os erros de codificação encontrados pelo programador. É exemplo de métrica externa a taxa anual de falhas do sistema. ⊗

3.4. Métricas do processo

As métricas do processo têm vindo a ser uma área importante de investigação, atendendo à necessidade da sua avaliação como garantia de melhoramento do processo e factor principal em estudos de capacidade de manutenção de um produto [Briand,L.97].
São poucos os atributos internos do processo que podem ser medidos directamente. Vários autores [Fenton,N.97; Conte,S.86; Pressman,R.94] indicam três tipos:

- Duração, quer do processo, quer das actividades.
- Esforço, associado à execução das tarefas e aos agentes que as executam.
- Localização e contagem de incidentes (erros, modificações), ocorridos durante o desenvolvimento do processo.

Para aumentar a quantidade de informação fornecida por estas métricas directas, são usadas métricas indirectas construídas a partir das métricas directas.
Exemplo 3.4.1.: São exemplos de métricas internas do processo o custo/esforço por unidade de erro encontrada numa inspecção ou teste. O custo médio de erro encontrado numa inspecção é o custo total da inspecção a dividir pelo número de erros encontrados. O custo médio do erro encontrado na fase de teste de integração é o custo da fase de integração a dividir pelo número de erros encontrados nessa fase. ⊗

Os atributos externos do processo, tradicionalmente medidos pelo modelo de maturidade do processo (CMM) do Software Engineering Institute (SEI) [Pfleeger,N.90;Paulk91], são a observabilidade, a controlabilidade e a estabilidade.
A quantificação destes atributos é feita indirectamente, com recurso ao modelo de maturidade que classifica o processo numa escala ordinal pela análise da totalidade dos seus atributos. A medição é realizada utilizando um questionário, cujas questões referem os vários atributos do

processo. Conforme a resposta seja positiva ou negativa, as questões são pontuadas. O somatório dos pontos classifica o processo num nível determinado e indica a necessidade, caso esse nível seja baixo, de reorganizar o processo como forma de obter um nível mais alto. Nos níveis baixos da escala de maturidade os técnicos que executam o processo não o compreendem, uma vez que o processo não está sequer definido.

À medida que o nível de maturidade aumenta o processo torna-se melhor definido e, como tal, mais compreendido. Cada nível indica igualmente a observabilidade, a controlabilidade e a estabilidade do processo.

No entanto, não existe uma definição consensual dos atributos a medir. Existe um certo consenso em relação à formulação dos modelos que permitem medi-los. O raciocínio que leva ao estabelecimento do consenso é muito semelhante para os três conceitos acima mencionados.

3.4.1.Observabilidade

Nos dois primeiro níveis do modelo CMM [Pfleeger,N.90;Paulk91], (processo "ad hoc" e repetível), a observabilidade do processo é nula. No nível repetível, apesar de o gestor saber que há actividades, não planeia a sua execução e apenas se preocupa com o controlo do resultado das actividades. Em conclusão, no segundo nível a observabilidade é restrita aos executantes das actividades.

A partir do terceiro nível (processo definido) é possível medir o estado em que o processo se encontra em relação à sua definição e a qualidade dos produtos que produz. Nos quarto e quinto níveis (processo gerido e optimizado) aumenta a observabilidade porque é possível observar as fases do processo e as decisões que levam à mudança de fases. A possibilidade de efectuar medições do processo está dependente da visibilidade do processo, porque só pode ser medido o que é visível. Logo as métricas que podem ser obtidas ao longo de um processo são um bom indicador do seu grau observabilidade.

Exemplo 3.4.1.1.: A visibilidade pode ser medida pelo número de fases definidas para o processo e o número de documentos obtidos no final de cada fase. ⊗

Na tabela 3.4.1.1. estão descritas as métricas que se podem obter em cada nível.

Tabela 3.4.1.1.- Relação entre os níveis do processo do CMM e a observabilidade do processo

Nível	Definição do Nível	Métricas Possíveis
1.Ad-hoc	Entradas mal definidas Saídas expectáveis Processo não controlado	Apenas depois do processo finalizado podem ser avaliados os recursos utilizados e medida a solução obtida
2.Repetível	Entradas e limitações definidas Recursos disponíveis avaliados Saídas definidas Processo não definido	Tamanho e volatilidade dos Requisitos Esforço, Limitações de custos e de calendário Tamanho do Código e Documentação
3.Definido	Entradas e limitações definidas Recursos disponíveis avaliados Saídas definidas Processo definido	Métricas sobre atributos: de requisitos, da arquitectura, dos testes, do código e da documentação Esforço, limitações de custos e de calendário, por fase do processo. Métricas dos vários modelos da solução ao longo do processo
4.Gerido	Entradas e limitações definidas Recursos disponíveis avaliados Saídas definidas Processo definido e gerido	Uso das medidas obtidas em cada fase como informação para estabelecer prioridades de actuação nas fases seguintes ou redesenho das anteriores Métricas de avaliação da eficiência da globalidade do processo
5.Optimisado	Entradas e limitações definidas Recursos disponíveis avaliados Saídas definidas Processo definido, gerido e optimizado	Métricas das actividades usadas para melhorar o processo

Pelo tipo de medidas que se podem obter durante um processo podemos concluir o grau de observabilidade do mesmo. Nesse caso usamos os níveis do modelo como medidas da observabilidade.

A aplicação deste modelo apresenta problemas porque num processo existem sempre partes (que podem ser fases ou mesmo sub-fases) com graus de maturidade e organização diferentes. Para essas partes faz sentido a recolha de medidas.
Exemplo 3.4.1.2.: Num processo classificado no nível 2 (sem visibilidade), uma das suas fases pode ter uma actividade de teste bem definida. Para essa actividade faz sentido a recolha de número de erros identificados e o uso desta métrica como informação para gerir o processo, o que é característico de um processo de nível 4. ⊗

Consideramos que, embora a aplicação do modelo de maturidade forneça uma indicação com significado do estado de observabilidade do processo, esse grau de observabilidade é um grau global e torna-se necessário, na sequência da sua aplicação, obter mais informação sobre se a observabilidade de cada fase é diferente. Para essas situações não existe nenhum modelo de avaliação estabelecido. A prática seguida consiste em identificar as métricas características, e possíveis de obter em cada fase, e considerar como de baixa observabilidade as fases onde não existem medidas possíveis de obter (classificação qualitativa). Essas fases serão alvo de reestruturação para aumento da observabilidade global do processo, e esta, medida de acordo com os níveis do modelo de maturidade.

3.4.2.Controlabilidade

A controlabilidade está igualmente ligada à visibilidade do processo e ao prévio estabelecimento do modelo de tomada de decisão ao longo do mesmo.
No primeiro nível (processo "ad-hoc"), não é possível estabelecer qualquer estratégia de controlo.
No segundo nível (processo repetível), a visibilidade do processo continua nula, mas estando definidas as limitações do processo é possível efectuar o controlo das limitações. Assim o calendário, o orçamento e as normas gerais de organização do trabalho podem ser controlados durante o desenvolvimento do processo.
No terceiro nível (processo definido), uma vez que as fases estão individualizadas, é possível o controlo da evolução da qualidade dos sucessivos modelos que vão sendo construídos, bem como o controlo dos riscos de desenvolvimento e a produtividade de cada fase.
No quarto nível (processo gerido), o controlo de cada fase é realizado pela avaliação permanente da eficiência de cada uma das suas actividades. São efectuados o controlo e disponibilização dos recursos, fundamentados na avaliação contínua do processo. A

informação de controlo é utilizada quer no redesenho de fases anteriores quer na modificação do tipo de abordagem a fazer às fases seguintes.

Tabela 3.4.2.1. Relação entre os níveis do processo do CMM e a controlabilidade do processo

Nível	Definição do Nível	Controlos Possíveis
1.Ad-hoc	Entradas mal definidas, Saídas expectáveis, Processo não controlado.	Não é possível qualquer actividade de controlo.
2.Repetível	Entradas e limitações definidas, Recursos disponíveis avaliados, Saídas definidas, Processo não definido.	Controlo das limitações: Calendário, Orçamento, Normas gerais.
3.Definido	Entradas e limitações definidas, Recursos disponíveis avaliados, Saídas definidas, Processo definido.	Controlo da evolução da qualidade dos sucessivos modelos que vão sendo construídos. Controlo dos riscos de desenvolvimento e produtividade no fim de cada fase.
4.Gerido	Entradas e limitações definidas, Recursos disponíveis avaliados, Saídas definidas, Processo definido e gerido,	Controlo de cada fase pela avaliação permanente da eficiência de cada uma das suas actividades. Controlo e disponibilidade dos recursos disponíveis fundamentados na avaliação continua do processo.
5.Optimizado	Entradas e limitações definidas, Recursos disponíveis avaliados, Saídas definidas, Processo definido, gerido e optimizado.	Controlo da mudança do processo provocada pela optimização.

No quinto nível (processo optimizado), o tipo de controlo exercido é um controlo da mudança do processo provocada pela optimização.

O tipo de controlo que é possível organizar ao longo de um processo é um bom indicador do seu grau controlabilidade.
Na tabela 3.4.2.1. estão descritos os controlos realizáveis em cada nível. Pelo tipo de controlo realizável durante um processo podemos concluir o grau de controlabilidade do mesmo. Nesse caso, usamos os níveis do modelo como medidas da controlabilidade.
Neste caso, também, a aplicação deste modelo apresenta problemas, porque num processo existem sempre partes (que podem ser fases ou mesmo sub fases) com graus de maturidade e organização diferentes. Para essas partes faz sentido a implementação de tipo de controlo diferente.
Exemplo 3.4.2.1.: Num processo classificado no nível 2 (sem visibilidade), uma das suas fases pode ter uma actividade de teste bem definida. Para essa actividade faz sentido controlar o número de erros encontrados e usá-los como um dos possíveis indicadores da qualidade do produto. ⊗

No caso da controlabilidade também consideramos que, embora a aplicação do modelo de maturidade ao processo forneça uma indicação com significado acerca do grau global deste atributo, torna-se necessário, na sequência da sua aplicação, obter mais informação sobre se o controlo de cada fase é diferente. Para essas situações não existe nenhum modelo de avaliação estabelecido. A prática seguida é idêntica à utilizada para a observabilidade e já descrita na secção 3.4.1.

3.4.3.Estabilidade

A estabilidade do processo já não tem a ver com a visibilidade, mas sim com a sua capacidade de convergência para a solução.

Surpreendentemente, há processos complexos do primeiro nível do SEI [Pfleeger,N.90; Paulk91], que não dispõem de qualquer estrutura visível e que, devido a factores aleatórios, convergem rapidamente para uma solução.
Consideramos que, idealmente, um processo tem inicialmente uma fase muito criativa e como tal, pouco estável. Nessa fase, são investigadas várias hipótese de desenvolvimento, algumas das quais geram teorias (ou especificações). As especificações, ou são reificadas numa solução, ou são abandonadas.
Frequentemente, o número de hipóteses de desenvolvimento é elevado. Causas que se podem

apontar para este facto são o conhecimento incompleto do sistema a implementar e o reduzido número de pessoas necessárias a esta fase do processo.
Na segunda fase, o número de hipótese é mais reduzido. Em casos como as tecnologias emergentes, frequentemente há mais de uma hipótese a reificar.
Esta fase é menos criativa que a primeira, mas mais eficaz no sentido da aproximação da solução. Finalmente, numa terceira fase, uma única hipótese é desenvolvida e a velocidade da convergência para a solução aumenta.
Esta descrição é ideal e o processo pode não seguir as três fases sequencialmente. No mundo real acontece passar-se da fase de implementação da solução para a fase inicial de hipótese de desenvolvimento, porque se avaliou mal o risco de uma hipótese ou até foram introduzidas condicionantes no ambiente do problema pelo próprio desenvolvimento da solução proposta.
Exemplo 3.4.3.1.: O paradigma da espiral de Boehm, que usa a eliminação dos riscos de desenvolvimento da hipótese de trabalho, representa uma tentativa de controlo da estabilidade de convergência para a solução, uma vez que só se desenvolvem hipóteses e especificações sem risco e, como tal, mais estáveis. ⊗

Nas secção 3.5 é fornecida uma revisão das métricas usadas habitualmente na medição do produto, para evidenciar as dificuldades do seu uso no controlo dos processos de software e justificar a métrica alternativa que propomos.

3.5. Métricas do produto

O produto é medido em Engenharia do Software pelo seu tamanho *("size"),* que é caracterizado por três conceitos que representam os três atributos fundamentais que em conjunto o definem:

- Comprimento: refere o tamanho do produto;
- Funcionalidade: pretende capturar a quantidade de funções contidas num produto;
- Complexidade: diz respeito ao conjunto de recursos usados.

Em conjunto, os três atributos fundamentais fornecem uma compreensão do produto completa e, como tal, utilizável em avaliações de esforço, produtividade e custo para a sua obtenção. Em separado, o seu uso não fornece medidas com o grau de exactidão exigido pela indústria.

Actualmente, vários autores [Fenton,N97] juntam uma métrica de reutilização a estes três atributos, atendendo ao crescente do uso da metodologia de reutilização que influencia o custo do produto.

Os três atributos fundamentais foram apresentados por ordem crescente da elaboração do conceito preexistente. Consideramos que o comprimento é a métrica menos confiável para sozinha caracterizar o produto. Não existe consenso sobre se, em separado, a funcionalidade ou a complexidade caracterizem melhor o produto. Todavia, estes dois últimos tipos de métricas têm sido investigados por grupos, conjuntos de investigadores e utilizadores que introduzem continuamente melhorias, quer nas normas de aplicação, quer na validação dos resultados. Consideramos que a escolha entre funcionalidade e complexidade, se tiver de ser feita, deve atender aos objectivos da métrica e às condições existentes no projecto a medir. Nas secções seguintes detalhamos cada um dos referidos tipos de métrica.

3.5.1. Comprimento

Existem métricas de comprimento [Fenton,N.97;Conte,S.86] das especificações da arquitectura e da codificação.

As métricas em relação ao comprimento do código estão muito melhor definidas do que as métricas de comprimento da especificação e arquitectura.

Assim, para definir o comprimento de uma especificação ou arquitectura, ter-se-á de identificar, em função da metodologia usada, os objectos atómicos (primitivas) relevantes e, seguidamente, usar a sua contagem como comprimento. O significado que se pode obter desse valor dependerá da metodologia e terá de ser analisado para cada modelo.

Exemplo 3.5.1.1.: Por exemplo, se nos requisitos forem adoptados diagramas de fluxo de dados, os objectos atómicos utilizados são o processo, as entidades externas, os armazéns e os arcos de fluxo. Se nos requisitos se adoptar a linguagem de especificação formal Z, os objectos atómicos podem ser as linhas dos esquemas. ⊗

O comprimento de um programa pode ser medido em número de linhas de código. Existem várias definições de linhas de código, que são usadas de acordo com o objectivo da medição pretendida. Dependendo do objectivo de se pretender medir espaço de armazenamento do código, produtividade do programador ou dificuldade de teste, incluem-se, ou não, na contagem linhas em branco, linhas de comentários e declarações de dados, ou contam-se unicamente linhas executáveis [ConteS.86;Grady,R.92]. Havendo, portanto, vários tipos de

métrica de número de linhas de código, a sua definição deve ser determinada antes da sua utilização.

O comprimento de um programa foi também definido por Halstead [Halstead,M.77], usando quatro métricas diferentes: η_1 (número único de operadores de um programa), η_2 (número único de operandos de um programa), N_1 (número de ocorrências de operadores) e N_2 (número de ocorrências de operandos). O comprimento do programa é definido como $N=N_1+N_2$. São definidas outras métricas de comprimento; o vocabulário do programa $\eta = \eta_1+ \eta_2$ e o volume $V= N.\log_2\eta$.

A partir destas métricas, foram definidas várias outras. Actualmente, as métricas de comprimento têm pouca utilização e foram repetidamente questionadas na sua validade porque a sua formação não tornava clara a relação estabelecida no mundo real [Fenton,N.96;Coulter,N.83].

Existem outras alternativas muito semelhantes às descritas nos parágrafos anteriores como, por exemplo, o número de *Bytes* do ficheiro compilado do programa ou o número de caracteres no texto do programa. As seguintes limitações são comuns a todas as métricas de comprimento:

- capturam um único aspecto do produto, nem sempre o mais relevante para a gestão e melhoria do processo. Um programa longo não pode ser classificado com segurança como mais difícil de produzir que um outro mais curto com arquitectura mais complexa.
- são difíceis de aplicar quando se usam imagens de programação visual e ambientes gráficos baseados em janelas, onde os objectos pré-criados são usados na codificação, sendo o código escrito muito menor que o comprimento do código dos objectos utilizados.
- não têm em conta a reutilização de sub-rotinas, bibliotecas de funções e primitivas que aumentem a produtividade e a qualidade do código, permitindo ao programador utilizar o seu esforço na parte inovadora do problema.

A contagem da percentagem de reutilização incluída num programa ou módulo é difícil de ser linearmente relacionada com o esforço de programação. É diferente o esforço necessário para incluir um módulo sem modificação e um outro que tem de ser adaptado à nova realidade, embora tendo ambos o mesmo comprimento relativo à totalidade do comprimento do programa.

Existem classificações de níveis de reutilização usadas [SPC95], que classificam o código reutilizado conforme a sua percentagem de modificação. Na medição do comprimento do software, nestes casos, são evidenciados os comprimentos de código escrito bem como cada tipo de código reutilizado. Tal abordagem possibilita atribuir esforço diferente a cada um deles no cálculo do esforço total da escrita.

3.5.2. Funcionalidade

Existem várias abordagens para a obtenção da funcionalidade de um sistema [Fenton,N.96;Conte,S.86]:

Na metodologia de Albrecht's – Pontos de Funções – usa-se a especificação para obter a funcionalidade do produto. Esta é medida em Pontos de Função, que são calculados contando o número de itens de cada uma de cinco classes. Cada classe representa [Albrecht,A.79] um tipo diferente de funcionalidade.

Na metodologia de Boehm [Boehm,B.81], os Pontos Objecto representam também a funcionalidade e são contados de forma distinta, sendo contabilizados os números de écrans, relatórios e componentes de linguagens de terceira geração envolvidos na aplicação.

Na metodologia de DeMarco, a métrica é baseada na análise estruturada. A metodologia cria duas métricas chamadas métricas pesadas de especificação. Uma delas é obtida a partir da contagem dos níveis mais detalhados das bolhas e pelo número de saídas (*tokens*) que cada bolha usa. Esta métrica é chamada função "Bang". Outra métrica é obtida contando o número de entidades no modelo entidade-relacionamento, pesado pelo número de relacionamentos que cada entidade tem.

As métricas de funcionalidade apresentam vantagens em relação às métricas de comprimento:

- Medem um comprimento funcional, o que as torna muito mais precisas no cálculo de previsões de custo e esforço.
- Sendo calculadas durante a especificação, permitem avaliar no início do processo e o esforço necessário para completar o programa e são independentes da linguagem em que a solução é implementada.

Das três métricas referidas, o modelo de Boehm tem a vantagem de poder ser implementado automaticamente em ambientes *CASE* que suportem a análise estruturada.

Todas as métricas de funcionalidade apresentam a desvantagem de incluírem subjectividade na avaliação de pesos que usam para o cálculo.

Vários grupos de trabalho [Kirkman,G.02;INSEAD,01] construíram e analisaram bases de dados, usando as três metodologias de obtenção da funcionalidade como suporte da previsão de custos e tempo de entrega do produto. Progressos têm sido conseguidos pelos grupos de trabalho referidos. Todavia, a sua precisão não consegue ainda atingir as normas requeridas pela indústria [Kitchenham,B.98;Kitchenham,B.01]. A sua aplicabilidade é pouco eficaz em empresas com um nível baixo de definição de processo e uma prática deficiente de recolha de medidas.

As métricas de funcionalidade, contrariamente ao comprimento, já fazem uma abordagem mais abrangente dos atributos do produto e dão origem a modelos de previsão que têm de ser calibrados para cada instituição. Esses modelos apenas podem ser utilizados com sucesso por organizações muito estáveis, em termos ambientes de desenvolvimento e desempenho das equipas.
São métricas globais, e com uma formação mais próxima da perspectiva usada nesta dissertação para cálculo de uma métrica global do produto que possa ser calculada automaticamente e sirva para monitorizar o estado de convergência do processo para a solução. No entanto, as métricas da funcionalidade obrigam a que existam dados estatísticos fiáveis que fundamentam a calibragem da função em que se baseiam.

Consideramos como desvantagens destas métricas:

- implicitamente admitem que a empresa que tem o seu modelo calibrado mantém imutáveis as suas características de ambientes, de conhecimento e de gestão dos recursos humanos, situação que consideramos muito pouco provável, mesmo ao longo de um único processo.
- a utilização destas métricas não permite reconhecer que cada processo é um caso e que existem questões sensíveis ao longo do processo que podem alterar drasticamente o seu desfecho.

Pretendemos que as métrica que utilizamos nesta dissertação não revelem os inconvenientes das métricas de funcionalidade de exigência de calibração do processo e dependência de condições particulares do processo.

3.5.3. Complexidade em Engenharia de Software

A noção de complexidade na engenharia do software (CES) aparece na literatura com vários significados, alguns dos quais sobrepostos ao conceito de funcionalidade abordado na secção 3.5.2.. Em seguida apresenta-se uma lista das definições mais utilizadas:

- a complexidade computacional é um atributo do problema, que o produto modela e soluciona [Fenton,N.97; Wegner,P.95],
- a complexidade algorítmica mede a eficiência do algoritmo implementado para resolver o problema. Alguns autores, neste caso, chamam-lhe eficiência [Fenton,N.97; Harel, D.92],
- a complexidade estrutural mede o número e a estrutura das primitivas do algoritmo da solução [Fenton,N.97;Whihy,R.90],
- a complexidade cognitiva mede o esforço requerido para entender o problema ou a solução [Coulter,N.83].

Dos enunciados anteriores ressalta que existem noções ligadas, quer à complexidade do problema quer à complexidade da solução.
A complexidade do problema é diferente da complexidade da solução. São conhecidas soluções complicadas de problemas simples e o mesmo problema pode conduzir a soluções de diferentes complexidades.

Exemplo 3.5.3.1.: O problema de ordenação de um ficheiro é exemplo deste último caso. Os diferentes algoritmos existentes para esse fim têm, segundo qualquer um dos tipos de complexidade antes mencionados, valores bastante diferentes. Por exemplo, a eficiência do algoritmo de ordenação por inserção é menor que a eficiência do algoritmo de ordenação por acervo (*heap*), respectivamente N^2 e N log N em que N é a dimensão dos dados. Já na complexidade estrutural, existe uma relação inversa entre os algoritmos de ordenação por inserção e por acervo: o primeiro é executado com um ciclo dentro de outro ciclo, o segundo já exige a operação de restabelecimento da condição de acervo e eliminação de elementos de maior prioridade. ⊗

Segundo alguns autores [Fenton,N.97; Wegner,P.95] a complexidade do problema é o conjunto dos recursos necessários para implementar a solução óptima do problema. Naturalmente coloca se o problema de como definir óptimo. Não conhecemos qualquer regra para o efeito. No entanto, temos observado que, na prática, se optimiza localmente a solução em relação ao atributo medido.

Exemplo 3.5.3.2.: Por exemplo, se estivermos a medir a complexidade algorítmica que usa o atributo eficiência, o recurso optimizado será o tempo de execução, se o conceito usado for o da complexidade estrutural, a optimização será em relação ao tempo de implementação. ⊗

Passando às análise das definições encontradas para a complexidade da solução, não há unanimidade. Referenciamos duas definições conhecidas:

- A complexidade da solução "é a quantidade dos recursos necessários para implementar essa particular solução".
- A complexidade da solução "é o grau de complicação do sistema (modelo da solução), determinado por factores tais como o número e a *complicação* das interfaces, o número e a *complicação* das decisões condicionais, o grau de recorte (*indent*) dos tipos de estrutura de dados e outras características do sistema".

A primeira referência [Fenton,N.97; Conte,S.86,Zuse,98; Halstead,M.77; Hill,P.99; Weyuker, E.88], é uma definição operacional e que tem servido de base à maioria das construções das métricas de complexidade que enunciamos na secção 3.5.3.1.

Citamos a segunda [IEEE83], embora ambígua, visto ser uma norma do IEEE e fornecer a perspectiva da multiplicidade de utilizações que este conceito encerra. Dá-nos também a oportunidade para esclarecer que as noções de "complicação" e de "complexidade" não devem ser utilizadas enquanto tendo o mesmo significado. A primeira consiste numa característica de sistemas cujo entendimento pode ser melhorado através da decomposição (análise parcelar) dos seus elementos constituintes; enquanto que a segunda caracteriza os sistemas cuja natureza (complexa) provém fundamentalmente da interacção entre aqueles elementos.

Analisaremos resumidamente as métricas da complexidade (CES), evidenciando os aspectos da sua ligação ao mundo real, a fim de concluirmos (como já fizemos com a funcionalidade), que necessitamos de outro tipo de métricas para controlar o processo.

O algoritmo que lhe serve de modelo à solução é frequentemente utilizado para efectuar a métrica da complexidade. Existem várias estratégias para a obtenção desta métrica:

- Calculado a partir da chamada Eficiência do Algoritmo.
  É uma métrica que usa o número de operações primitivas relevantes para o desenvolvimento desse algoritmo, necessárias para produzir uma determinada saída e a notação "Big 0" [Fenton,N.97] para garantir a condição de representação da métrica [Zuse,H.91]. Considera-se que, quanto menos operações primitivas são necessárias para obter um resultado, menos complexo é o algoritmo.
- Calculado a partir da estrutura do algoritmo. Existem três tipos de métricas de estrutura [Witty,R.90;Conte,S.86]:

  - as que medem a complexidade da estrutura de controlo do algoritmo;
  - as que medem a complexidade do tratamento de dados pelo algoritmo;
  - as que medem a complexidade dos dados independentes do programa.

Para medir a complexidade da estrutura de controlo de um algoritmo, o mesmo é transformado num fluxograma [Fenton,N.97;Conte,S.86], a que está provado corresponder uma decomposição única em hierarquia de primitivas. Sobre essas árvores constroem-se métricas de complexidade que são chamadas métricas hierárquicas.
Exemplo 3.5.3.1.1.: São exemplos da medida a complexidade da estrutura de controlo de um algoritmo: o número de nós, o número de arcos, a métrica *k* da maior primitiva, a métrica *d* da estruturação. Outro exemplo muito conhecido é a métrica da complexidade ciclomática de McCabe, que representa o número de caminhos independentes que existem para percorrer o fluxograma. Várias outras métricas do tipo idêntico às primeiras foram propostas para este tipo de medição: a métrica de complexidade essencial [McCabe,J.89], a métrica de Vinap [Bach,R.90] e a métrica Knot [Woodward,M.93]. ⊗

É ainda usada outra aproximação para medir complexidade da estrutura. Alguns autores consideram que a estrutura de um módulo está relacionada com a dificuldade de o testar. Esse tipo de métricas [Bach,R.90a] medem o número de casos de teste e a eficiência do teste e consideram que estão a medir a complexidade.

Para medir complexidade de tratamento de dados pelos programas analisam-se as relações entre os módulos. Transforma-se uma arquitectura de um programa num *module call-graph,* que desenha a hierarquia dos módulos e indica a troca de informações entre eles [Fenton,N.97;Conte,S.86].
Para analisar a dependência de dados dentro dos módulos usam-se outros tipos de diagramas [Fenton,N.97;Conte,S.86;Bieman,N.90].
Exemplo 3.5.3.1.2.: São exemplos de métricas dependência de dados dentro dos módulos a modularidade global *("global modularity"),* que calcula a medida do comprimento dos módulos e que corresponde à ideia de que, quanto menor for o tamanho dos módulos, menor é a complexidade do produto e a impureza hierárquica ("*tree impurity"*), que traduz o afastamento de um dado módulo "*call-graph"* de uma árvore. ⊗

Esta métrica está suportada pela ideia de que, quanto mais um módulo "*call-graph"* se afasta de um árvore, mais complexo ele é.

São também usadas como métricas de complexidade métricas de acoplamento e de coesão entre módulos [Yourdan,E.79;Fenton,N.97;Conte,S.86], considerando que, arquitecturas com módulos muito acoplados e com fraca coesão interna são considerados complexos.

Constroem-se as denominadas métricas de informação medindo o nível total do fluxo de informação através do sistema e entre cada módulo com o restante sistema.
Exemplo 3.5.3.1.3.: São exemplos destas métricas: a complexidade do fluxo de informação *("information flow complexity"*), que usa o comprimento módulo, o seu "fan-in*"* e o seu "fan-out*"* de Henry e Kafura [Fenton,N.97]; a "shepperd complexity*"* [Swepperd, M.93], que mede o fluxo de informação de um módulo. ⊗

A ideia pré-existente será que, quanto mais fluxo de informação passa ou é gerado por um módulo, mais complexo ele se torna e mais tempo leva a ser desenvolvido.
Para medir a complexidade da estrutura de dados foram usadas métricas análogas às da estrutura dos algoritmos. Neste caso, cada tipo de dados é considerado uma primitiva e é medida estrutura de dados na hierarquia formada por essas primitivas [Elliot,J.88;Berg,K.95].
Zuse lista 187 métricas de complexidade [Zuse,H.91], dizendo que existem mais de mil. A

listagem que fizemos incluiu a caracterização dos tipos de métricas que apresenta.

3.6. Dificuldades no uso das métricas da CES para controlo do processo

Na secções 3.4 e 3.5 mostramos que cada métrica representa uma única qualidade ou perspectiva do produto ou do processo. Há inúmeros relatos do insucesso do seu uso, nomeadamente na precisão com que medem a realidade [Fenton,N.97;Kitchenham,B.98; Kitchenham,B.01]. Falta, no entanto, identificar as razões de tal insucesso. Consideramos que estas constatações são provocadas pelas seguintes dificuldades [Cardoso,A.01]

- Estas métricas são dependentes das linguagens e dos modelos utilizados para construir o produto que é medido.
  Como tal, as métricas de complexidade não permitem comparar programas escritos em linguagens diferentes, ou diferentes especificações e arquitecturas, as quais tenham sido definidas usando modelos distintos.

- Nas várias fases de implementação de uma solução, o software é representado por vários modelos diferentes.
  Requisitos, especificação, documentação, interfaces com o utilizador, código, são representados através de texto escrito, gráficos, linguagem simbólica ou iconográfica, linguagem formal, código binário, etc.. As métricas utilizadas para cada tipo de representação são diferentes. Assim, não existe uma métrica aplicável a todos os modelos, o que torna impossível avaliar a evolução de uma solução ao longo das várias fases do processo. Assim sendo, várias métricas, frequentemente incomparáveis e incompatíveis, têm de ser usadas para medir cada tipo de modelo da mesma solução durante o seu desenvolvimento.

- Constatamos ainda, que o resultado da aplicação de uma métrica da complexidade é, com frequência, um número com pouco significado qualitativo e do qual se desconhece a unidade de medida. Para a maioria dessas métricas também não estão definidos o que é um valor alto ou o que é um valor baixo de complexidade. Obter conclusões a partir de métricas com estas características é difícil. As actuais métricas de complexidade

permitem apenas comparar produtos medidos com a mesma métrica que foram obtidos por processos definidos com o paradigma, ambiente e equipa de desenvolvimento idênticos.

- Nova dificuldade surge quando se pretende estabelecer leis fundamentais a respeito do produto ou do processo e nos deparamos com a fragmentação das métricas existentes, cada uma dizendo respeito a um ou dois aspectos singulares produto ou processo. A necessidade de usar várias métricas para caracterizar o produto ou o processo globalmente e estabelecer as leis fundamentais que os referem é difícil, visto as métricas serem frequentemente contraditórias e obtidas em tipo de escalas diferentes.

São estas constatações que nos motivaram a tentar encontrar uma métrica calculável automaticamente sobre qualquer modelo intermédio da solução e que permitisse comparar produtos, independentemente das linguagens ou modelos usados para os construir.
No próximo capítulo analisaremos as chamadas métricas de complexidade, usadas no controlo de sistemas complexos, e estudaremos a sua aplicabilidade na engenharia do software. Usando uma métrica empregue na identificação, por exemplo, das áreas funcionais do genoma [Li,W.97], na atribuição de texto a autores [Schenkel,A.92], em estudos de doenças cardíacas [Harvard.ed.01], apresentaremos uma estratégia para controlar o processo, independentemente do seu modelo, e avaliar a sua convergência para a solução.

3.7. Síntese e contribuições do capítulo

Verificámos que as métricas existentes para medir o produto não respondem às necessidades do gestor do processo. Nomeadamente, sendo dependentes dos modelos usados para construir o produto, é difícil atribuir um significado aos valores que tomam e caracterizam apenas aspectos particulares do produto.

A contribuição deste capítulo para a hipótese de investigação é a justificação da impossibilidade das métricas actualmente usadas permitirem uma avaliação global do sistema. Em consequência, as métricas não permitem a verificação quantitativa das leis gerais de desenvolvimento software.

4. Os sistemas complexos

> ***...**As novas Ciências do caos e da complexidade fornecem metáforas valiosas e métodos que podem desafiar a pesquisa em gestão no próximo (actual) século...(com)... as imagens da auto organização, estruturas dissipativas e complexidade dinâmica....*
>
> *Overman*

4.1. Introdução

O objectivo deste capítulo é apresentar as propriedades fundamentais dos sistemas complexos, algumas medidas da complexidade, ou a esta associadas. Desta formam, o capítulo estabelece o enquadramento necessário à apresentação das hipóteses formuladas nesta dissertação.

4.2. Propriedades e questões importantes na área dos sistemas complexos

Os exemplos de sistemas complexos encontram-se por toda a parte: mercados financeiros, ecologias, redes de regulação genética, sociedades, sistemas de computação distribuída, a internet, etc.

Naturalmente, não é fácil definir o que é a Complexidade [Vilela Mendes,R.99]. No entanto, uma série de propriedades comuns à generalidade dos sistemas complexos pode ser identificada. A característica mais frequentemente encontrada em todos estes sistemas é a de serem, em geral, compostos por um grande número de elementos e o seu comportamento apresentar propriedades que não podem ser facilmente deduzidas a partir das propriedades do comportamento dos elementos quando isolados. A emergência de propriedades colectivas, qualitativamente diferentes das individuais e a não-linearidade das interacções entre os elementos são as características fundamentais do comportamento dos sistemas complexos.

A decomposição dum sistema nos seus elementos, para melhor o compreender, foi durante muito tempo o grande método de estudo das ciências tradicionais. Porém, se o sistema for um

sistema complexo, as suas propriedades não são decomponíveis.
A essência dos sistemas complexos reside na natureza e efeito das interacções e nas propriedades colectivas que elas criam. Daí que, cada vez mais, haja na comunidade científica um esforço para tentar desenvolver uma teoria unificada dos sistemas complexos, através dum estudo global e comparativo.

As questões mais importantes na área dos sistemas complexos são:

Relação forma-função: Em que medida a estrutura particular dum sistema influencia a sua funcionalidade e quão vasto é o leque de estruturas que correspondem à mesma função?

Estruturas colectivas emergentes: Que estruturas colectivas emergem nestes sistemas e como elas se relacionam com os tipos de interacção entre os elementos do sistema?

Dinâmica evolutiva: Que tipos de dinâmica colectiva são característicos destes sistemas? Como é que a dinâmica colectiva se relaciona com as dinâmicas individuais?

Quantificação da complexidade e sua evolução: Como é que se pode quantificar a complexidade dum sistema e como é que este tende a evoluir, para uma maior ou menor complexidade? E em que circunstâncias?

De uma maneira geral pode-se dizer que o estudo dos sistemas complexos segue duas vias complementares, cada uma delas com objectivos distintos. A primeira via está direccionada para a melhoria da capacidade de prever o comportamentos dos sistemas complexos, enquanto a segunda via está associada ao estudo da criação de estruturas. É esta segunda via, que exploramos nesta dissertação.

Quer as estruturas, quer quaisquer outros aspectos investigados na área dos sistemas complexos envolvem, directa ou indirectamente pelo menos três propriedades fundamentais: a não linearidade, a interdependência e a emergência.

Não-linearidade significa que o comportamento dinâmico do sistema não pode ser visto como

uma simples sobreposição dos comportamentos elementares dos seus constituintes.

Interdependência significa que a reacção de cada um dos elementos do sistema depende fortemente do comportamento dos outros, dum modo auto-consistente e difícil de prever *a priori*.

Emergência caracteriza o aparecimento de propriedades colectivas, qualitativamente diferentes do comportamento individual. A emergência, ou criação, de estruturas é uma das propriedades mais frequentemente estudadas na área da Complexidade.

Definição 4.2.1.1: um sistema complexo é um sistema formado por um elevado número de elementos, caracterizado por uma dinâmica não-linear, por interdependência entre o comportamento dos seus elementos e pelo aparecimento de propriedades colectivas, qualitativamente diferentes do comportamento individual. A essência dos sistemas complexos reside na natureza e no efeito das interacções entre os seus elementos. ⊗

Exemplo 4.2.1.1. A minha família é exemplo de sistema complexo. É formada por um número elevado de elementos, perto de cem. Tem uma dinâmica *não linear* visto que o que acontece a um dos elementos não tem apenas influência sobre ele mas afecta fortemente todos os outros e altera inclusivamente a evolução da família. A morte ou doença grave de um dos elementos leva a que negócios familiares possam não ser realizados, porque a *interdependência* afectiva entre os comportamentos dos seus elementos é muito forte. É ainda possível identificá-la por *propriedades colectivas* qualitativamente *diferentes* do comportamento individual. Uma atitude optimista como forma de resolver grandes dificuldades, é uma característica *emergente* da minha família, proveniente da certeza da inter ajuda existente e coexistente com a atitude individual pessimista de vários membros. ⊗

A designação de sistema como complexo provém da dificuldade de obter, reunir e organizar informação suficiente para a sua representação.

Nas secções que se seguem apresentamos algumas medidas da complexidade. Dado o âmbito do estudo desta dissertação, as medidas apresentadas estão, em parte e sempre que possível,

restritas à quantificação da complexidade em sistemas estáticos, ou seja, não se pretende explorar as medidas cuja funcionalidade recai sobre a caracterização da dinâmica dos sistemas complexos.

Várias das definições de complexidade aparecem associadas à quantidade de informação necessária à descrição, à reconstrução (capacidade de previsão) ou à classificação dos sistemas complexos.

De acordo com a referência de entre as principais quantidades propostas para caracterizar os diversos aspectos dos sistemas complexos *i)* umas caracterizam a maior ou menor natureza aleatória dos sistemas, *ii)* outras caracterizam as dificuldades de reprodução do sistema e *iii)* outras ainda caracterizam as relações entre o sistema e os seus elementos constituintes [Vilela Mendes,R99]. Estas últimas são as medidas de auto-organização.

Podemos ainda separar as medidas de auto-organização e as de auto-organização dinâmica. As primeiras incidem sobre a caracterização da criação de estruturas, as segundas incidem sobre a quantificação das relações da dinâmica colectiva do sistema com as dinâmicas individuais (dinâmica evolutiva).

Dado o objectivo central desta dissertação, a medida da complexidade do software (ou do produto resultante do processo de desenvolvimento de software), interessa principalmente explorar as medidas da complexidade vocacionadas para a caracterização da regularidade dos sistemas complexos. Esta é também uma das maneiras de caracterizar (através da quantificação) a existência de estrutura num determinado sistema. São as seguintes as medidas que, mais frequentemente, servem de instrumento a esta caracterização:

- medidas da complexidade
    - computacional,
    - bruta,
    - algorítmica,
    - efectiva,

- medidas de entropia
    - de Shannon
    - de Kolmogorov-Sinai e os expoentes de Lyaponov,
    - de Renyi

- leis de escalamento e potência
    - a lei de Zipf

- medidas de correlação

4.3. Medidas de complexidade

As definições de complexidade seguintes aparecem associadas à quantidade de informação necessária à descrição do sistema em estudo. Cada uma das definições usa uma estratégia distinta para quantificar essa quantidade de informação.

4.3.1.Complexidade computacional

Definição 4.3.1.1.: A complexidade computacional [Edmonds,B.00;Balcázar,J.88;Bőrger, E89] de um problema é medida pelo tempo mínimo, necessário para encontrar, num computador universal, uma solução para esse problema. ⊗

Um computador universal é um computador geral, idealizado e padrão com capacidade infinita e dispondo de todo o tipo de hardware e software.

O tempo mínimo para encontrar a solução e o computador universal, são utilizados nesta definição de complexidade para garantir que esta medida não depende nem do conhecimento que o programador dispõe, nem da máquina utilizada para encontrar a solução.

A complexidade computacional é uma medida difícil de obter, visto não podermos ter a certeza de que o algoritmo que dispomos para encontrar a solução é o que corresponde ao tempo mínimo. A não ser que seja formalmente provado, poderá sempre surgir um outro algoritmo, no futuro, que seja mais rápido.
A complexidade computacional não é, também, uma medida que traduz a noção intuitiva de

complexidade, uma vez que não se refere a tempo de descrição de propriedades do problema, mas antes a tempos de execução das funcionalidades da solução do problema. O tempo pode ser longo porque se repete a execução do igual (compactável na descrição).
O programa que implementa a solução pode demorar porque, por exemplo, repete a fixação dos resultados no écran e não porque esteja a implementar funcionalidades várias do sistema. Esta última situação é intuitivamente muito mais complexa.
Este tipo de medida é útil para analisar o que acontece com o tempo de execução quando o tamanho de um problema (da sua descrição) aumenta (vai sendo acrescentado).
A complexidade computacional é, pois, uma medida utilizada na análise da dinâmica da evolução das soluções e não a utilizaremos neste trabalho.

4.3.2.Complexidade bruta

Definição 4.3.2.1.: A complexidade bruta [Gel-Mann,M.85] de um sistema é o comprimento da mais pequena mensagem usada por um observador para descrever o sistema, com uma dada precisão e utilizando uma dada linguagem, a um outro observador com quem compartilhe igual conhecimento do objecto descrito e da linguagem da descrição. ⊗

Esta medida da complexidade, baseada no tamanho, é também difícil de obter, por ser igualmente impossível demonstrar que uma mensagem é a mais curta que descreve o sistema. Esta medida é ainda dependente da escala da descrição, dos observadores e da linguagem utilizada na descrição. Só tem utilidade quando usada tendo em conta estas limitações.
Não a utilizaremos, por isso, neste trabalho.

4.3.3.Complexidade algorítmica e conteúdo algorítmico da informação

Andrey Kolmogorof e Gregory Chaitin em 1965 [Kolmogorov,A.58;Chaitin,G.66] introduziram simultaneamente o conceito de conteúdo algorítmico de informação, também chamado de complexidade algorítmica. Para calcularem essa grandeza obtêm a descrição de um sistema com uma determinada precisão, exprimem essa descrição numa linguagem que é seguidamente codificada dando origem a uma sequência de zeros e uns. O conteúdo algorítmico de informação do sistema é medido a partir dessa sequência.

Definição 4.3.3.1.: O conteúdo algorítmico de informação (AIC) de uma sequência de bits é o comprimento do mais pequeno programa de uma máquina universal que imprima a sequência e pare. ⊗

Para a medida AIC, nunca podemos ter a certeza de temos obtido o valor correcto. Podemos, quando muito, dizer que o valor que encontramos para um objecto real é o limite superior dessa medida. Nada nos garante também que possamos encontrar um outro algoritmo, que encontre mais regularidade, ainda escondida na sequência e dela faça uma descrição mais concisa.

Também esta medida não traduz o conceito intuitivo de complexidade. Uma sequência aleatória exige a listagem explícita de todos os elementos a imprimir. A sequência de números pares é gerada pelo programa de um contador, que é incrementado de 2. O primeiro programa é mais complexo (a sua descrição é maior) do que o segundo. Não é uma medida operacional porque nunca podemos saber o seu valor exacto e não é indicadora de complexidade mas sim de aleatoriedade da sequência, uma vez que é máxima para uma sequência aleatória. Não utilizaremos esta medida visto pretendermos quantificar a existência da regularidade.

4.3.4.Complexidade efectiva

Outra medida de complexidade, que é mais coincidente com o conceito intuitivo de complexidade, é a complexidade efectiva. Neste caso, em vez de procurar, como com AIC, o comprimento da mais concisa descrição da entidade, procuramos o comprimento da mais concisa descrição do conjunto das regularidades da entidade. Tal abordagem conduz a que uma sequência aleatória, porque não contém regularidades, e uma sequência totalmente regular tenham ambas complexidade efectiva próxima de zero.

Definição 4.3.1.4.1.: A complexidade efectiva [Gel-Mann,M.85;Edmonds,B.00] é o comprimento da mais concisa descrição do conjunto das regularidades do mais pequeno programa do computador universal que imprima a sequência e pare. ⊗

A complexidade efectiva é uma medida muito mais próxima do conceito intuitivo de complexidade do que as anteriormente descritas. Quer a complexidade bruta, quer o conteúdo algorítmico de informação medem o comprimento de uma descrição concisa do sistema, tendo em conta os aspectos aleatórios (incompressíveis) e os aspectos regulares (compressíveis) do mesmo. Como as restantes medidas de complexidade que apresentámos esta não é uma medida operacional porque nunca podemos saber o seu valor exacto. É

construída sobre uma descrição que é no momento a mais concisa, mas que pode ser substituída por outra, encontrada depois. Por consequência, não utilizamos nesta dissertação a medida de complexidade efectiva.

Na tabela que Tabela 4.3.4.1 são resumidas as quatro medidas diferentes de complexidade.

Tabela 4.3.1.4.1 Algumas Medidas de Complexidade

Nome	**Complexidade computacional**	**Complexidade bruta**	**AIC**	**Complexidade efectiva**
Conceito usado	Tempo	Comprimento	Comprimento	Comprimento
Elementos definidores	1. Máq. Universal 2. Tempo mínimo para encontrar a solução	1.Máq. Universal 2. Mensagem usada para descrever o sistema 3. Comprimento dessa mensagem	1. Máq. Universal 2. Mensagem usada para descrever o sistema 3. Programa mínimo que imprime a mensagem	1. Classes de regularidades do sistema 2. Descrição das classes
O que mede	Tempo de execução	Concisão da descrição	Concisão do programa que reproduz a descrição	Regularidades do sistema

4.4. Medidas de entropia

Dado que várias das definições de complexidade aparecem associadas à quantidade de informação necessária à descrição do sistema em estudo, então é natural que na caracterização dos sistemas complexos também se empreguem medidas associadas à quantificação da incerteza ou da falta de informação. É o caso das medidas de entropia que descrevemos nesta secção.

No capítulo 5 chega-se à conclusão que as medidas de entropia não são suficientes para a caracterização da complexidade de programas escritos em C. No mesmo capítulo são igualmente discutidas as possíveis causas de tal resultado.

4.4.1.Entropia de Shannon.

Segundo Shannon, [Shannon,C.49] a distribuição de um conjunto de símbolos pertencentes a um alfabeto *s(i)* de tamanho n, num determinado texto {*s(1),s(2),....s(k)*}, pode ser caracterizada por uma função chamada entropia, designada por *H* e definida da seguinte forma:

$$H = -\sum_{i=1}^{n} p(i) \log p(i)$$

sendo *p(i)* a probabilidade de cada símbolo *s(i)* do alfabeto ocorrer no texto.

A entropia de Shannon mede a falta de conhecimento do observador a respeito de que símbolo pode ser esperado pertencer a um determinado texto quando só conhece a distribuição de probabilidade dos símbolos que podem ser usados para escrever o texto.

Quando os símbolos de um texto são escolhidos aleatoriamente podemos considerar que cada ocorrência é independente das outras e todas as ocorrências tem probabilidades iguais a 1/*n*. Esse texto, conhecido como texto de Bernoulli, tem uma entropia igual a - *log (n)*. No outro extremo encontra-se o caso em que para um dado símbolo *i* se tem *p(i)* =1 e *p(j)* =0 para $j \neq i$, resultando então que *H* é igual a zero.

A Entropia de Shannon dá-nos portanto uma medida da uniformidade ou da homogeneidade de uma distribuição de probabilidades, uma vez que, quanto mais próximo de zero estiver o valor da entropia, menor é a uniformidade de referida distribuição.

4.4.2.Entropia de Rényi

Uma extensão da entropia de Shannon é a entropia de Rényi [Beck,C.93].

A entropia de Rényi toma o valor da entropia de Shannon para α igual a 1. Para α diferente de 1 calcula-se a partir da fórmula seguinte:

$$Hr = \frac{1}{1-\alpha} \log \sum_{i=1}^{n} p(i)^{\alpha}$$

O uso de uma potência na probabilidade permite valorizar a influência, quer das

probabilidades grandes, quer das probabilidades pequenas na distribuição de probabilidade que se analisa.

Se o α >1 as probabilidades altas da distribuição são valorizadas em relação às baixas. Se α<1 então são valorizadas as probabilidades baixas da distribuição.

4.4.3.Entropia de Kolmogorov-Sinai e os expoentes de Lyapunov

Para além das medidas de entropia de Shannon e de Rény existem outras medidas que, ainda que também não tenham sido definidas com o propósito de medir a complexidade, permitem pôr em evidência a descoberta de padrões ou de relações persistentes entre os elementos de um sistema. Já aqui se afirmou que a essência dos sistemas complexos reside na natureza (e efeito) das relações (e interacções) entre os seus elementos; também se viu que várias das medidas da complexidade recorrem à descoberta de regularidades presentes no sistema. Assim sendo interessa completar a apresentação das medidas de complexidade e de incerteza com a inclusão de um instrumento indicador de existência de relações persistentes nos sistemas em observação, ou seja a sua complexidade dinâmica.

A dinâmica dos sistemas complexos exibe dependência sensível às condições iniciais. Esta propriedade caracteriza o facto que uma pequena variação no ponto inicial do sistema causar uma grande variação ao longo da trajectória do sistema. O maior dos expoentes de Lyapunov de um sistema, $\lambda_{1,}$ é a medida desta sensibilidade.

Definição 4.3.2.3.1. O expoente de Lyaponov define-se como sendo a média da taxa de crescimento exponencial da variação inicial, ao longo de uma trajectória do sistema, no limite das pequenas variações [Vilela Mendes,R.99]. ⊗

$$\lambda_{1=} \lim_{\Delta(0)\to 0 t\to\infty} \frac{1}{t}\log\frac{\Delta(t)}{\Delta(0)}$$

Injectando-se na condição inicial uma pequena perturbação $\Delta(0),$ analisa-se como essa perturbação evolui ao longo do tempo.

O expoente de Lyaponov, tendo sido definido com média ao longo da trajectória, tem o seu valor dependente da taxa de expansão do sistema e das regiões onde a trajectória permanece

mais tempo. Se houver taxas de variação de expansão distintas nas várias direcções, torna-se mais difícil reconstruir o sistema inicial.

Para uma perturbação genérica $\Delta(0)$ das condições iniciais, o limite da equação definidora de λ_1 é a maior taxa de expansão. Porém em cada ponto da trajectória, em vez da expansão numa direcção apenas, poderíamos ter considerado a matriz completa da expansão em todas as direcções. Tomando o valor médio do produto dessas matrizes ao longo da trajectória e diagonalizando a matriz resultante obtém-se um espectro de expoentes de Lyaponov.

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \ldots$$

O sistema diz-se caótico quando pelo menos um destes expoentes é positivo, isto é quando os erros das condições iniciais são amplificados ao longo do tempo.

O expoente de Lyaponov tem sido adoptado para modelar, por exemplo, a dinâmica de fluidos em turbulência e mercados financeiros.

A entropia de Kolmogarov-Sinai está também ligada à estrutura de probabilidades do sistema dinâmico. É definida a partir da construção de uma partição do espaço de estados do sistema e da probabilidade de permanência de uma trajectória do sistema em cada conjunto da partição [Vilela Mendes,R.99].

A entropia de Kolmogarov-Sinai mede o valor médio da taxa de produção de informação do sistema por unidade de tempo.

No caso de termos um sistema sensível às condições iniciais, é de esperar que a maior parte das trajectórias irão rapidamente divergir umas das outras. Nesse caso existe uma relação quantitativa entre a entropia de Kolmogarov-Sinai, *h*, e os expoentes positivos de Lyaponov dada pela seguinte fórmula:

$$h \leq \sum_{\lambda \geq 0} \lambda_i$$

Uma vez que a entropia de Kolmogarov-Sinai e os expoentes de Lyapunov são medidas da dinâmica. Uma vez que queremos medir também a parte estática da estrutura elas não são utilizadas nesta dissertação.

4.5. Leis de escalonamento ou leis de potência

O conceito de lei de potência é invocado com frequência no estudo dos sistemas complexos. É com base neste conceito, cuja aplicação pressupõe a identificação de um expoente característico, i.e. igual em todas as escalas, em medidas do sistema, que se procura assemelhar um conjunto de fenómenos observados em áreas do conhecimento tão distintas como as ciências naturais e as ciências sociais.

O conceito de lei de potência é contra-intuitivo devido à falta de uma escala característica, i.é. única. Os exemplos mais notórios da sua aplicação vão da Economia à Biologia, passando pela Matemática onde serve de base à definição de uma das entidades mais emblemáticas dos sistemas complexos: a dimensão fractal.

Os sistemas complexos apresentam em geral uma aparência aleatória aquando da observação de algumas das suas propriedades e que, por isto mesmo, uma das formas de caracterizar a complexidade de um sistema é quantificar a sua menor natureza aleatória, separando o que é aparente daquilo que é real. Uma das ferramentas estatísticas mais utilizadas na caracterização empírica de um conjunto de dados (de uma série, por exemplo, temporal) é o estudo da sua distribuição.

4.5.1.Leis de escalonamentos em diversos domínios

Há 100 anos o economista italiano W. Pareto investigou a relação entre os rendimentos individuais dos elementos de uma economia e observou que estes obedecem a uma distribuição em lei de potência:

$$y(x) \sim x^{-\lambda}$$

onde *y(x)* corresponde ao número de elementos com rendimento igual ou superior a *x* e λ toma o valor 1.5.

Ainda no âmbito da Economia mas com muitos anos de distâncias das observações de Pareto, decorrem inúmeros trabalhos de investigação que dão conta de leis de potência em distribuições encontradas em sistemas tão complexos como o Mercado Financeiro

[Gabaix,X.03; Gabaix,X.03a; Matia,K.03;Stanley H. E.,02; Mossa,S.02; Stanley H. E.,02; Plerou, V.00; Gopikrishnan, P.99;Stanley H. E.,99].
Mais ainda, é com base nesta propriedade empírica que os modelos de sistemas complexos têm sido validados em Economia.

É de especial interesse, dado o âmbito desta dissertação, que os trabalhos dos autores acima referidos também incluam a verificação da ocorrência de leis de escalonamento em sistemas doutra natureza, em Biologia, nas Ciências Médicas e nas Linguagens naturais.

Em Biologia, sabe-se que a probabilidade de encontrar determinadas sequências em regiões não codificadas do DNA segue uma lei de potências.
Eugene Stanley e a sua equipa de investigação do departamento de biologia do MIT, evidenciaram que as regiões codificadas e não codificadas do DNA apresentavam propriedades estatísticas diferentes, [StanleyH.E.,00; Havlin,S.95; StanleyH.E.,94; Peng,C.K.94 ;Mantegna,R.N.95; Peng,C.K.93;Buldyrev,S.V.93; Peng, C.K.92;].
Usaram a representação simbólica do DNA como uma sequência de um alfabeto que representa os quatro elementos básicos existentes no DNA: A (adenina), C (citosina), T (tiamina) e G (guamina) [Buldyrev,S.V.95].
Os investigadores verificaram que se dividissem a sequência em grupo de seis símbolos, e obtivessem a frequência de cada grupo encontrado podiam ordenar os grupos por ordem crescente das frequências encontradas.
Assim cada grupo diferente g_i de seis símbolos teria uma ordem n e uma frequência f_i.
Encontraram, nas regiões não codificadas do DNA, as sequências de tamanho 6 obedecem à seguinte lei de potência.

$$f_i = n^{-0,34}$$

Para as regiões codificadas do DNA essa lei de potência não é observada.

Estão também descritas distribuições em lei de potências para descrição de fenómenos fisiológicos humanos [Schulte-Frohlinde, V.02; Ashkenazy Y.01].
Por exemplo, os intervalos entre respiração de crianças com 39 semanas obedece também à

distribuição de potência, onde *y(x)* corresponde à frequência dos intervalos com um determinado tamanho *x* e λ é igual a 2,07.

Quando se trata de crianças de 61 semanas a lei de potência é igualmente observada sendo que λ é igual a 3,55. O aumento de λ, de 2,07 para 3,55, caracteriza a maturidade dos pulmões [Suki, B.03].

Também os cardiologistas motivados pela necessidade de prever o risco de morte súbita por fibrilhação ventricular (frequência não estacionária da contracção do ventrículo) procuraram encontrar regularidade estatística na distribuição dos batimentos cardíacos.

F(n), definida como a raiz quadrada das flutuações dos batimentos cardíacos durante *n* batimentos cardíacos obedece à seguinte lei de potência:

$F(n)/n = n^{0,74}$ para indivíduos saudáveis

$F(n)/n = n^{0,5}$ para indivíduos doentes

O expoente da distribuição caracteriza mais uma vez, a complexidade do sistema e toma valores perto de 0,5 quando o sistema está a degradar-se [Ashkenazy Y.01].

No que se refere ao estudo das linguagens naturais, usando o método desenvolvido por Peng [Peng,C.92] Schenkel analisou vários textos literários. Em todos eles encontrou a lei de potência, em que λ toma um valor característico diferente do encontrado num texto gerado aleatoriamente. A secção 4.6.2. explicaremos o método e indicaremos os argumentos da função no caso da escrita humana.

No que se refere à caracterização da Internet, Albert-László Barabási [Barabási,A.00; Yang,H.03] descreve a WWW como um grafo cujos vértices são os documentos e os arcos são as ligações *(links)* entre dois documentos. Considerou que a Web crescia continuamente e demonstrou que a probabilidade *P(k)* de um documento na Web estar ligado a *k* documentos segue uma lei de potência:

$P(k)=k^{-2,9}$

Uma vez que esta probabilidade é independente do tempo este resultado realça que mesmo sendo a Internet uma estrutura em constante crescimento, existe um expoente que a caracteriza e esse expoente é independente da escala.

Também S. Dorogovtsev e J.F.F. Mendes [Dorogovtsev,S.00] desenvolveram estudos nesta área. Na análise que desenvolveram definiram a probabilidade de um documento ser retirado da Web e de novas ligações serem incluídas em documentos antigos.

Demonstraram que nesse caso probabilidade *P(k)* de um documento na Web apontar *k* documentos e a conectividade média ao longo do tempo *t* de um site com uma determinada idade s, $\bar{k}\,(s,t)$, seguem leis de potência.

$$P(k)=k^{-\lambda}$$

$$\bar{k}\,(s,t)=(s/t)^{-\beta}$$

onde λ e β dependem apenas da constante c que representa o numero de ligações adicionadas em cada instante.

No caso em que $c \geq 1$, β toma o valor de $\frac{(1+2c)}{2(1+c)}$ e λ toma o valor de $2+\frac{1}{1+2c}$.

De volta às leis de potência em Biologia, em 1992 o investigador C-K.Peng [Peng,C.92] introduz o conceito de DNA-walk e mais tarde o método conhecido por Detrended Fluctuations Analysis (DFA) [Peng,C.K.94], o qual tem sido frequentemente empregue na caracterização dos já referidos sistemas económicos. Voltaremos a este tópico (DNA) na secção 4.6.2. ao apresentarmos a utilização das medidas de correlação na caracterização da complexidade.

4.5.2.A lei de Zipf

A Lei de Zipf estabelece uma relação entre a ordenação dos elementos de um sistema segundo um determinado critério e os valores de uma qualquer grandeza medida (quantificada) para os elementos deste sistema. A relação estabelecida tem a forma de uma lei de potência, ou seja, se *i* é o número de ordem de cada elemento e G(*i*) é a quantidade da grandeza de *i*, então:

$$G(i) = i^{-\lambda}$$

onde λ é o expoente característico.
Um dos exemplos mais conhecidos de aplicação da Lei de Zipf consiste naquele que estabelece uma relação entre o número de ordem da frequência de ocorrência das palavras mais comuns num dado texto em inglês, dispostas por ordem da frequência mais alta de ocorrência e a frequência de ocorrência destas palavras no texto. É conhecido que neste caso as frequências de ocorrência são sucessivamente proporcionais a 1, 1/2, 1/3, 1/4 ..., ou seja, o expoente característico λ, tem valor igual a um.

Leis deste tipo são também conhecidas por definidoras da hierarquias ou organização em escalões.

Descrevemos na secção 5.4.3 como a lei de Zift é seguida nos vários programas da amostra que analisámos. Veremos que basta um programa deter apenas correcção sintáctica, para esta lei ser verificada.
No entanto, a lei de Zipf não é suficiente para discriminar estruturas funcionais.

4.6. Medidas de correlação

Frequentemente, as séries temporais repetem padrões ou exibem propriedades que demonstram que valores ou sequências de valores observados numa parte da série têm relação com valores ou sequências de valores surgidos noutras partes. Esta situação indica a presença de estrutura na série, uma vez que a repetição de padrões indica possibilidade de compressão do modelo usado na descrição do fenómeno.

4.6.1.Autocorrelação

A autocorrelação ou a correlação de série dá-nos a medida do grau de existência dessa repetição e, nesse sentido, é uma medida indicativa da existência de estrutura quando a autocorrelação é alta e de irregularidades na estrutura quando a autocorrelação é baixa.
A autocorrelação é determinada a partir da autocovariância e da variância de uma série. A autocovariância tem o significado literal: indica como uma série varia em relação a si própria.

Definição 5.2.3.2.1.: Considerando a série $x_1 \ldots x_n$ e a média dos seus valores $\bar{x}$, para um intervalo *m* entre os valores considerados da série, a autocovariância é definida por:

$$\mathrm{cov}(x_1 \cdots x_n) = \frac{1}{n}\left[\sum_{t=1}^{n-m}(x_t - \bar{x})(x_{t+m} - \bar{x})\right]$$

a variância é definida por:

$$\mathrm{var}(x_1 \cdots x_n) = \frac{1}{n}\left[\sum_{t=1}^{n}(x_t - \bar{x})^2\right]$$

Normalizando autocovariância com a variância da série, tem-se a expressão da autocorrelação:

$$\text{Autocorrelação } (x_1 \ldots x_n) = \left[\sum_{t=1}^{n-m}(x_t - \bar{x})(x_{t+m} - \bar{x})\right] \Big/ \left[\sum_{t=1}^{n}(x_t - \bar{x})^2\right] \qquad \otimes$$

Na equação obtém-se um valor da variação da série temporal em relação a si mesma. Como foi normalizado, o valor pode ser comparado com a autovariação de outras séries. Esta variável normalizada é chamada autocorrelação e toma valores entre -1 e 1.

- O valor de 1 indica que os dois segmentos que comparamos são exactamente iguais e o produto das diferenças tomadas coincide com a variância da série. Considera-se assim que existe correlação entre os valores da série.

- O valor -1 é obtido quando a autovariância apresentar todos os seus factores negativos, ou seja, as variações calculadas em cada parte da série têm sempre o sinal contrário e são idênticas. Considera-se assim que existe anti-correlação entre os valores da série.

- O valor 0 indica que a autovariação é nula e, como tal, não existir qualquer correlação entre os valores considerados.

4.6.2.Correlação de gama longa

A correlação de gama longa é muitas vezes, mencionada como correlação estrutural, uma vez que não tem uma escala de comprimento característica. Teoricamente é invariante para todas as escalas de todos os níveis da estrutura do objecto em estudo. Indica, pois, um valor único que caracteriza a estrutura. Na realidade existem limites, embora altos, onde a correlação de gama longa desaparece. Conforme é observado na secção 5.4., os limites não afectam os resultados, quando a amostra é suficientemente grande.

A correlação de gama longa pode ser detectada obtendo a função de autocorrelação entre dois pontos de uma sequência para os diferentes afastamentos dos mesmos e examinando se seguem uma lei de potência. Quando tal se verifica, diz-se que não existe uma escala característica para a correlação.

A correlação de gama longa representa um mecanismo de "memória". Um valor ou conjunto de valores da sequência estão relacionados, não só com os que imediatamente os precedem, mas com outras flutuações de valores afastados na sequência. O "longo" do conceito de correlação de gama longa significa que a correlação existe também entre estruturas afastadas. Assim o valor que determinamos depende não só dos valores da sua vizinhança, mas depende também dos valores afastados (observados a distâncias longas) da série.

Tal como na lei de Zipf, veremos na secção 5.4.4. que a auto correlação, por si só, não é suficiente para discriminar as estruturas funcionais.

Aplicação em DNA

C. K. Peng [Peng,C.92] propôs um método para estudar as propriedades estatísticas da sequência dos nucleotidos do DNA. A sua motivação era encontrar diferentes propriedades estatísticas nas sequências codificadas (correspondentes a uma função) e nas sequências não codificadas do DNA.
Usou a representação simbólica do DNA como uma sequência de um alfabeto dos quatro elementos básicos nele existentes: A (adenina), C (citosina), T (tiamina) e G (guamina). A e G são purinas, C e T são pirimidinas.

Especificou regras [Buldyrev,S.V.95] para mapear esta sequência numa outra sequência numérica $\{u_i\}$ em que u_i tomava o valor 1 ou –1. Por exemplo se era encontrada uma purina u_i tomava o valor 1, se era encontrada uma pirimidina u_i tomava o valor -1.

Para quantificar a correlação existente em $\{u_i\}$ construiu uma representação gráfica - o denominado passeio aleatório unidimensional conhecido, neste caso, como passeio DNA (*"DNA walk")*.

Um passeio aleatório unidimensional é caracterizado pelo valor do quadrado das suas flutuações, $F^2(l)$, que se calcula da seguinte forma:

Seja $f(l)$ a deslocação no DNA walk depois de l movimentos

$$f(l) = \sum_{i=1}^{l} u(i)$$

A diferença de posição entre um deslocamento l_0 e outro à distância l é determinadas por:

$$d_l = f(l_0 + l) - f(l_0)$$

A média do quadrado das flutuações é designada por $F^2(l)$

$$F^2(l) = d_l^2 - (d_l)^2$$

l representa o tamanho da janela onde é calculada a distância.

O cálculo de $F^2(l)$ faz-se tomando o deslocamento l inicialmente o valor 1 que vai sendo incrementado até ao máximo de metade do tamanho da série.

Exemplificando, l toma inicialmente o valor igual a 1 e então são calculadas as médias das distâncias *f(2)-f(1), f(3)-f(2), f(4)-f(3),..., f(n)-f(n-1)* que são usadas no cálculo de *F(1).* Seguidamente l toma o valor 2 e são calculadas médias das distâncias *f(3)-f(1), f(4)-f(2), f(5)-f(3),..., f(n)-f (n-2)* que são usadas no cálculo de *F(2). l* sucessivamente toma o valor 3, 4, 5,...n-1 e de forma idêntica são construídos *F(3), F(4)....,F(n).*

No limite da correlação de gama longa, será uma função de escala na forma

$$F(l) \approx l^{\alpha}$$

α obtém-se representando $F(\ell)$ em função de *l* em escala dupla logarítmica e a inclinação da recta terá valor igual a α. α distingue sequências que apresentam correlação de gama longa das que são constituídas por símbolos não correlacionados.

$\alpha \neq 0.5$ indica a presença de correlação de gama longa
$\alpha = 0.5$ indica que a sequência é formada por símbolos não correlacionados.

Aplicação em textos

De volta às linguagens naturais, Schenkel [Schenkel,A.92] utilizou o método de Peng para analisar propriedades estatísticas de vários escritos.
Analisou textos em inglês. Um texto em inglês é escrito com 32 símbolos, letras e sinais de pontuação. Sendo $35=2^5$, um código de 5 bit possibilita a codificação de cada símbolo.
Verificou que para um *l*, referido no DNA walk, a partir de um determinado valor os detalhes do código binário utilizado não eram importante.
Codificou pois cada letra com um código arbitrário de 5 bits e transformou o texto numa sequência binária. Aplicou à sequência binária do texto o método do DNA walk obtendo para cada texto um valor de α. Na Tabela 4.3.4.2.1.estão registados os valores de α para cada texto analisado.

Tabela 4.3.4.2.1. Valores de α para textos

Texto	α
"A Máquina do Tempo" de Herbert G. Wells	0.52
"Guerra dos Mundos" de de Herbert G. Wells	0.62
"Alice no País das maravilhas" de Lewis Carol	0.65
"Hamlet" de Shakespeare	0.56
"Romeu e Julieta" de Shakespeare	0.60
"Dicionário de 1911", publicado em versão electrónica pela Croweel Company	0.67
"Bíblia" na versão electrónica da Universidade da Califórnia at Berkeley	0.87
"Corão", publicado por Tarik Tarsil	0.56

Pela diferença entre os valores de α dos textos de Herbert G. Wells e de Shakespeare, uma vez que textos escritos pelo mesmo autor têm valores de α distintos, Schenkel considerou que a medida não caracteriza o autor.

Surpreendeu-se com o valor alto de α para o dicionário, visto considerar que o significado de cada palavra não teria grande relação com os anteriores ou seguintes e a ordem de aparecimento das palavras não era importante. No entanto, ao analisar o dicionário depois de ter colocado as palavras por ordem aleatória, obteve para α o valor 0,5 o que indica a aleatoriedade na nova ordenação.
Analisou segundo o mesmo método programas de computadores, e encontrou em linguagens de programação sempre α perto de 0.9. Considerou que os programas apresentavam uma correlação mais forte. No entanto, não aprofundou mais o trabalho.
Nesta dissertação, não só alargamos o âmbito do estudo de Schenkel, como tiramos conclusões dos resultados.

Na secção 4.6.3. explicitamos os cálculos que serão usados para analizar os programas em C.

4.6.3.A correlação de gama longa para programas escritos na linguagem C

Como salientámos na secção 4.3.4.2., a correlação de gama longa representa um mecanismo de "memória". A correlação de gama longa quantifica o facto de um valor, ou conjunto de valores, da sequência estarem relacionados não só com os que imediatamente os precedem, mas com outras flutuações de valores afastados na sequência. O "longo" do conceito de correlação de gama longa significa que a correlação existe também entre estruturas afastadas. Um valor que observamos depende não só dos valores da sua vizinhança, mas depende também dos valores afastados (observados a distâncias longas) da série.
Atendendo a que:

- intuitivamente, a correlação de gama longa coincide com a avaliação da presença de estrutura que pretendemos que um programa detenha,
- o seu cálculo, usando o método de Peng, pode ser facilmente introduzido num ambiente de desenvolvimento e
- o seu valor traduz uma qualidade global do sistema.

Consideramos que, conjuntamente com as medidas da entropia, a correlação de gama longa é uma métrica adequada à análise da estrutura dos programas escritos em C.

Descrevemos de seguida o método de cálculo da correlação de gama longa que utilizamos.
O método do passeio aleatório é executado [Peng,C.92] adaptando–o da seguinte forma: transformamos o programa escrito em C num passeio aleatório. Para tal, e porque na linguagens naturais são as palavras as unidades portadoras de significado, usamos uma tabela de codificação de todas as palavras chave entre *-N/2* e *N/2*, sendo *N* a cardinalidade do conjunto das palavras chave. Os restantes símbolos léxicos são ignorados.
O passeio aleatório é construído interpretando cada valor da sequência como uma descida, se é positiva a diferença entre os códigos, ou subida se é negativo. O tamanho da descida ou da subida será o do valor absoluto do código.
Constituído o passeio aleatório uni-dimensional passamos ao cálculo da flutuação, $F(\tilde{\ })$, segundo a definição dada na secção 4.6.2..

Exemplo 4.6.3.1: Para calcular a correlação de gama longa segundo o método do passeio aleatório consideramos a seguinte série de treze valores:

<2, -3, 4, 0, -5, 3, 1, 1, -2, -1, 4, 3, -3>

O passeio aleatório cumulativo é dado pela soma de dois valores consecutivos da série:

<2, 2-3, -3+4, 4+0, 0-5, -5+3, 3+1, 1+1, 1-2, -2+1, -1+4, 4+3, 3-3>=
<2, -1, 1, 4, -5, -2, 4, 2, -1, -3, 3, 7, 0>

De seguida é calculada a tabela das diferenças e quadrados das diferenças com os vários comprimentos *l*. Para *l*=1,as diferenças *F(l)* são calculadas como:

< -1-2, 1- -1, 4-1, -5-4, -2- -5, 4- -2, 2-4, -1-2, -3- -1, 3- -3, 7-3, 0-7 >=
<-3, 2, 3, -9, 3, 6, -2, -3, -2, 6, 4, -7 >

Para a série deste exemplo, o cálculo das diferenças e quadrados das diferenças é apresentado

na tabela 4.6.3.1

Tabela 4.6.3.1. Diferenças e quadrados das diferenças

l	Diferenças	Quadrados das diferenças
1	-3 2 3 -9 3 6 -2 -3 -2 6 4 -7	9 4 9 81 9 36 4 9 4 36 16 49
2	-1 5 -6 -6 9 4 -5 -5 4 10 -3	1 25 36 36 81 16 25 25 16 100 9
3	2 -4 -3 0 7 1 -7 1 8 3	4 16 9 0 49 1 49 1 64 9
4	-7 -1 3 -2 4 -1 -1 5 1	49 1 9 4 16 1 1 25 1
5	-4 5 1 -5 2 5 3 -2	16 25 1 25 4 25 9 4
6	2 3 -2 -7 8 9 -4	4 9 4 49 64 81 16
7	0 0 -4 -1 12 2	0 0 16 1 144 4
8	-3 -2 2 3 5	9 4 4 9 25
9	-5 4 6 -4	25 16 36 16
10	1 8 -1	1 64 1
11	5 1	25 1
12	-2	4

Por fim, F(l) é calculado com base na tabela 4.6.3.2..

O esforço computacional do cálculo de F(l) é fácil de identificar. Para uma sequência de comprimento N, o esforço é igual a duas vezes (N-1)+(N-2)+...+1. Assim, o esforço computacional é $O(N^2)$.

A validade do α é dada pelo coeficiente de ajustamento da recta de representação de F(l) em relação a l, na escala dupla logarítmica. Havendo variações significativas nas partes iniciais e perca de estatística nas partes finais, como pode ser observado, por exemplo, na figura 5.4.4.1, o comprimento do texto deve ser superior às duas janelas a eliminar.

O capítulo cinco apresenta os resultados da aplicação do método de cálculo da correlação de gama longa em programas escritos em C, seguidos pela discussão acerca da sua interpretação.

Tabela 4.6.3. 2. Cálculo de *F(l)*

l	**Soma distância**	**Quadrado da Soma distância**	***F(l)***
1	-2	266	20.4
2	6	370	28.2
3	8	202	15.1
4	1	107	8.2
5	5	109	8.2
6	9	227	16.9
7	9	165	12.2
8	5	51	3.7
9	1	93	7.1
10	8	66	4.6
11	6	26	1.7
12	-2	4	0.2
13	0	0	0

4.7. Apresentação das hipóteses formuladas nesta dissertação

Tendo sido apresentados os conceitos de entropia, de lei de escalonamento e de correlação, acrescentamos definições utilizadas nas hipóteses formuladas nesta dissertação.

Definição 4.4.1.: Um programa é uma sequência de código, escrita com o propósito de resolver um problema. ⊗

Definição 4.4.2.: Um compilador é um programa que transcreve um texto escrito numa linguagem de programação de alto nível em código directamente reconhecido e executado por uma máquina. ⊗

Definição 4.4.3.: Uma palavra-chave ou palavra-chave de uma linguagem é uma palavra que representa uma instrução ou um tipo de dados dessa linguagem. Está sujeita, por isso, a restrições no seu uso pelas regras gramaticais da linguagem. ⊗

A linguagem utilizada nesta dissertação é o C, que é uma linguagem de programação de propósito geral, não específica de qualquer máquina ou sistema operativo [Kernighan,B.88].

Hipóteses formuladas:

As hipóteses testadas nesta dissertação podem ser reunidas em 4 grupos:

- hipóteses acerca da pesquisa de regularidade,
- hipóteses sobre a pesquisa de regularidade na estrutura sintáctica e na estrutura funcional e
- hipóteses do uso das medidas obtidas para a classificação dos programas.
- hipóteses de aplicação da métrica estudada no controlo do processo de software.

Os dois primeiros grupos de hipóteses são estudados no capítulo 5. O terceiro grupo, atendendo à sua especificidade, é objecto de um capítulo separado – o capítulo 6. No capítulo 7 apresentamos a aplicação da métrica no controlo do processo de uma amostra de programas.

No capítulo 5 testamos as seguintes hipóteses:

Hipótese nº1 – A distribuição de frequências de palavras-chave de compiladores escritos em linguagem C é distinta da distribuição de frequências de palavras-chave de um conjunto significativo de outros programas escritos na mesma linguagem.

Hipótese nº2: A classe funcional dos compiladores codificados em linguagem C obedece à Lei de Zipf ajustada.

Hipótese nº3: A correlação de gama longa mede a existência de estrutura em algoritmos codificados em linguagem C, permitindo distinguir este grupo de algoritmos doutros, cujo código tenha sido gerado aleatoriamente.

As três hipótese têm por objectivo verificar se a medida de frequências, as constantes da lei de Zipf e a correlação de gama longa são suficientes para caracterizar a estrutura de um programa escrito em C.

4.8. Contribuição deste capítulo para a defesa da tese que é objecto desta dissertação

A contribuição deste capítulo para a defesa da tese que é objecto desta dissertação consiste na análise comparativa das métricas de complexidade, na selecção das métricas potencialmente mais úteis para determinação da regularidade de programas escritos em C e na formulação das hipóteses a serem testadas nos restantes capítulos desta dissertação.

5. Regularidade em programas

...Em Ciência, noções como complexidade, imprevisibilidade, caos, por mais sugestivas que elas sejam, são noções vagas, sem grande valor, até ao momento em que são traduzidas em quantidades que possam ser expressas matematicamente e correspondam a grandezas que possam ser medidas...

R. Vilela Mendes in Colóquio Fundação Gulbenkian

5.1. Introdução

Neste capítulo descrevemos a amostra (o conjunto dos dados) sobre a qual trabalhamos, os programas utilizados no estudo experimental que efectuamos e apresentamos os resultados obtidos na experimentação. Concluímos este capítulo com a discussão das três primeiras hipóteses formuladas nesta dissertação.

5.2. Descrição da amostra

Na sua totalidade a amostra utilizada consiste num conjunto de programas escritos na linguagem C. Este conjunto pode ser descrito como constituído por dois grupos fundamentais: o grupo dos *compiladores* e o grupo dos *não compiladores*.

Neste último grupo encontram-se programas que implementam diversas funcionalidades, de entre elas: a contagem de elementos de um ficheiro, a ordenação de um ficheiro, a geração de números pseudo aleatórios e o cálculo de datas. E ainda: cálculo matricial, cálculo numérico em tratamento de séries, execução de contabilidade, execução de informação para gestão e aquisição de dados através de varrimento e outros dispositivos. Os programas foram escritos por diferentes programadores. Na sua totalidade, o conjunto de programas corresponde a cerca de 20.000 linhas de código. A variedade dos programas e o tamanho da amostra garantem a sua representatividade estatística.

O primeiro grupo é mais homogéneo, sendo constituído por 36 programas escritos em linguagem C, que implementam um compilador de uma linguagem estruturada em blocos, denominada zeta. Os referidos programas foram construídos por alunos do 3º ano da licenciatura em Engenharia Informática e Computadores do Instituto Superior Técnico, no ano lectivo 2000/2001, como trabalho prático da disciplina de compiladores, leccionada pelo Professor Doutor Thibault Langlois. O enunciado do trabalho prático referido encontra-se no anexo 5.2.1. Todos os programas têm mais de 2.000 linhas de código e em média 1400 palavras-chave. A implementação dos compiladores foi feita usando código gerado e código escrito. O código gerado foi obtido com os geradores de analisadores léxicos lex e geradores de analisadores sintácticos yacc. Estes geradores geram programas em C, que executam, respectivamente, a análise léxica e sintáctica do compilador. No anexo 5.2.2. encontra-se impresso um exemplo dos programas "makefile" que geram o compilador. O "makefile" varia de projecto para projecto.
A tabela 5.2.1 resume as principais características dos dois grupos fundamentais da amostra.

Tabela 5.2.1 Descrição dos dois grupos fundamentais da amostra

Grupo	**Tipo**	**Nº de Programas**	**Nºde linhas de código**
Primeiro	Compiladores	36	74982
Segundo	Programas variados	118	19992

Na verificação da primeira hipótese, segundo a qual, a classe funcional dos compiladores escritos em linguagem C tem uma distribuição de frequências de palavras-chave, distinta da distribuição de frequências de palavras-chave característica de um conjunto significativo de outros programas escritos em linguagem C, calculamos a distribuição de frequências das palavras-chave para cada um dos dois grupos fundamentais e obtivemos resultados substancialmente diferentes.

A partir dos programas constituintes do grupo fundamental foi possível verificar a validade da segunda hipótese, segundo a qual, a classe funcional dos compiladores codificados em linguagem C obedece à lei de Zipf ajustada. Os resultados da verificação de ambas as hipóteses são apresentados na secção 5.4. Entretanto recorda-se que a principal objectivo desta dissertação recai sobre a pesquisa de regularidades na estrutura sintáctica e na estrutura funcional dos programas escritos em C. Assim sendo, é necessário incluir na amostra grupos de programas, os quais por construção careçam de estrutura. Estes grupos, cujos elementos serão gerados aleatoriamente, servirão de controlo à experimentação das hipóteses. Mais ainda, torna-se necessário distinguir ausência de estrutura sintáctica e ausência de estrutura funcional. Para tal, a amostra passa a incluir mais três grupos de programas além dos dois grupos fundamentais. Os novos grupos são designados Grupo A, Grupo B e Grupo C têm a seguinte constituição:

Grupo A: um conjunto de 36 programas gerados a partir dos programas do primeiro grupo fundamental, sendo cada um dos programas do grupo A o resultado da sequenciação aleatória das palavras-chave de cada um dos programas do primeiro grupo fundamental. Como consequência estes programas: não têm estrutura sintáctica (apresentam erros de compilação) e tão pouco têm estrutura funcional, ou seja, não desempenham qualquer função.

Grupo B: um conjunto de 36 programas gerados, com o mesmo numero de linhas e sem a mesma distribuição de palavras-chave encontradas nos programas do primeiro grupo fundamental e construídos de forma a garantir em cada um dos programas do grupo B:

- a ausência de erros de sintaxe e
- o não desempenho de qualquer função

Como consequência estes programas têm estrutura sintáctica (não apresentam erros de compilação) mas não têm estrutura funcional, ou seja, não desempenham qualquer função.

Grupo C: um conjunto de 36 programas gerados a partir do mesmo conjunto de palavras-chave encontradas nos programas do primeiro grupo fundamental e construídos de forma a garantir em cada um dos programas do grupo C:

- a ausência de erros de sintaxe,
- o não desempenho de qualquer função e
- a mesma distribuição de frequências de palavras-chave do primeiro grupo fundamental,

Como consequência estes programas têm estrutura sintáctica (não apresentam erros de compilação) e não têm estrutura funcional, ou seja, não desempenham qualquer função.
Quanto à distribuição de frequências das palavras-chave, é aproximada com uma diferença de menos de 1%.

Na tabela 5.2.2. apresentamos os três novos grupos da amostra – os grupos de controlo - todos eles com 36 elementos, e as suas principais propriedades. O termo "preservação da sintaxe" indica a situação de correcção sintáctica e o termo "preservação da distribuição" indica que se manteve a distribuição de frequências de palavras-chave característica do grupo de compiladores.

Tabela 5.2.2. Descrição dos três grupos de controlo da amostra

Grupo	**Nº de linhas de código**	**Preservação da Sintaxe**	**Preservação da Distribuição**
A	74982	Não	Sim
B	~74000	Sim	Não
C	~74000	Sim	Sim

5.3. Descrição dos programas utilizados na experimentação

Registamos uma breve descrição dos programas utilizados sobre os cinco grupos da amostra. Pretendemos facilitar a compreensão dos processos de cálculo utilizados e evitar a necessidade de interpretação dos programas que listamos no anexo 5.3.1.

Keywords - recebe um ficheiro de texto e retorna um ficheiro que contém a sequência das palavras-chaves da linguagem C pela ordem encontrada no ficheiro inicial. Cada linha do ficheiro corresponde a uma palavra-chave.

Lcr - recebe um ficheiro de texto com uma sequência de palavras-chaves da linguagem C e retorna o gráfico das flutuações em relação à distância usada para calcular as flutuações *F(l)* e os valores da correlação de gama longa correspondentes.

Ale - recebe um ficheiro de texto com uma sequência de palavras-chaves da linguagem C e retorna, para uma sequência aleatória, criada com as mesmas palavras-chaves da sequência de entrada, o gráfico das flutuações em relação à distância usada para calcular as flutuações *F(l)* e os valores da correlação de gama longa correspondentes.

Entropia - recebe um ficheiro de texto com uma sequência de palavras-chaves da linguagem C e retorna, os valores da entropia de Shannon da série e da sucessão das entropias locais calculadas sobre partições da série de duas a dez partes.

Gerador 1 - gera um programa sintacticamente correcto em C, com um determinado número de linhas de código.

Gerador 2 - gera um programa sintacticamente correcto em C, com um determinado número de linhas de código e uma determinada distribuição de frequências de palavras-chaves fornecidos como parâmetros.

Cluster - que executa uma análise taxonómica recebendo um ficheiro com os dados respeitantes aos atributos escolhidos para caracterizar cada programa e o número de grupos onde se pretende separar a amostra e retorna a uma lista contendo o grupo de cada programa.

5.4. Resultados

Descrevemos de seguida as medidas obtidas com a utilização das métricas descritas no capítulo 4. Em anexo 5.4. estão as tabelas dos valores calculados para cada compilador, que serviram de base às estatísticas e aos gráficos desta secção.

5.4.1.Entropia de Shannon dos dois grupos fundamentais da amostra

Na verificação da primeira hipótese, segundo a qual, a classe funcional dos compiladores escritos em linguagem C tem uma distribuição de frequências de palavras-chave distinta da distribuição de frequências de palavras-chave característica de um conjunto significativo de

outros programas escritos em linguagem C, calculamos a distribuição de frequências das palavras-chave para cada um dos dois grupos fundamentais e obtivemos resultados substancialmente diferentes.
Tal como já apresentado no capítulo 4, a entropia de Shannon dá-nos uma medida da uniformidade de uma determinada distribuição de probabilidades.
Utilizamos esta medida para investigar a existência de uma distribuição de frequências que pudesse caracterizar a classe funcional dos compiladores escritos em linguagem C.

Para tal começamos por avaliar a entropia de Shannon de cada compilador, ou seja, de cada elemento do primeiro grupo fundamental da amostra. A tabela 5.4.1.1 lista os resultados obtidos nesta avaliação.

Tabela 5.4.1.1. Estatísticas para a entropia de Shannon dos compiladores

Média	3,7990
Desvio	0,1614
Valor Mínimo	3,3067
Valor Máximo	4,5552

A regularidade encontrada nos valores da entropia de cada elemento do segundo grupo fundamental confirma a hipótese da existência de uma distribuição de frequências característica deste grupo de programas.

A representação gráfica dos valores da entropia de Shannon de cada compilador é apresentada na Figura 5.4.1.1.

Em seguida avaliamos a mesma propriedade (a entropia de Shannon) nos programas pertencentes ao segundo grupo fundamental da amostra (programas com várias funcionalidades) obtivemos os resultados apresentados na tabela 5.4.1.2. Nota-se que estes apresentam um valor médio mais baixo, estatisticamente diferente da entropia de Shannon calculada na amostra dos programas pertencentes ao primeiro grupo fundamental da amostra (compiladores) (teste t de Student amostras não emparelhadas, duas caudas com $p<0,05$).

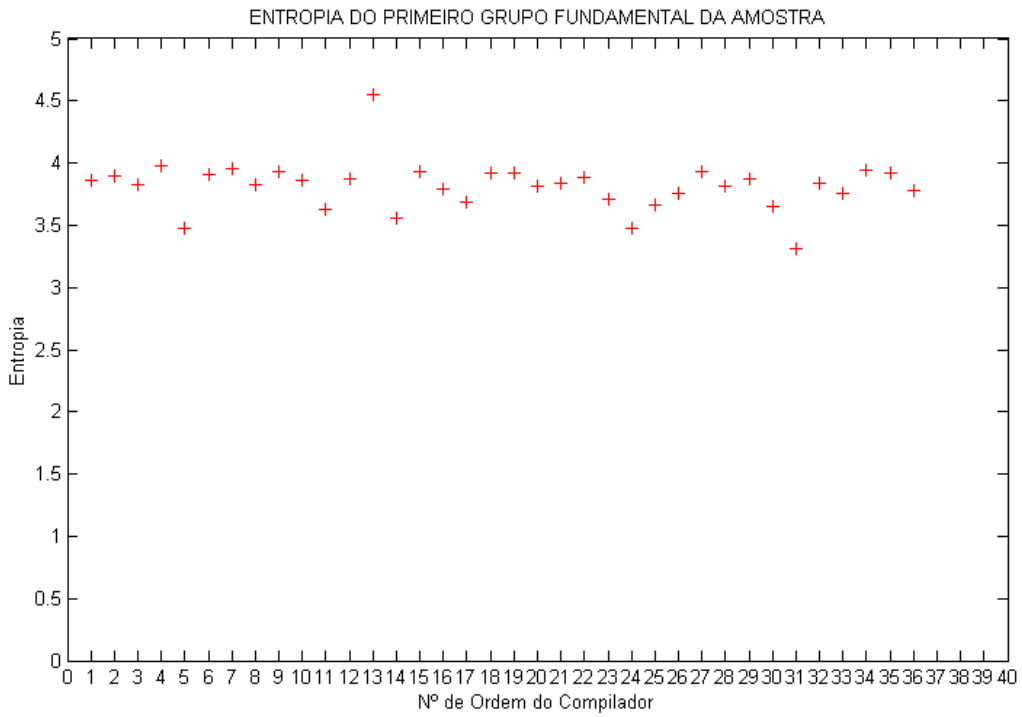


Figura 5.4.1.1. Valores da Entropia de Shannon para os programas do primeiro grupo fundamental da amostra

Tabela 5.4.1.2. Estatísticas de entropia de Shannon para os programas do segundo grupo da amostra

Média	2,8203
Desvio	0,402
Valor Mínimo	1,9411
Valor Máximo	3,7103

Podemos, pois, verificar que a classe funcional dos compiladores escritos em linguagem C tem uma distribuição de frequências de palavras-chave distinta da distribuição de frequências

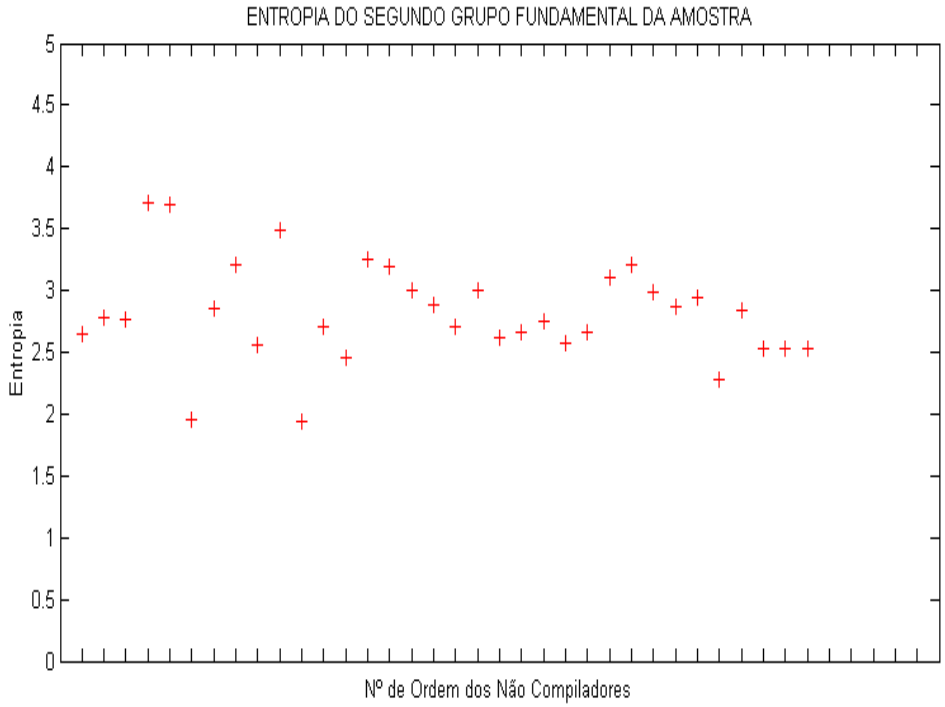


Figura 5.4.1.2. Valores da entropia de Shannon para os programas do segundo grupo fundamental da amostra

de palavras-chave característica de um conjunto significativo de outros programas escritos em linguagem C.

Os resultados suscitam os seguintes comentários.

1. A regularidade encontrada na entropia do primeiro grupo pode dever-se a uma parte significativa dos programas ser gerada pelas mesmas ferramentas e a restante parte ser
2. implementada por projectistas (estudantes do mesmo curso de licenciatura) com práticas de programação semelhantes.
3. A maior dispersão dos valores de entropia do segundo grupo pode ser justificada pela maior variedade dos objectivos funcionais dos programas e/ou pela diferença de práticas de programação dos projectistas.
4. A diferença das médias na entropia dos dois grupos é de esperar, porque não é possível garantir que qualquer das amostras seja representativa de todos os programas escritos em C.

5.4.2.Entropia de Rényi dos dois grupos fundamentais da amostra

Depois de identificar as entropias de Shannon para os dois grupos fundamentais, coloca-se a questão de saber se todas as palavras-chave contribuem da mesma forma para a determinação das regularidades.

No prosseguimento da pesquisa de distribuições características das frequências de palavras-chave nos compiladores e nos não compiladores escritos em C, optamos por calcular para o primeiro grupos fundamental da amostra as entropias de Rényi. Pretendemos verificar com esta medida, a qual pesa de forma diferente as probabilidades próximas de um e as probabilidades próximas de zero, se as entropias de Rényi dos programas deste grupo variam substancialmente.

Na expressão da entropia de Rényi, a probabilidade levantada a uma potência (α), permite valorizar a influência quer das probabilidades grandes, quer das probabilidades pequenas, na distribuição de probabilidades que se analisa. Se $\alpha > 1$, as probabilidades altas da distribuição são valorizadas em relação às probabilidades baixas. Se $\alpha < 1$, então são as probabilidades baixas da distribuição a serem valorizadas. Como tal, escolhemos calcular esta entropia para dois valores de alfa menores que um: um oitavo e um terço, e para dois valores de alfa maiores que 1: três e oito.

Em relação aos valores de α menores que um, a amostra não apresenta grande dispersão sendo o desvio padrão, respectivamente, 2% e 4% da média. Podemos dizer que a distribuição de probabilidade dos símbolos mais frequentes se mantém estável. Pelo contrário, em relação aos valores do α maiores que um, a amostra apresenta grande dispersão sendo o desvio padrão, respectivamente, 256% e 588% da média.

Podemos dizer que a distribuição dos símbolos menos frequentes é variável e, como tal, não pode ser usada para caracterizar a classe funcional dos compiladores.

Os resultados suscitam os seguintes comentários.

1. Na linguagem C, a redução de frequência de algumas palavras-chave é devida à sua aplicação muito restrita, por exemplo, os qualificadores *register* e *volatile*, ou por a sua utilização ser fortemente desaconselhada, por exemplo, o *goto*.
2. As palavras-chave menos frequentes não devem ser eliminadas das amostras, porque

em programas com outros objectivos e implementados por outras classes de projectistas podem ter maior peso. Por exemplo, num programa de micro-controlo, o qualificador *register* pode ser importante porque permite um acesso mais rápido a variáveis.

Tabela 5.4.2.1. Estatísticas da entropia de Rényi para o primeiro grupo fundamental da amostra

	alfa			
	1/8	**1/3**	**3**	**8**
Média	4,5684	4,5643	-0,0160	-0,000079
Desvio	0,1301	0,1851	0,0411	0,000465
Min.	4,0636	4,1384	-0,2530	-0,002793
Máx.	4,8532	5,3876	-0,0030	0,000000
Desvio/Média	0,028478	0,040554	2,56875	5,88608

5.4.3.A lei de Zipf ajustada para o segundo grupo fundamental da amostra

Nesta secção apresentamos os resultados obtidos ao longo da verificação da segunda hipótese,

Tentamos a aplicação da lei de Zipf à classe funcional dos compiladores codificados em C e encontramos uma relação entre o número de ordem da frequência de ocorrência das palavras-chave dos programas e frequência de ocorrência das palavras-chave.

Tendo em conta a expressão:

$$p(r) = A.r^{-\lambda}$$

onde:

r é o número de ordem de uma dada palavra-chave numa lista ordenada de acordo com a frequência de ocorrência das palavras-chave,

p(r) é a probabilidade de ocorrência da palavra de ordem *r* e *A* e λ são constantes.

calculamos valores de A e λ para um programa do segundo grupo fundamental da amostra (programas com diferentes funcionalidades), para três programas do primeiro grupo fundamental (compiladores) e para um programa do grupo de controlo B.

Na tabela 5.4.3.1 estão registados os valores de λ e da constante de ajustamento para os casos analisados.
Para os três compiladores, o expoente característico λ vale 1,56, 1,63, 1,81, enquanto que a constante de ajustamento (A) toma os valores de 2227.92, 2298.26, 4378.24, respectivamente.

Tabela 5.4.3.1. Valores das constantes da lei de Zipf

	Programa do segundo grupo	**Compilador 1**	**Compilador 2**	**Compilador 4**	**Programa do grupo de controlo B**
λ	1.55	1.56	1.63	1.81	0.67
A	927.90	2227.92	2298.26	4378.24	210.91

Os resultados, representados em escala dupla logarítmica, são apresentados nas figuras 5.4.3.1 para um programa do segundo grupo fundamental da amostra, 5.4.3.2 para um dos programas do primeiro grupo fundamental da amostra e 5.4.3.3 para o programa do grupo de controlo B.

Os programas pertencentes ao primeiro grupo (os compiladores) apresentam valores das duas constantes mais altos do que os encontrados no programa do segundo grupo.
Em relação aos programas pertencentes ao grupo de controlo B da amostra (programas que não desempenham qualquer função, correctos do ponto de vista sintáctico e sem a mesma distribuição de frequências de palavras-chave dos programas do segundo grupo fundamental), estes também obedecem à lei de Zipf, mas os valores das de λ e A são mais baixos que os observados nos dois grupos principais.
Para os três programas foi possível encontrar os valores de λ e A com um coeficiente de ajustamento acima de 0.89, sendo que no caso dos programas pertencentes ao primeiro grupo

(compiladores) o valor médio do coeficiente de ajustamento é 0,91 o que traduz a fiabilidade dos valores encontrados.
Podemos pois dizer que a lei de Zipf ajustada é verificada.

Os resultados suscitam os seguintes comentários:

1. A lei de Zift ajustada ser seguida para programas do grupo de controlo B era previsível, visto estes serem gramaticalmente correctos, e como tal deterem estrutura sintáctica.
2. Os valores das constantes λ e A do grupo de controlo B são mais baixos que os observados nos dois grupos principais. Este facto sugere a existência de alguma sensibilidade da lei de Zipf à estrutura funcional.
3. Todavia, se a lei de Zipf fosse aplicada a sequências aleatórias, com a mesma frequência de palavras-chave dos programas do primeiro grupo, seriam identificados os mesmos valores de A e λ, o que não permite a sua diferenciação.

Logo, tal como na entropia de Shanon e de Rényi, a lei de Zipf não é suficiente para obter a discriminação que se pretende atingir nesta dissertação.

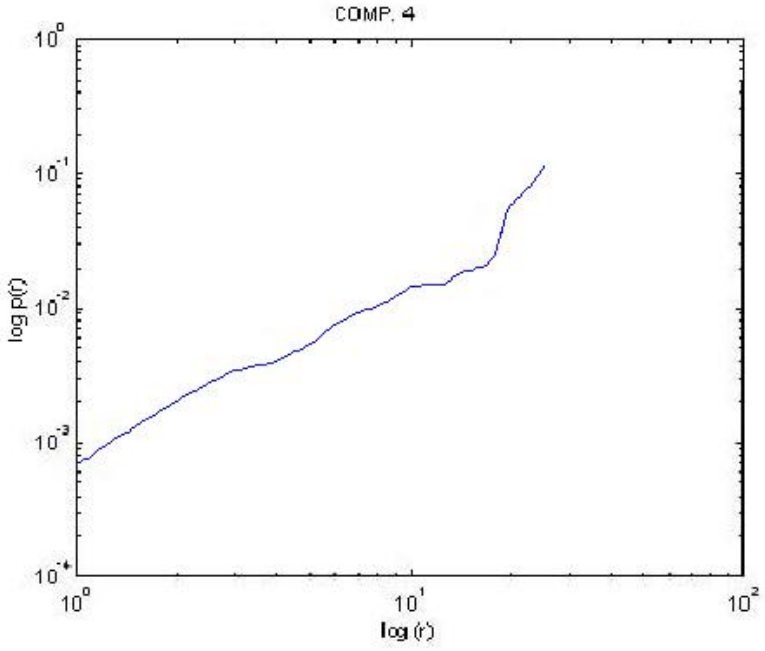


Figura 5.4.3.1. Lei de Zipf para programa "Vários" do segundo grupo fundamental da amostra

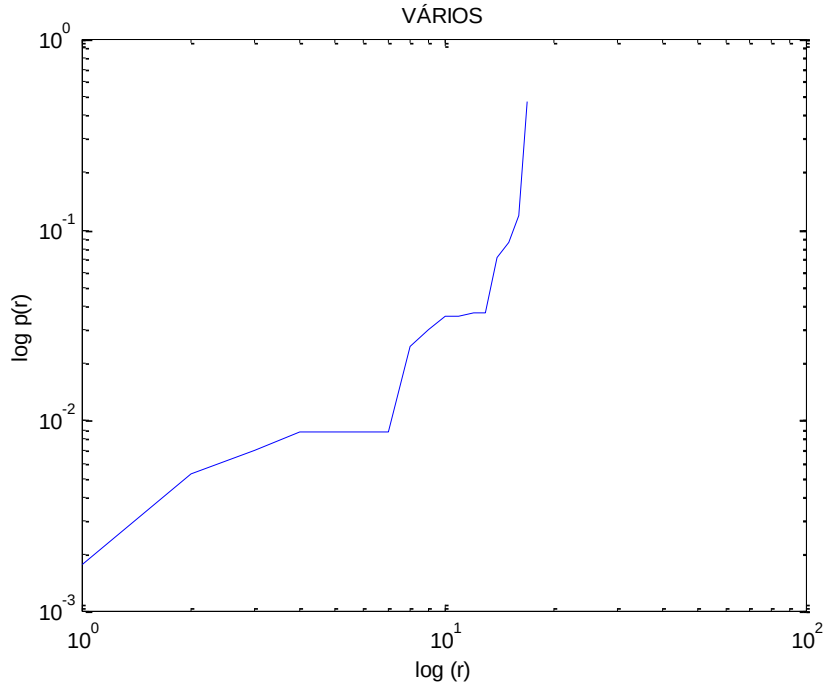


Figura 5.4.3.2. Lei de Zipf para um programa do primeiro grupo fundamental da amostra

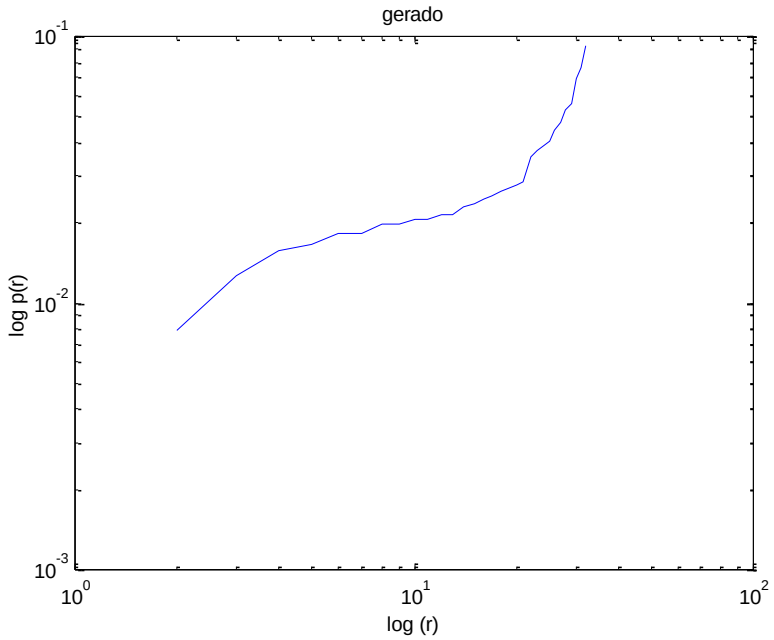


Figura 5.4.3.3. Lei de Zipf para um programa do grupo B de controlo

5.4.4.Regularidades nas estruturas sintáctica e funcional dos algoritmos

Até aqui temos trabalhado em redor da pesquisa de regularidades na estrutura sintáctica dos algoritmos pertencentes aos primeiro e segundo grupos fundamentais. A partir de agora passamos a utilizar, tal como fizemos na verificação da lei de Zipf para o último exemplo apresentado, os programas gerados e designados como pertencentes aos grupos de controlo A, B e C.

Passamos então a trabalhar sobre a pesquisa de regularidades que possam ser provenientes da estrutura funcional dos algoritmos. Sendo compilável, um algoritmo obedece à estrutura gramatical da linguagem e executando uma (ou mais) funções, tem uma estrutura funcional. A estrutura sintáctica é identificada por analisadores sintácticos, o que não acontece com a estrutura funcional.

Como já se apresentou anteriormente, a correlação de gama longa (CGL) é uma medida da auto-correlação existente entre os vários pontos de uma sequência, sendo muitas vezes mencionada como correlação estrutural. Quando existe, representa um mecanismo de "memória", indicando que um elemento, ou conjunto de elementos, da sequência em observação estão relacionados, não só com os que imediatamente os precedem, mas com outros conjuntos de valores afastados na sequência.

Uma vez que pretendemos analisar uma estrutura sintáctica e funcional na sua forma mais simples, centrámos o nosso estudo nos elementos básicos da linguagem, válidos em todas as implementações. Esses elementos são as palavras-chave. Considerámos, pois, que a sequência de palavras-chave era suficiente para representar as estruturas a medir. Como tal, criou-se o programa *Keywords*, que recebe cada um dos programas que constituem cada compilador e os transforma na sequência das palavras-chave da linguagem C ANSI, eliminando todos os restantes elementos léxicos. Esse programa é listado no anexo 5.3.1..

Por que necessitamos de valores numéricos para o cálculo da CGL, construímos um código numérico balanceado, a fim de transformar a sequência alfanumérica em sequência numérica centrada em zero. As palavras-chave e os códigos respectivos são indicados na tabela 5.4.4.1.

Tabela 5.4.4.1.– Código das palavras-chave

palavra	código	palavra	código	palavra	código	palavra	Código
auto	-16	double	-8	int	0	struct	8
break	-15	else	-7	long	1	switch	9
case	-14	enum	-6	register	2	typedef	10
char	-13	extern	-5	return	3	union	11
const	-12	float	-4	short	4	unsigned	12
continue	-11	for	-3	signed	5	void	13
default	-10	goto	-2	sizeof	6	volatile	14
do	-9	if	-1	static	7	while	15

Depois de codificadas as palavras-chave realizámos o cálculo da correlação de gama longa usando o método do passeio aleatório, tal como apresentado na secção 4.3.8.2..

Na figura 5.4.4.1. está representada uma das curvas obtidas para um programa do primeiro grupo fundamental. Lembramos que a CGL indica a existência de auto-correlação sempre que o valor do expoente característico (α) é significativamente superior a 0,5 (valor que caracteriza as sequências aleatórias). Quando este expoente apresenta o valor 0,5 não há auto-correlação.
Interessa-nos portanto investigar a existência de CGL, quer para o primeiro grupo fundamental da amostra, quer para os grupos de controlo A, B, C, de forma a poder precisar até que ponto diferentes valores para a CGL permitem distinguir programas com diferentes estruturas sintácticas e funcionais.

Usamos o primeiro grupo fundamental da amostra (os compiladores) e o grupo de controlo A (não preserva a sintaxe mas preserva a distribuição) para testar se a correlação de gama longa é sensível à existência de estrutura sintáctica. Por uma questão de representatividade estatística, os valores usados para o grupo A foram obtidos gerando, por cada compilador, 50 programas. O valor do expoente da CGL é neste caso a média dos valores do expoente da

CGL para os 50 programas gerados sem correcção sintáctica e com a distribuição de frequência de palavras-chave de cada compilador.

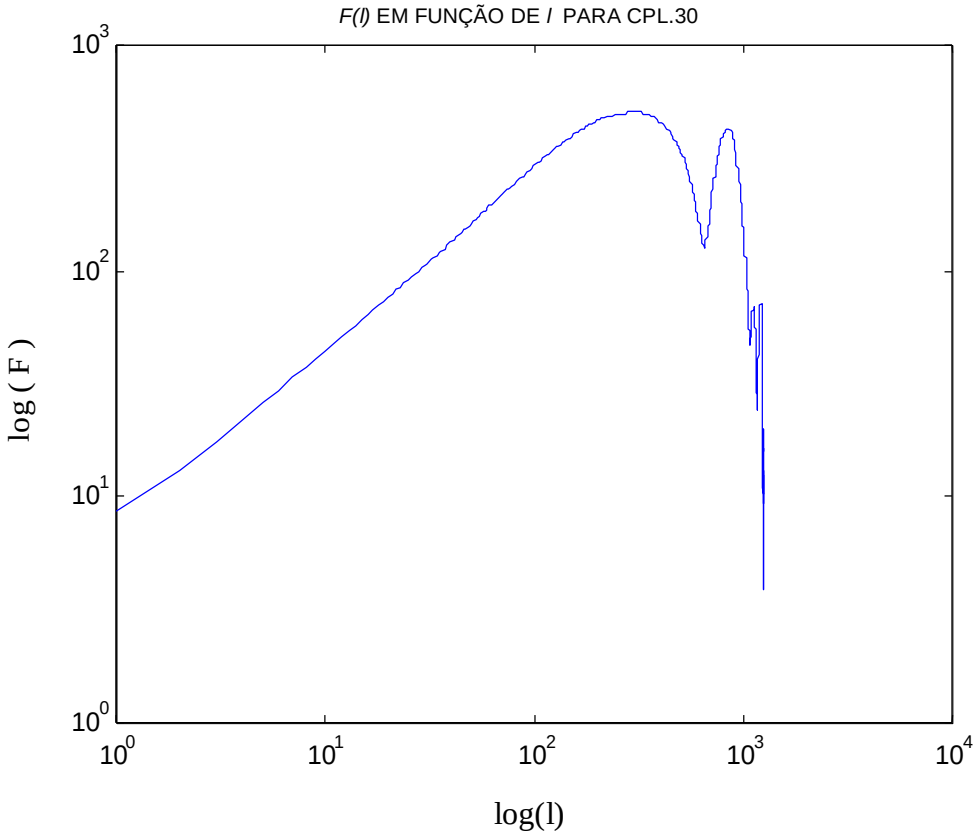


Figura 5.4.4.1. *F(l)* em função de *l* representada em escala logarítmica para o compilador 30

Na tabela 5.4.4.2 encontram-se as estatísticas das medidas de correlação da gama longa obtidas para o primeiro grupo fundamental da amostra (compiladores). Na tabela 5.4.4.3. encontram-se as estatísticas das medidas de correlação da gama longa obtidas para o grupo de controlo A.

Tabela 5.4.4.2. Estatísticas das medidas dos programas do primeiro grupo fundamental da amostra

Média	0,8179
Desvio Padrão	0,0551
Máximo	0,8997
Mínimo	0,6841

Tabela 5.4.4.3. Estatísticas das medidas dos programas do grupo de controlo A

Média	0,4832
Desvio Padrão	0,0243
Máximo	0,5137
Mínimo	0,4162

A figura 5.4.4.2 representa o expoente da CGL obtido para o primeiro grupo fundamental da amostra (compiladores) e a figura 5.4.4.3 representa o expoente da CGL obtido para o grupo de controlo A.

Tal como apresentado nas tabelas 5.4.4.2 e 5.4.4.3 e nas figuras 5.4.4.2 e 5.4.4.3 verifica-se que para os programas do primeiro grupo fundamental (compiladores) o valor médio do expoente característico é de cerca de 0,81 e a amostra tem um desvio padrão de cerca de 6% da média. Ao mesmo tempo, para os programas do grupo de controlo A (programas que não têm estrutura sintáctica e tão pouco têm estrutura funcional) os valores do expoente são ainda menos dispersos, com média de 0,48, muito mais próxima de 0,5 do que a média da primeira série e com mais de 75% dos valores entre 0,48 e 0,52.

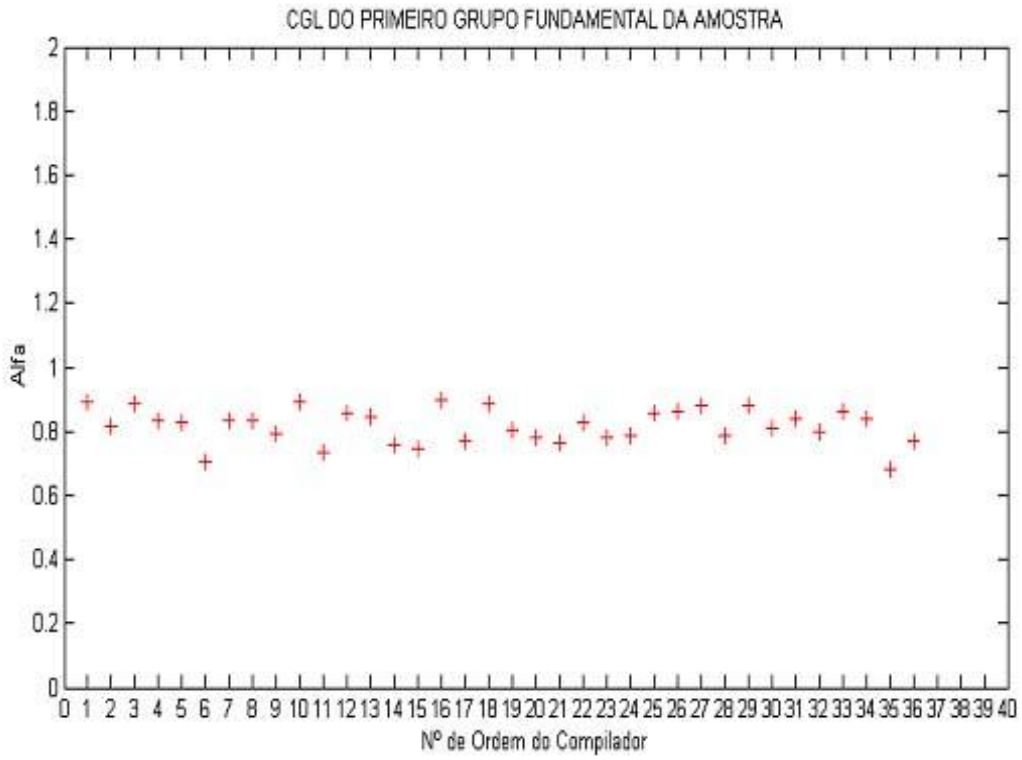


Figura 5.4.4.2. Valor o expoente da CGL para o primeiro grupo fundamental da amostra

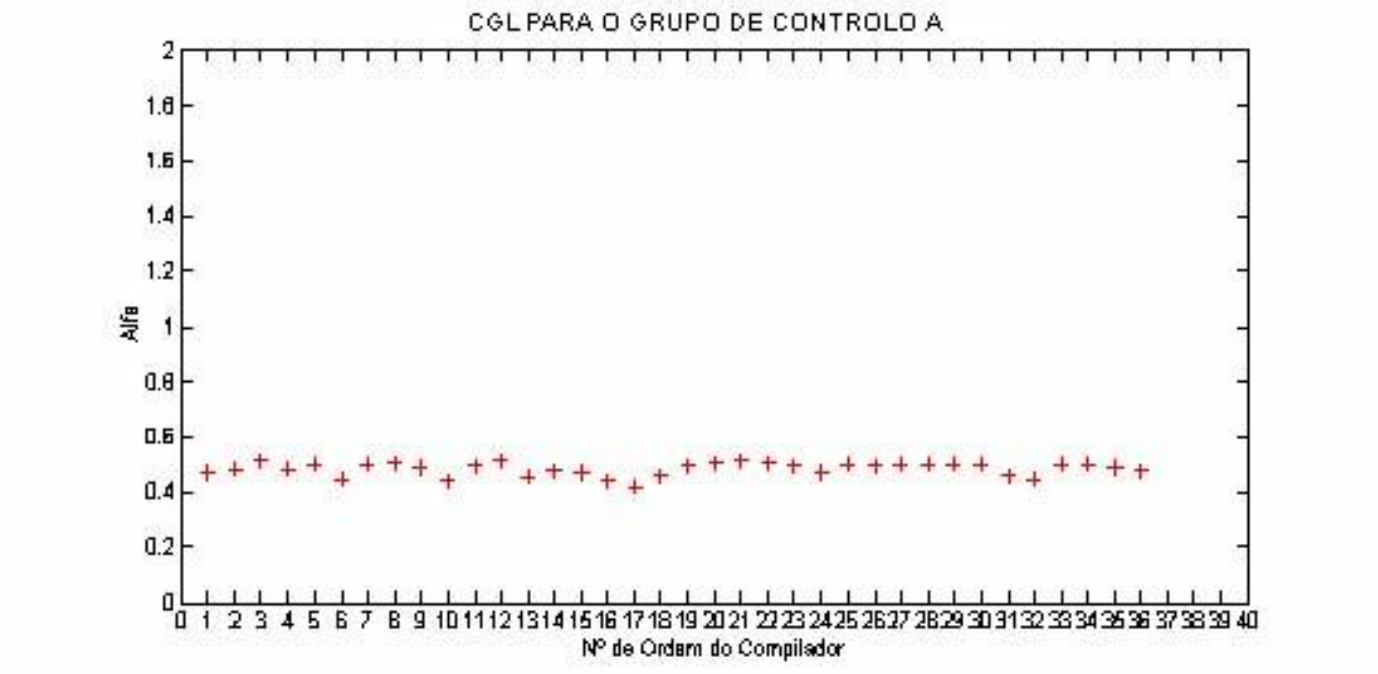


Figura 5.4.4.3. valores o expoente da CGL para o Grupo A

Nesta amostra, os valores maiores que 0,68 indicam que a sequência representa um algoritmo codificado na linguagem C compilável. Valores do intervalo 0,4832 ± 0,0243 têm probabilidade alta de representarem sequências aleatórias. A diferença das médias confirma a sensibilidade da correlação de gama longa à existência de estrutura sintáctica, ou seja, aquela que é imposta pela correcção sintáctica.

De seguida, usamos o primeiro grupo fundamental (compiladores) e o grupo de controlo B (preserva a sintaxe, não preserva a distribuição) para investigar o comportamento da correlação de gama longa em relação a estruturas sintácticas semelhantes.
Obtivemos medidas cujos valores estatísticos estão expressos na tabela 5.4.4.4.

Tabela 5.4.4.4. Estatísticas das medidas dos programas do grupo de controlo B

Média	0,3899
Desvio Padrão	0,1156
Máximo	0,5991
Mínimo	0,1233

A maior parte destas medidas pertencem ao intervalo de valores que indica a aleatoridade. Dado que neste grupo, os programas não têm a mesma medida em relação à distribuição de frequências das palavras-chave dos programas do primeiro grupo fundamental (compiladores), os mesmos não podem ser comparados com o primeiro grupo fundamental da amostra, optamos então pelo cálculo do valor da CGL para os programas do grupo de controlo C, ou seja, para aqueles que não desempenham qualquer função, preservam a sintaxe e mantém a mesma distribuição de frequências de palavras-chave do grupo dos compiladores. A ideia subjacente é investigar a relação entre os valores da CGL e os valores da referida distribuição.

Obtivemos medidas cujos valores estatísticos estão expressos na tabela 5.4.4.5.
Na figura 5.4.4.4. encontra-se o gráfico do expoente da CGL deste grupo de controlo C.

Verificamos, assim, que a organização inserida com a imposição da mesma distribuição de frequências de palavras-chave aproxima o valor de α do valor característico dos compiladores. Suspeitamos que a imposição da mesma distribuição de frequências de palavras-chave aliada à imposição da correcção sintáctica resultou numa estrutura mais próxima da estrutura funcional original.

Nem todos os programas do grupo de controlo C repõem a correlação de gama longa, por exemplo, os compiladores identificados pelos números 8, 10, 11, 31 e 33 não a repõem. Optamos então por investigar a proveniência, ou as características específicas dos programas que não efectuam esta reposição. Avançamos como hipótese que a referida reposição possa estar associada à localização das palavras-chave dentro dos programas. A fim de esclarecer os fundamentos desta suposição, medimos, para cada programa, o que designamos por entropia local.

Tabela 5.4.4.5.- Estatísticas das medidas para os programas do grupo de controlo C

Média	0,6158
Desvio Padrão	0,0817
Máximo	0,8173
Mínimo	0,4569

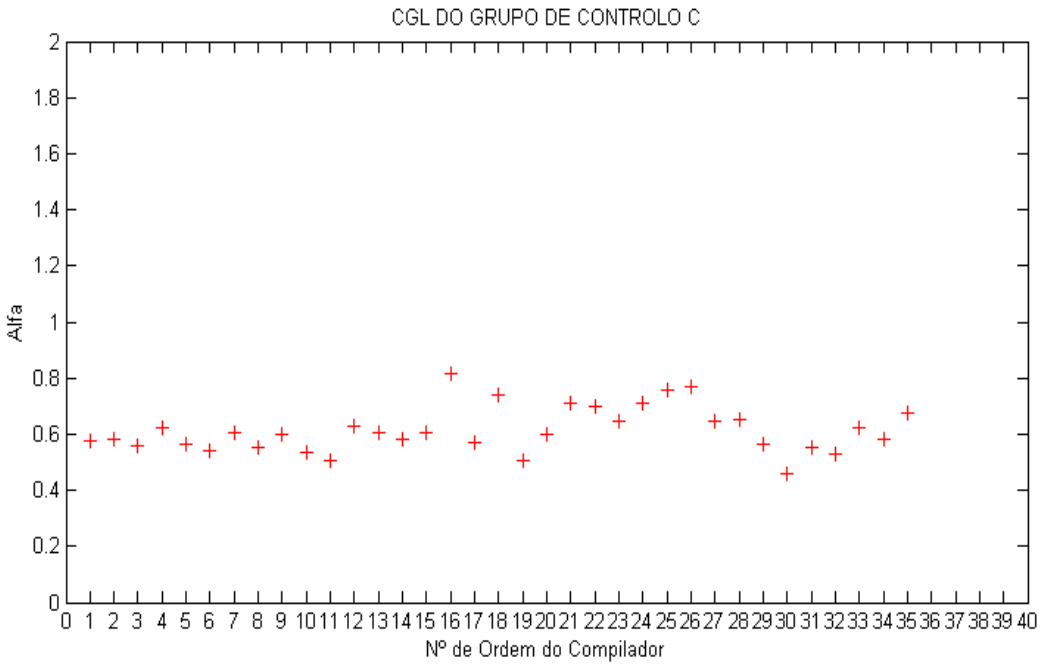


Figura 5.4.4.4. Valores do expoente da CGL para grupo de controlo C

Definição 5.4.4.1:A entropia local é a medida da entropia de Shannon efectuada para uma parte da série. ⊗

A entropia local é calculada da seguinte forma:

Cada programa é representado por uma sequência de palavras-chave x_m $m=1,2,....,n-1,n$ que tomam valores no conjunto das palavras-chave da linguagem x_k $k=1,2,....,36$

Divide-se a sequência $\{x_m\}$ em p partes de igual tamanho t, mais uma parte $r \leq t$, tal que:

$$n = \sum_{p} t + r \text{ e } n \geq tp$$

Designamos por $N(k,p,i)$ o número de ocorrências da palavra-chave k na parte de ordem i da divisão de sequência x_k em p partes ($i \leq p$).

Indica-se por $P(k,p,i)$ a frequência de ocorrência da palavra-chave k na parte de ordem i da divisão de sequência $x(k)$ em p partes ($i \leq p$).

Assim sendo

$$P(k,p,i) = N(k,p,i) \; / \sum_{k} N(k,p,i)$$

Calcula-se a Entropia de Shannon utilizando a expressão descrita na secção 4.3.5., para cada uma das partes.

$$HL\,(p,i) = \sum_{k} P(k,p,i) \,.\, log\, P(k,p,i)$$

Por exemplo, *HL (4,3)* indica o valor da entropia local da terceira parte do programa dividido em 4 partes.
O valor médio da entropia local medida na divisão do programa em *p* partes é a média das entropias locais obtidas para cada uma das partes.

$$HM\,(p) = \frac{1}{p} \sum_{i=1...p} HL\,(p,i)$$

Pesquisamos o valor da *HM(p)* para os programas 8, 10, 11, 31 e 33, do grupo C. Obtivemos medidas cujos valores estão expressos nos gráficos 5.4.4.5. a 5.4.5.9. As tabelas com os valores correspondentes encontram-se no anexo 5.4.2..

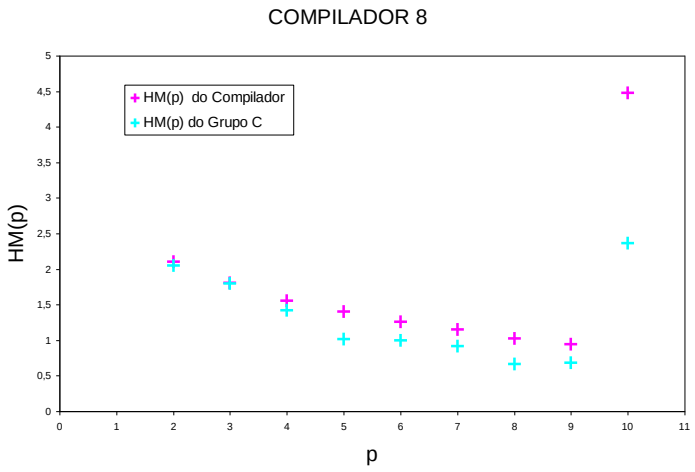


Figura 5.4.4.5. - Comparação das entropias médias locais para o compilador 8 e seu correspondente do grupo de controlo C

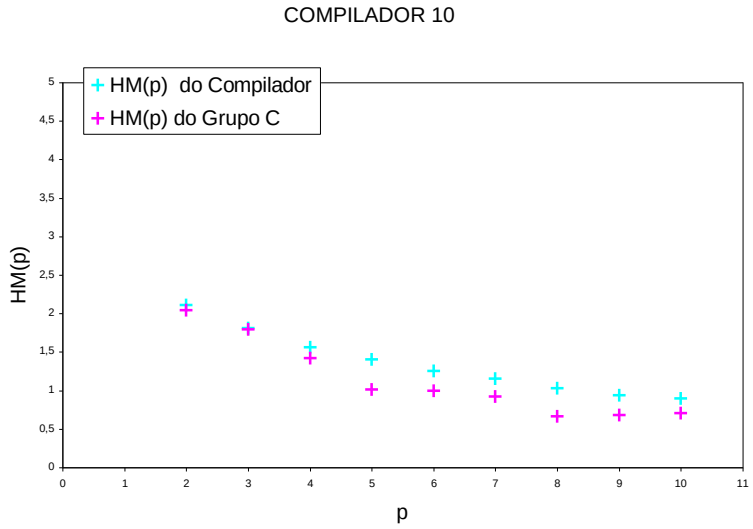


Figura 5.4.4.6. - Comparação das entropias médias locais para o compilador 10 e seu correspondente do grupo de controlo C

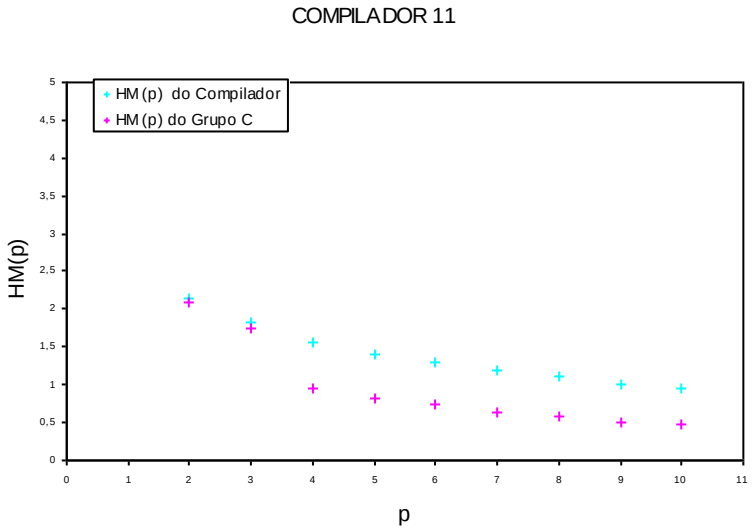


Figura 5.4.4.7. - Comparação das entropias médias locais para o compilador 11 e seu correspondente do grupo de controlo C

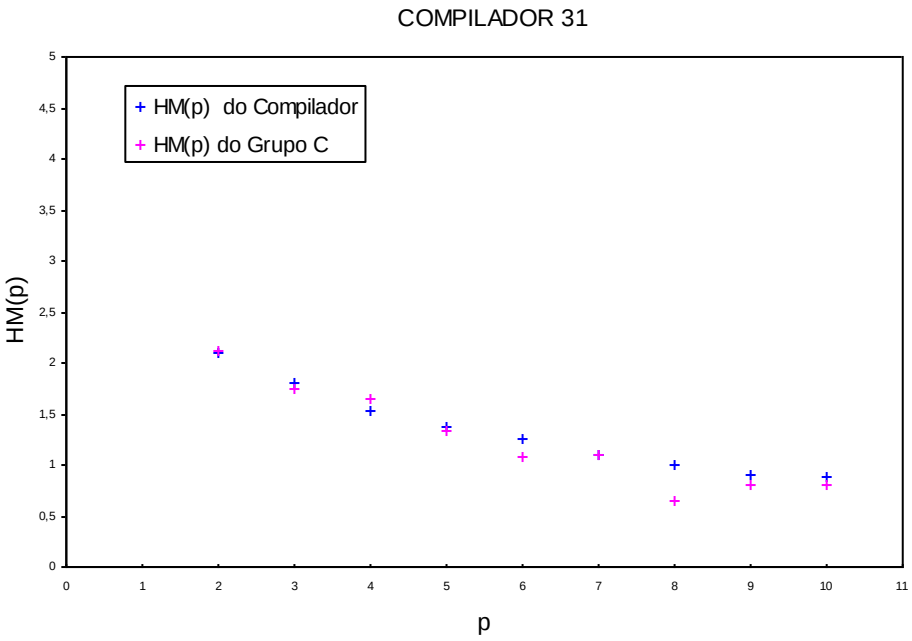


Figura 5.4.4.8. - Comparação das entropias médias locais para o compilador 31 e seu correspondente do grupo de controlo C

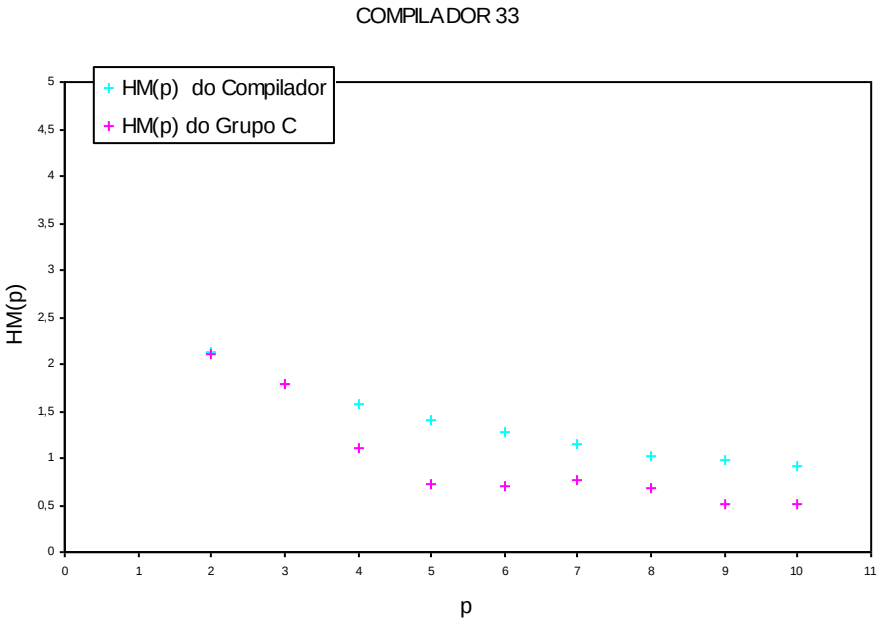


Figura 5.4.4.9. - Comparação das entropias médias locais para o compilador 33 e seu correspondente do grupo de controlo C

Observando os gráficos acima nota-se que, a partir da divisão em um número de partes superior a cinco, em geral, há uma significativa redução do valor da entropia média local do programa do grupo C, se compararmos com o mesmo valor calculado para o programa homólogo no primeiro grupo fundamental. Estes resultados confirmam a nossa suposição acerca da particular localização de palavras-chave nos programas do grupo C para os quais não se conseguiu um valor da CGL mais elevado com a imposição da distribuição original. Dada uma determinada distribuição de palavras-chave, se houver uma disposição consecutiva das mesmas palavras-chave (concentração) em partes (locais) da especificação do programa haverá uma consequente perda do mecanismo de memória traduzido no valor (do expoente) da CGL. É natural que, sendo a medida da CGL uma medida de memória longa da série em análise, esta memória se perca com a referida concentração.

Para confirmar as observações do parágrafo anterior, optamos por investigar o valor da *HM(p)* para os dois programas 28 e 39, do grupo C que efectuam esta reposição. Obtivemos medidas cujos valores estão expressos nos gráficos 5.4.4.10. e 5.4.4.11.. Neles podemos observar que as entropias se mantêm muito próximas mesmo para a divisão em 10. Neste caso a estrutura dos programas gerados é próxima da dos compiladores correspondentes.

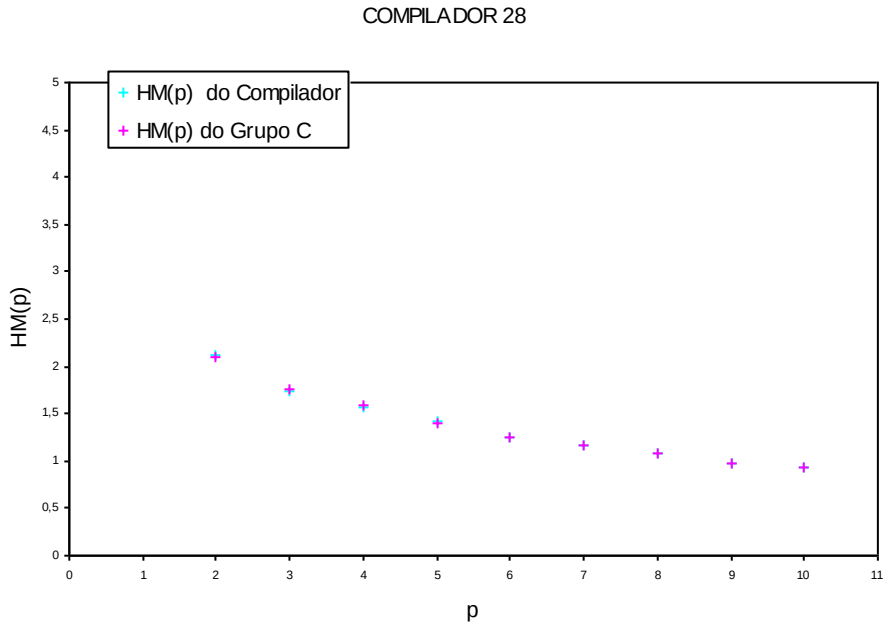


Figura 5.4.4.10. - Comparação das entropias médias locais para o compilador 28 e seu correspondente do grupo de controlo C

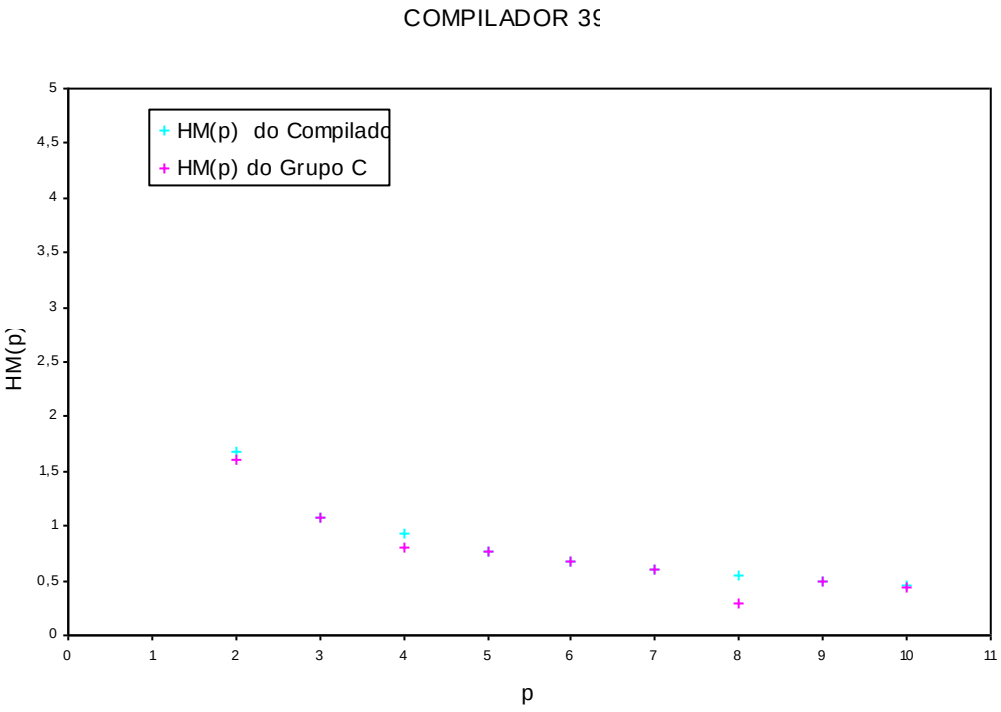


Figura 5.4.4.11. - Comparação das entropias médias locais para o compilador 39 e seu correspondente do grupo de controlo C

Estes resultados reforçam a nossa suposição acerca do efeito de uma particular localização de palavras-chave em alguns programas do grupo C.

Dada uma determinada distribuição de palavras-chave, se a mesma palavra for repetida muitas vezes em partes (locais) da especificação do programa, haverá uma consequente perda do mecanismo de memória traduzido no valor (do expoente) da CGL.

Os resultados suscitam os seguintes comentários:

1. Era de esperar que a correlação de gama longa não fosse mantida para o grupo de controlo A (não preserva a sintaxe e preserva a distribuição) visto uma sequência aleatória de palavras-chave, ainda que limitada pela imposição da frequência não é um programa, como tal só por acaso poderia deter estrutura sintáctica ou funcional.
2. Também era de esperar que a correlação de gama longa no grupo de controlo B (preserva a sintaxe e não preserva a distribuição) apresentasse vários valores próximos do aleatório, embora mais dispersa e com valores perto 0,38 e 0,6, visto a correcção sintáctica conferir alguma estrutura.
3. O facto de a correlação de gama longa de alguns dos elementos do grupo C (preserva a sintaxe e preserva a distribuição) apresentar valores fora do intervalo da aleatoriedade,

ainda que mais baixos do que os verificados no grupo dos compiladores, era de esperar, visto a imposição conjunta da correcção sintáctica e da distribuição de palavras obriga a uma maior aquisição de estrutura da série gerada.

4. Ainda para o grupo C, os programas em que a correlação de gama longa é preservada, disporem de uma estrutura local semelhante à dos programas compiladores correspondentes, era igualmente de esperar, visto que a sua estrutura se aproxima de uma estrutura sintáctica correcta e funcionalmente válida.
5. Embora a correlação de gama longa esteja presente com um valor elevado nos programas com estrutura funcional, também pode surgir em programas sem essa estrutura desde que gerados com limitações que obriguem a uma aproximação da estrutura funcional.

Atendendo ao tamanho dos programas da amostra sentimos necessidade de verificar os resultados obtidos em séries mais longas e utilizando um alfabeto com menos elementos. Na secção 5.5. apresentamos essa análise.

5.5. Regularidade em ficheiros objecto

Nesta dissertação analisou-se a estrutura sintáctica e funcional na sua forma mais simples. Centrámos o nosso estudo nos elementos básicos com significado da linguagem, válidos em todas as implementações. Os elementos portadores de significado são as palavras-chave.

São conhecidas algumas abordagens diferentes da que apresentamos para o cálculo da correlação de gama longa em textos [Schenkel,A.92] e algoritmos codificados em linguagens de programação [Kokol,P.99]. A especificação sobre a qual se vai avaliar esta medida é o programa objecto. Estas abordagens revelam duas grandes vantagens:

Podem ser comparados programas codificados em linguagens diferentes, porque a técnica aplica-se sobre uma sequência de zeros e de uns, qualquer que tenha sido a linguagem usada na codificação.

Como um programa objecto é muito maior do que o programa fonte correspondente, esta técnica permite medir programas que não teriam, enquanto programas fonte, o tamanho suficiente para a obtenção da correlação de gama longa com validade.

Exemplo 5.5.1.: Os tamanhos das várias versões de um pequeno programa fonte, abaixo listado, mostram essas diferenças.

```
---------------------
main()
{
printf("Ola\n");
}
-----------------------
```

Este ficheiro tem o tamanho de 31 Bytes.
O ficheiro correspondente em código objecto tem o tamanho de 832 Bytes.
O ficheiro correspondente executável num computador Sun-Sparc, com sistema operativo Sun-Os, tem o tamanho de 23230 Bytes. A Maior parte do programa executável são rotinas, dados do ambiente e funções executáveis da biblioteca. ⊗

A abordagem diferente, a que nos referimos, é feita sem atender aos elementos com significado. Os autores [Schenkel,A.92;Kokol,P.99], consideram que, para sequências longas, os resultados são os mesmos, mas não oferecem nenhuma justificação teórica para tal.
Utilizando a medição da correlação de gama longa sobre a sequência das palavras-chave não é possível medir, em separado, cada um dos programas que constituem os compiladores. Os programas escritos pelos autores são sequências curtas, que não atingem a dimensão necessária à validade de cálculo dessa estatística.
Medir os programas obtidos com os geradores de análise lexica (*lex*), os programas obtidos com os geradores de análise sintáctica (*yacc*) e os programas escritos pelos alunos, permitiria reforçar a validade da correlação de gama longa como medida da estrutura. Tratando-se de programas que sabemos detêm estruturas diferentes, se a medida confirmar esta realidade a sua validade é reforçada.

Seguimos a seguinte estratégia para obter medidas válidas para todos os programas que constituem cada compilador:
Estabelecemos uma relação entre as medidas calculadas sobre a sequência das palavras-chave que representam cada compilador, e as medidas calculadas sobre as sequências de zeros e uns que constituem o programa compilado de cada compilador.
Obtida a indicação de que existia uma relação linear entre as duas, concluímos que a partir da medida calculada sobre a sequência de zeros e uns era possível obter a outra, calculada sobre

a sequência das palavras-chave.
Assim, calculamos o passeio aleatório do método descrito na secção 4.3.4.2. usando a sequência de zeros e uns correspondentes ao programa objecto e transformando cada zero em menos um para o código ser centrado. Na sequência estão incluídas, além da estrutura sintáctica e semântica já descritas, a estrutura de controlo do pré compilador e a estrutura de controlo de execução do programa incluindo bibliotecas.

Utilizando um método de identificação da CGL para ficheiros objectos, adaptado do método de Peng, Peter Kokol [Kokol,P.01] encontrou valores característicos para algoritmos codificados em Fortran e valores característicos para algoritmos codificados em Pascal.
Calculamos a correlação de gama longa usando esse método para o primeiro grupo principal da amostra. Obtivemos resultados registados na tabela 5.5.1. Podemos observar para cada compilador os seguintes valores:

- calculado sobre o ficheiro objecto registado na coluna "compilador",
- para o ficheiro objecto do módulo de análise léxica lex.yy.c, gerado pelo analisador léxico lex, registado na coluna "lex.yy.c",
- para o ficheiros objecto do módulos zeta.tab, gerado através do analisador sintáctico yacc, registado na coluna "zeta.tab.c",e,
- para uma média da medida dos restantes programas escritos por cada equipa, registado na coluna "Méd.outros".

Os valores da correlação de gama longa no programa objecto, usando esta última metodologia, são muito regulares em toda a mostra, tal como acontecia com a análise usando o método das palavras-chave.
Todavia, α tem um valor médio menor do que os calculados sobre o programa fonte, o que quer dizer que as estruturas nos diversos níveis (sintácticas e funcionais, existentes nos ficheiros de pré-compilação e de compilação) são menos regulares.

Se procurarmos os valores correspondentes, calculados pelo método das palavras-chave, encontramos que os dois valores são directamente proporcionais com uma relação traduzida pela equação:

$$y = 0.18\,x + 0.57$$

em que y é a medida obtida pelo método das palavras-chave e x é a medida obtida no programa objecto.

Esta equação foi obtida pelo método dos mínimos quadrados com $R^2 = .82$, que representa um bom ajustamento. Este resultado reforça a validade das medidas obtidas sobre o programa fonte.

Na Figura 5.5.1. visualiza essa relação.

As três últimas colunas da tabela 5.5.1. mostram a diferença da correlação de gama longa conforme a autoria dos programas:

- Os programas gerados pelo analisador léxico tomam um conjunto de valores com média de 0,7169 e muito pouco dispersos.
- Os programas gerados pelo analisador sintáctico tomam um conjunto de valores com média de 0,7393 e muito pouco dispersos.
- As pequenas variações do alfa nas componentes geradas pelos analisadores léxico e sintáctico são devidas ao código inserido, pelo programador, nos locais onde são identificados emparelhamentos ("*handlers"*). Em geral, o código inserido pelo programador no ficheiro de especificação yacc é maior que o código inserido no ficheiro de especificação do lex.
- A média da correlação de gama longa para os programas escritos pelos programadores é 0,6510, muito mais baixo do que os anteriores e com valores muito mais dispersos. Alguns dos módulos apresentam valores de alfa quase aleatórios, o que parece entrar em contradição com os valores de alfa dos compiladores, listados na figura 5.4.4.2. Tal contradição é aparente, na medida que os módulos, isolados, representam apenas parte de funcionalidades. Só na ligação ("*link"*) é que emerge uma estrutura, medida por valores mais elevados de alfa. Esta observação reforça a tese de que os programas são modelados por sistemas complexos.

No gráfico 5.5.2. estão representados os três grupos mencionados.

A regularidade dos programas gerados é evidente.

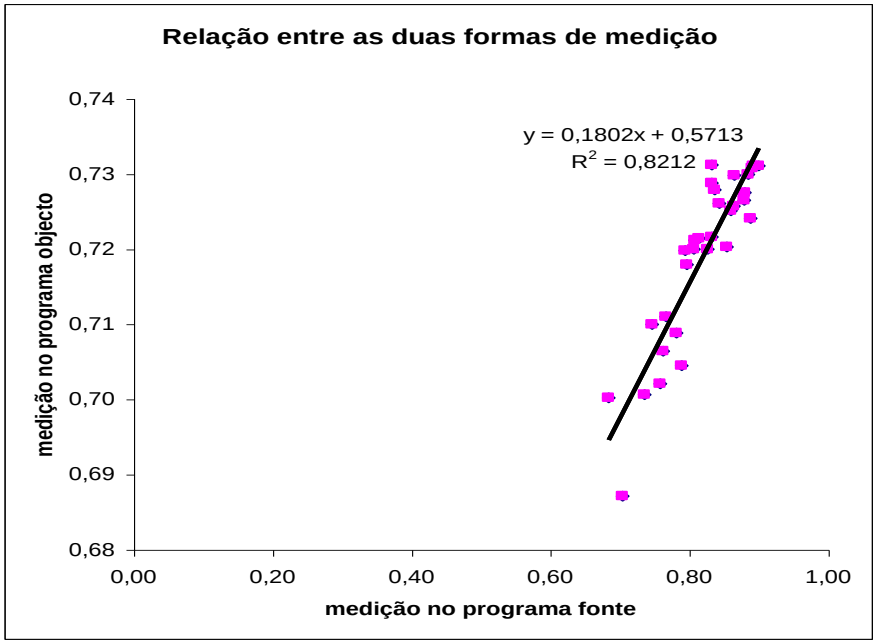
A variação para os programas escritos por programadores também está patente.

A tabela 5.5.2. contém os valores da correlação de gama longa para os módulos escritos pelos programadores para cada compilador. Os dados em falta correspondem a valores de alfa para os quais não foi encontrado ajustamento.

Nota-se a grande variação de valores, muitos deles perto do valor 0,5, que indica aleatoriedade.

Tabela 5.5.1. - Valores de correlações de gama longa para programas com a mesma funcionalidade calculados sobre o programa objecto.

Nº de ordem	Compilador	lex. yy.c	zeta.tab.c	Méd.outros
1	0,7308	-	-	-
2	0,7214	0,7183	0,7356	0,6809
3	0,7241	0,7274	0,7384	0,8061
4	0,7216	0,7131	0,7570	0,6601
5	0,7200	0,6992	6,7529	0,6985
7	0,6871	0,6967	0,7351	0,5374
6	0,7312	0,7571	0,7044	0,7577
8	0,7288	0,7312	0,7348	0,5403
9	0,7198	0,7108	0,7562	0,6509
10	0,7311	0,7138	0,7412	0,5238
11	0,7006	0,7303	0,7509	0,6907
12	0,7203	0,7009	0,7513	0,5677
13	0,7261	0,7163	0,7106	0,5281
14	0,7021	0,7260	0,7132	0,7601
15	0,7100	0,7221	0,7525	0,5801
16	0,7311	0,7198	0,7189	0,5817
17	0,7110	0,7121	0,7418	0,5886
18	0,7299	0,7102	0,7248	0,7715
19	0,7200	0,7135	0,7167	0,7755
20	0,7088	0,7155	0,7507	0,7890
21	0,7064	0,7362	0,7227	0,6526
22	0,7251	0,7063	0,8049	0,5765
23	-	-	-	0,5810
25	0,7298	0,7133	0,6812	0,7331
26	0,7275	0,6951	0,7327	-
27	0,7045	0,7027	0,7662	0,7382
28	0,7265	0,6987	0,7031	0,5595
29	0,7212	0,7245	0,7257	0,5440
30	0,7279	0,7235	0,7295	0,6104
31	0,7179	0,7217	0,7811	0,7396
32	-	0,7276	0,7464	0,7891
33	0,7257	0,7159	0,7254	0,5931
34	-	0,7157	0,7321	0,5521
35	0,7002	-	-	0,5857
36	-	0,7259	0,8186	0,6886



.

Figura 5.5.1. Relação entre os dois tipos de medidas

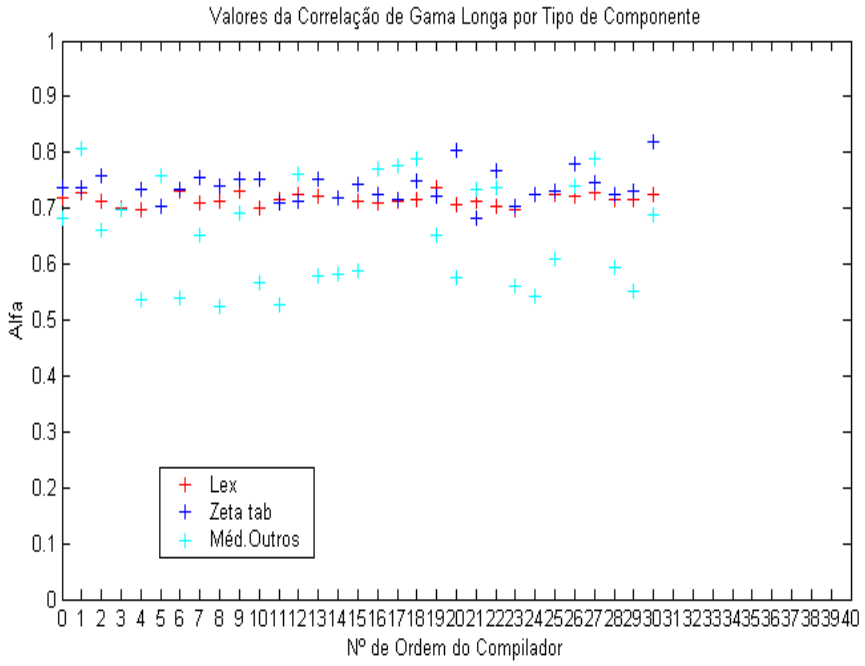


Figura 5.5.2. valores da CGL para as várias componentes dos programas do primeiro grupo fundamental da amostra (compiladores), calculados nos executáveis

Tabela 5.5.2. - Valores de correlações de gama longa para os módulos dos programas com a mesma funcionalidade calculados sobre o programa objecto.

Projecto	Média	Desvio	Correlação de gama longa de módulos escritos pelos autores							
1	0,6809	0,0336	0,7046	0,6572						
2	0,8061	0,0352	0,8184	0,8336	0,7664					
3	0,6601	0,0632	0,6716	0,5923	0,7370	0,5923	0,5985	0,6600	0,7533	0,6758
4	0,6985	0,1691	0,5720	0,8800	0,4792	0,8248	0,7364			
5	0,5374	0,0272	0,5187	0,5639	0,5574	0,5096				
7	0,7577	0,0401	0,7278	0,8029	0,7106	0,7992	0,7766	0,7293		
6	0,5403	0,0627	0,5536	0,5001	0,4746	0,6382	0,5349			
8	0,6509	0,0205	0,6440	0,6347	0,6740					
9	0,5238	0,0552	0,4795	0,5223	0,4914	0,6020				
10	0,6907	0,2457	0,8462	0,7467	0,5609	0,7881	0,7528	0,8392	0,0000	0,8160
11	0,5677	0,0611	0,5285	0,5121	0,5825	0,6475				
12	0,5281	0,0424	0,4711	0,5241	0,5699	0,5473				
13	0,7601	0,0835	0,7589	0,8343	0,7956	0,6187	0,7932			
14	0,5801	0,0409	0,5512	0,6090						
15	0,5817	0,0688	0,5937	0,5115	0,5499	0,6718				
16	0,5886	0,0766	0,6890	0,5244	0,6076	0,5334				
17	0,7715	0,0469	0,7642	0,7357	0,7465	0,8396				
18	0,7755	0,0146	0,7858	0,7652						
19	0,7890	0,0369	0,7827	0,8053	0,8263	0,7950	0,7804	0,7461	0,8069	0,7212
20	0,6526	0,0249	0,6702	0,6350						
21	0,5765	0,0262	0,5579	0,5950						
22	0,5810		0,5810							
24	0,7022	0,0653	0,6533	0,6769	0,7763					
25	0,7331	0,0941	0,8102	0,7364	0,7714	0,7408	0,7256	0,7888	0,5115	0,7799
26	0,7382		0,7382							
27	0,5595	0,0583	0,6007	0,5182						
29	0,5440	0,0234	0,5171	0,5561	0,5589					
30	0,6104	0,0324	0,5994	0,6468	0,5849					
31	0,7396	0,0175	0,7272	0,7520						
32	0,7891	0,0204	0,7926	0,7672	0,8076					
33	0,5931	-	0,5931							
34	0,5521	0,0477	0,5183	0,5858						
35	0,5857	0,0110	0,5775	0,5750	0,5950	0,5953				
36	0,6886	0,0605	0,7314	0,6458						

Os resultados suscitam os seguintes comentários.

1. A regularidade encontrada nos componentes gerados pelos geradores sintáctico e léxico é um resultado esperado, visto que os geradores geram programas sempre segundo as mesmas regras e com uma estrutura muito semelhante.

2. A dispersão encontrada nos valores de α nas componentes escritos pelos alunos é um resultado esperado, visto que cada programador tem o seu próprio estilo e, como tal, para cada uma a estrutura do programa que produz é diferente.
3. Uma explicação possível para que alguns módulos tenham valores quase aleatórios é o facto de, por isolados, representarem partes de funcionalidades que só depois de ligadas ("linked") representam funcionalidades de compilação da linguagem zeta.

5.6. Contribuição deste capítulo para a defesa da tese que é objecto desta dissertação

A contribuição deste capítulo para a defesa da tese que é objecto desta dissertação é a introdução de novos métodos de medição da estrutura do software e a sua aplicação à amostra seleccionada.

Verificamos que as entropias de Shanon e Rényi caracterizam de forma incompleta a estrutura de um algoritmo e a correlação de gama longa é uma métrica da estrutura sintáctica de um programa escrito em C. A correlação de gama longa está presente na generalidade dos programas com estrutura funcional, podendo entretanto, também pode surgir em programas sem essa estrutura. A correlação de gama longa é robusta em relação à medição sobre programas fonte e objecto.

6. Classificação de programas

....A classificação explica porque observamos padrões reconhecidos de comportamento global ou categorias dentro dos quais dois indivíduos ou acontecimentos nunca são exactamente iguais. Duas folhas de feto nunca são iguais, mas não deixam de ser, apesar disso, folhas de feto..... O que é importante é conhecer a verdadeira identidade das coisas.....
Ralph D. Stacey

6.1. Introdução

Neste capítulo descrevemos a tentativa de discriminação dos programas do primeiro grupo fundamental da amostra (compiladores) em relação ao número de funcionalidades que implementam.

Para tal, começámos por efectuar uma análise taxonómica, cujos resultados evidenciam que o grupo é homogéneo (os seus elementos não são diferenciáveis), quando medido pelo conjunto das métricas apresentadas no capítulo 4. O uso dessa técnica não nos permite, consequentemente, efectuar a discriminação pretendida.

Seguidamente, procurámos uma estratégia alternativa, usando a técnica de identificação de intrusão de vírus que fundamenta a construção do chamado "*self*" de cada compilador. Verificamos que os "*selfs*" dos dois grupos com maior e menor avaliação não se podem considerar diferentes. Tal como a análise taxonómica, esta técnica não nos permite efectuar a distinção entre compiladores com diferentes números de funcionalidades.

Concluímos, então, que as técnicas de agrupamento de dados, não são sensíveis ao número de funcionalidades que um compilador implementa.

6.2. A análise taxonómica

A análise taxonómica, também conhecida como análise classificativa ou de segmentação ("*clusters"*), é uma análise da estatística multivariada [Hartigan,J.75; Arabie,L.96;Caldas,J.91]. Esta técnica exploratória usa valores dos atributos das entidades para as classificar.

A análise taxonómica agrupa dados semelhantes em grupos mutuamente exclusivos (*"clusters"*) de tal maneira que a semelhança entre elementos do mesmo grupo é maximizada e a semelhança entre elementos de grupos diferentes é minimizada.

São exemplos de utilização da análise taxonómica para classificação, a catalogação do reino animal usada pelos biólogos, a caracterização de grupos de clientes através de padrões de compras, a caracterização de padrões de documentos na Web e a análise de tipos de terramotos a partir de bases de dados sismológicas. ⊗

O seu uso é indicado quando é necessário dividir um conjunto grande de entidades em subconjuntos, e não se conhecem critérios de classificação, dessas entidades, baseados nos valores disponíveis dos seus atributos.

São referidos três tipos de metodologias de classificação para obter a separação em grupos [Jain,A.88; Caldas,J.91]: técnicas de optimização da partição, técnica hierárquicas e técnicas de densidade. Todas elas usam um conceito de distância entre os elementos.

- As técnicas de optimização da partição separam os dados em partições, de tal maneira que a partição optimize um dado critério de distância.
- As técnicas hierárquicas constroem uma sequência de grupos aninhados, com um grupo que inclui todos os outros, no topo da hierarquia, e tantos grupos quantos os elementos do grupo, na base da hierarquia.
- As técnicas da densidade, baseadas na representação de elementos por pontos, conduzem à formação de grupos a partir de nuvens de pontos.

A técnica da partição é indicada quando se sabe que a distribuição dos elementos é tal, que não existem elementos de um dado grupo que estejam menos distantes dos elementos de outros grupos do que dos elementos do seu próprio grupo [Jain,A.88]. A técnica da densidade

pode ser aplicada se existir uma distribuição de pontos com áreas de grande densidade. Se tal não for verificado, a técnica hierárquica é a mais indicada [Jain,A.88].
Nesta dissertação usamos a técnica hierárquica, porque não podemos formular nenhuma das duas anteriores hipóteses sobre a distribuição dos programas do primeiro grupo fundamental da amostra.

De seguida descrevemos brevemente o algoritmo da análise taxonómica, usando a técnica hierárquica.

6.2.1.Algoritmo da análise taxonómica

Para executar a análise taxonómica, usando a técnica hierárquica, é necessário executar os seguintes passos:

1. definir que atributos caracterizam apropriadamente os objectos a classificar e a função de distância utilizada para os discriminar,
2. usando os valores dos atributos seleccionados como coordenadas dos objectos, gerar uma matriz de semelhança dos objectos a classificar,
3. criar a árvore hierárquica,
4. decidir sobre o número de grupos em que pretendemos separar os objectos,
5. obter a separação dos objectos e validá-la,

Descrevem-se de seguida os cinco passos com mais pormenor:

1. Selecção de atributos

A escolha do atributo será orientada pelo conhecimento disponível do domínio dos objectos, porque não existe descrita uma metodologia para a selecção dos atributos.
Habitualmente determina-se o número de atributos que permita que a análise, que é, computacionalmente muito pesada, possa ser efectuada no equipamento disponível. Seguidamente, de entre os atributos escolhem-se, os que conduzem, a classificações diferentes dos dados disponíveis. Muitas vezes simula-se o uso de vários grupos de atributos e escolhe-se, para utilizar na análise, o grupo que fornece o resultado mais de acordo com o que se passa no mundo real.

2. Geração da matriz de semelhança

Para gerar a matriz de semelhança, também chamada de distâncias, é necessário definir uma medida de distância entre os objectos.
O vector de distâncias entre n objectos é uma matriz $n \times n$ simétrica de diagonal igual a 0. Pode ser escolhida qualquer função de distância. Há três tipos de medidas de distância frequentemente utilizados nesta análise [Caldas,J.91]: coeficientes de correlação, medidas de distância euclidianas e coeficientes de associação.

3. Determinação do número de grupos

A decisão do número de grupos que se pretende formar é tomada atendendo ao número de elementos e à necessidade de obter grupos que traduzam uma discriminação entre os elementos. Frequentemente, o número óptimo de grupos obtém-se por simulação, até encontrar o número de grupos que, nem separa demais e por isso não traduz a semelhança requerida, nem agrega tanto a ponto de não discriminar.

4. Criação da árvore hierárquica

Depois de fixado o número de grupos é necessário distribuir os objectos pelos grupos. Existem dois tipos de métodos para a obtenção da árvore hierárquica [Caldas,J.91]: os métodos de ligação "*single-link*"e os métodos de ligação "*complete-link"*.
Estes métodos conduzem a uma hierarquia de partições $H=\{P_1,P_2,....,P_n\}$, do conjunto O de objectos em N grupos em que para cada par de partições, P_i e P_{i+1} cada grupo de P_{i+1} está incluído num grupo de P_i.
O algoritmo seguido nestes métodos é o seguinte:

1. Seja P_n, a partição do conjunto O em N grupos, cada um com um objecto.
2. No caso do método de ligação "*single-link*", agregam-se os dois grupos da partição corrente cuja distância mínima entre objectos, um de cada grupo, seja mínima. No caso do método de ligação "*complete-link"*, agregam-se os dois grupos da partição corrente cuja distância máxima entre objectos, um de cada grupo, seja mínima.

3. Repete-se este passo até todos os objectos pertencerem ao mesmo grupo.

Cada novo grupo criado é representado por um elemento novo, o centróide ou o medóide. As coordenadas desse novo elemento são a média das coordenadas dos elementos, no caso do centróide ou, no caso do medóide, a média ponderada, pela probabilidade, das coordenadas dos elementos.

A árvore hierárquica obtida é uma sucessão de ligações de dois elementos. Cada elemento pode pertencer ao conjunto inicialmente fornecido para classificação, ou pode ter sido nele incluído como representante de uma ligação efectuada.

O valor da ligação é atribuído de acordo com a ordem dos elementos que liga. Se liga elementos do conjunto inicial, o valor da ligação é um. Se liga elementos provenientes de ligações de elementos iniciais o seu valor, será dois, se liga elementos ligados por uma ligação de valor dois com outros de valor dois, ou menor, a sua ligação será de valor três, e assim sucessivamente. Cada grupo constituído fica assim caracterizado por um centro de grupo referenciado por um conjunto de coordenadas e por um valor da ligação

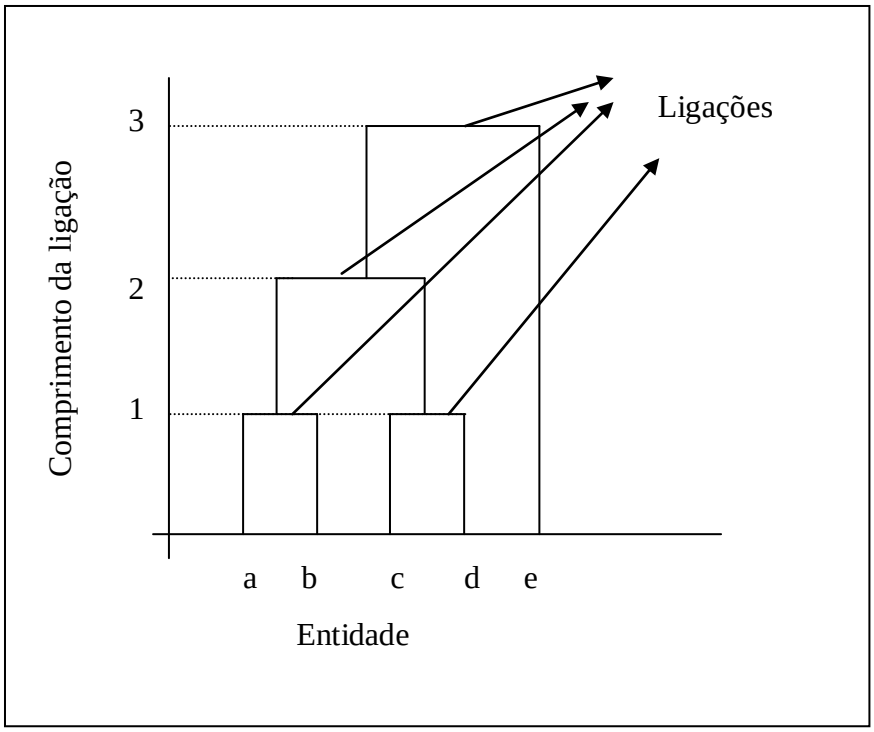


Figura 6.2.1.1 Dendograma

O resultado final desta análise taxonómica é o conjunto das coordenadas dos centros dos grupos construídos, a lista dos elementos de cada grupo e a árvore hierárquica que os liga.
A representação gráfica do processo de construção dos grupos é feita através de um dendograma [Matlab,01;Caldas,J.91].

Um dendograma é um gráfico da árvore hierárquica onde são representados os valores da ligação entre as variáveis. Um exemplo de um dendograma é representado na figura 6.2.1.1., em ordenadas são representados os valores de cada ligação, em abcissas os elementos.

A aplicação dos dois métodos conduz a resultados diferentes. Enquanto o método da ligação *"single-link"* tende a agrupar entidades bastantes dissemelhantes devido ao encadeamento de fusões que produz, mas garante uma boa separação entre os grupos, o método de ligação "*complete-link"* garante que os elementos agrupados são mais semelhantes dois a dois, mas pode separar deficientemente os grupos.

5. Separar e validar

Para verificar se a árvore hierárquica construída representa agrupamentos com significado, calcula-se um coeficiente de correlação entre o agrupamento representado pela árvore e a proximidade entre os dados representada pela matriz das distâncias. Esse coeficiente, designado por coeficiente de correlação "*cophenetic",* toma valores entre zero e um e traduz uma correlação aceitável para valores maiores ou iguais a 0.65.
Uma vez validada a árvore hierárquica construída, usam-se os valores das ligações para separar os elementos no número de grupos que se fixou previamente no passo três.
Existem duas maneiras de calcular o coeficiente de correlação:

Uma baseia-se no cálculo do coeficiente de inconsistência [Matlab,01] e constrói a divisão natural do conjunto de dados. O coeficiente de inconsistência representa o valor da comparação do comprimento de cada ligação com os comprimentos das ligações vizinhas que na árvore estão abaixo dela. Se o valor da ligação é próximo da média dos que estão abaixo, então esses elementos pertencerão ao mesmo grupo. Se não for, encontra-se inconsistência que representa a separação natural dos grupos. O coeficiente de inconsistência varia entre zero

e um e no cálculo dos grupos usando o Matlab [Matlab,01] pode ser especificado quando se invoca a função que cria os grupos.

Outra maneira de formar os grupos é decidir, à priori, quantos grupos se pretendem formar, nesse caso, é necessário observar o dendograma e cortá-lo por uma linha paralela ao eixo do *x* de forma que as linhas de ligações sejam interceptadas no número de grupos escolhidos.

Na figura 6.2.1.2., é representada, respectivamente, a formação de dois e de três grupos, usando o método da decisão à priori.

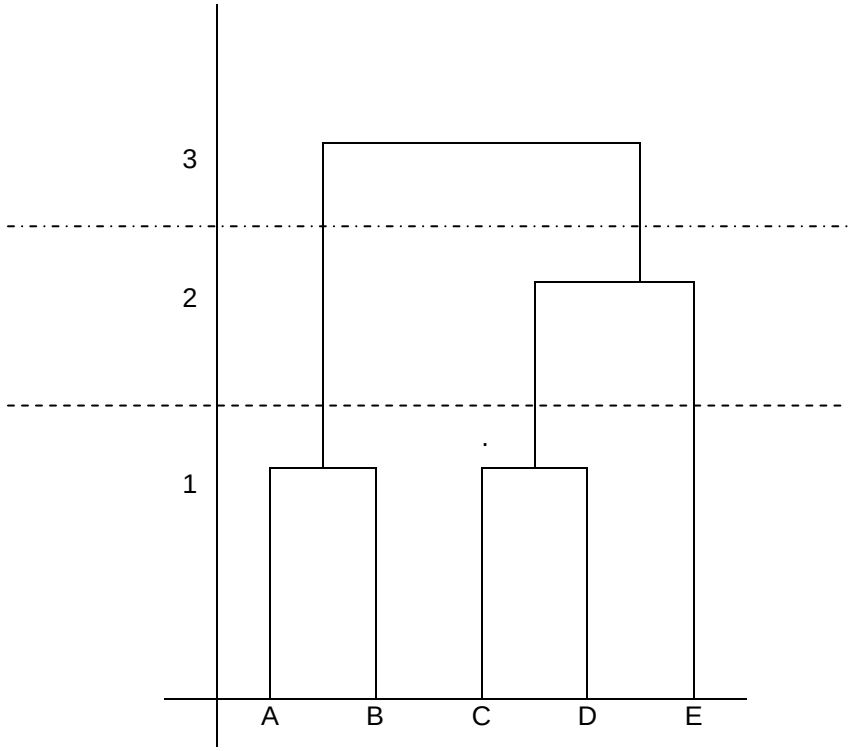


Figura 6.2.1.2. Formação de grupos

A traço ponto, está representada a linha que corta o dendograma e horizontalmente em duas ligações e forma dois grupos: um incluindo os elementos A e B, outro incluindo os elementos C, D e E.

A tracejado, está representada a linha que corta o dendograma horizontalmente em três ligações e forma três grupos: um incluindo os elementos A e B, outro incluindo os elementos

C, D e um terceiro incluindo o elemento E.

Na secção seguinte apresentaremos a análise da amostra dos programas do primeiro grupo fundamental da amostra.

6.3. Análise taxonómica dos programas do primeiro grupo fundamental da amostra

Não foi possível, usando qualquer uma das medidas calculadas no capítulo 5, obter uma separação dos compiladores, de acordo com a classificação atribuída a cada um pelo docente da disciplina. A classificação, recordamos, era feita de acordo com o número de funcionalidades implementadas em cada compilador.

O processo de classificação dos programas teve em conta os elementos sintácticos aceites pelo compilador implementado. A classificação de cada membro do grupo foi feita com base em provas individuais, por forma a verificar a homogeneidade dos grupos de alunos.

A avaliação do programa foi realizada utilizando uma bateria de testes funcionais. Foram empregues 15,10,5,10 testes para quatro classificações positivas: a classificação 1 corresponde a uma avaliação no intervalo de 10-13 valores, a classificação 2 no intervalo 14-15 valores, a classificação 3 no intervalo 16-17 valores e a classificação 4 no intervalo 17-20 valores, respectivamente. No enunciado do problema (anexo 5.2.2.1.), na secção Objectivos, é indicado o tipo de funções que o compilador deve transcrever para ser classificado em cada um dos intervalos. Existe ainda a classificação 0, atribuída a grupos que submeteram exclusivamente código que implementa funções fornecidas pelo docente da disciplina. Estes programas, embora tenham sido avaliados, porque representam código compilado que implementa transcrição de funções, obtiveram classificação de 0 valores por não representarem trabalho do aluno.

O conjunto de testes de cada classificação verifica se o compilador construído implementa a transcrição do tipo de funcionalidades valorizadas para esse intervalo. Por exemplo, os testes utilizados para avaliar o nível inicial entre 10 e 13 valores (transcrição pelo compilador de expressões aritméticas, expressões lógicas, testes (if), funções escrever e ler), consistiram na submissão do compilador avaliado à compilação de quinze programas diferentes para os quais eram conhecidos os resultados. Os resultados dessa submissão, por comparação com os resultados esperados, forneceram a indicação se o compilador compila correctamente a função

indicada.
Um compilador com uma classificação superior a um outro implementa maior número de transcrição de tipos diferentes de funções.

Todos os grupos são constituídos por programas com uma estrutura sintáctica correcta em linguagem C, uma vez que foram compilados com sucesso. Detêm ainda uma estrutura funcional, uma vez que implementam funcionalidades indicadas nos requisitos.

Nesta secção analisamos se as seis medidas em conjunto, utilizadas como coordenadas em R^6, permitem a criação de grupos homogéneos coincidentes com os grupos formados por programas com classificações idênticas. Escolhemos estas seis medidas porque cada uma mede características diferentes da sequência das palavras-chave de cada compilador.
Para realizar a análise taxonómica usamos um ficheiro, correspondente à tabela 6.3.1. construímos um programa Cluster listado (anexo 5.3.1) que recebe um ficheiro com os dados respeitantes aos atributos escolhidos para caracterizar cada programa e o número de grupos onde se pretende separar a amostra e retorna a uma lista contendo o grupo de cada programa.
Calculamos duas separações de acordo com o processo descrito na secção 6.2. Para ambas, usamos a distância euclidiana e a separação em grupos impondo um número fixo de grupos a criar. Para uma delas usamos o método de ligação *"single-link"* e para a outra o método de ligação "*complete-link"*.

Eliminamos o uso das medidas de correlação, como medidas de distância, uma vez que, dois compiladores diferentes podem ter correlação elevada, apenas porque contêm partes comuns geradas automaticamente. Assim sendo, se esta medida fosse usada, não poderíamos garantir que a classificação obtida indicasse semelhança de estrutura.
Usamos, então, a distância euclidiana, uma vez que esta medida é robusta, se a amostra for pouco dispersa, como é o caso em análise [Caldas,J.91].

Primeiro calculamos a separação dos compiladores fixando em cinco o número de grupos, a escala de avaliação.

Tabela 6.3.1 Medidas dos programas do primeiro grupo fundamental da amostra

	Entropia de Shannon	α	**Entropia de Rényi de expoente:**			
			1/3	1/8	3	8
1	3,8629	0,8906	4,5030	4,3650	-0,0030	-0,00000001
2	3,8968	0,8141	4,5455	4,4853	-0,0050	-0,00000002
3	3,8566	0,8880	4,6609	4,5968	-0,0040	-0,00000001
4	3,8313	0,8327	4,7043	4,7458	-0,0080	-0,00000007
5	3,9753	0,8265	4,0636	4,1384	-0,0100	-0,00000022
6	3,4748	0,7041	4,6567	4,7152	-0,0080	-0,00000007
7	3,9020	0,8326	4,6686	4,6483	-0,0060	-0,00000007
8	3,9553	0,8326	4,6074	4,5684	-0,0060	-0,00000006
9	3,8272	0,7944	4,6177	4,5890	-0,0050	-0,00000002
10	3,9296	0,8926	4,5665	4,5548	-0,0060	-0,00000005
11	3,6310	0,7355	4,5078	4,5074	-0,0280	-0,00001122
12	3,8781	0,8544	4,5927	4,6284	-0,0090	-0,00000019
13	4,0555	0,8428	4,6245	4,6564	-0,0360	-0,00002732
14	3,5588	0,7579	4,4334	4,4041	-0,0120	-0,00000105
15	3,9285	0,7467	4,6288	4,6247	-0,0060	-0,00000004
16	3,8748	0,8997	4,5763	4,5848	-0,0050	-0,00000002
17	3,6844	0,7665	4,5007	4,4780	-0,0100	-0,00000064
18	3,9142	0,8853	4,6265	4,6159	-0,0060	-0,00000004
19	3,8092	0,8067	4,6235	4,6050	-0,0080	-0,00000022
20	3,8373	0,7813	4,5762	4,5813	-0,0080	-0,00000010
21	3,8373	0,7620	4,6182	4,5986	-0,0060	-0,00000008
22	3,8892	0,8261	4,5897	4,5366	-0,0130	-0,00000265
23	3,7141	0,7810	4,2507	4,2583	-0,0110	-0,00000081
24	3,4713	0,7871	4,5772	4,4983	-0,0110	-0,00000064
25	3,6593	0,8595	4,5064	4,4958	-0,0080	-0,00000012
26	3,7558	0,8647	4,6128	4,5712	-0,0050	-0,00000003
27	3,9330	0,8798	4,6071	4,5741	-0,0080	-0,00000024
28	3,8085	0,7892	4,6530	4,6129	-0,0070	-0,00000010
29	3,8719	0,8792	4,4497	4,4511	-0,0120	-0,00000081
30	3,6542	0,8080	4,4539	4,3631	-0,0180	-0,00000256
31	3,3067	0,8368	4,6251	4,6122	-0,0070	-0,00000008
32	3,8345	0,7968	4,5103	4,5024	-0,0090	-0,00000034
33	3,7526	0,8637	4,6335	4,6295	-0,0060	-0,00000008
34	3,9427	0,8401	4,8532	5,3876	-0,2530	-0,00279300
35	3,9180	0,6841	4,6415	4,5833	-0,0070	-0,00000009
36	3,7706	0,7702	4,5948	4,5481	-0,0080	-0,00000010

A distribuição de elementos por grupos, igual para os dois tipos de métodos de construção dos

grupos, mostra que em relação ao conjunto das medidas utilizadas, a amostra é muito homogénea, visto 32 dos 36 elementos formarem o grupo um e os restantes grupos apenas terem um elemento.
Para estes grupos o coeficiente de correlação *cophenetic* tem o valor de 0.68 o que traduz uma correlação aceitável entre a árvore construída e a matriz de semelhança calculada.

Na tabela 6.3.2. e 6.3.3.estão registados esses resultados.

Tabela 6.3.2. Número de elementos em cada grupo

Grupo	Nºde elementos
1	32
2	1
3	1
4	1
5	1

Tabela 6.3.3. Número de elementos de cada classificação

Classificação	Nºde elementos
0	0
1	8
2	8
3	2
4	7

Comparando a distribuição fornecida pela análise taxonómica com a distribuição dos elementos por classificação obtida na disciplina, nota-se que as duas não coincidem.
Uma vez que, com cinco grupos não conseguimos obter resultados que discriminassem os programas considerados repetimos os cálculos para 8 grupos e obtivemos um resultado semelhante: 29 dos 36 elementos estão no grupo 1 e os restantes grupos têm novamente apenas um elemento.
Concluímos que, com a análise taxonómica, as seis medidas não discriminam os cinco tipos de compiladores estudados.

Tabela 6.3.4 Medidas dos programas com a mesma funcionalidade

Compilador	Classificação	Cluster para nº de cluster igual a 5	Cluster para nº de cluster igual a 8
1	0	1	1
2	4	1	1
3	0	1	1
4	2	1	1
5	3	3	6
6	4	2	5
7	0	1	1
8	2	1	1
9	1	1	1
10	0	1	1
11	3	1	1
12	2	1	1
13	2	4	7
14	2	1	1
15	4	1	1
16	2	1	1
17	2	1	1
18	2	1	1
19	2	1	1
20	1	1	1
21	0	1	1
22	1	1	1
23	2	1	4
24	2	1	2
25	0	1	1
26	1	1	1
27	1	1	1
28	4	1	1
29	0	1	1
30	1	1	1
31	1	1	3
32	4	1	1
33	1	1	1
34	4	5	8
35	4	1	1
36	0	1	1

A estrutura desses programas é muito semelhante na medida em que não só têm uma grande parte gerada por um mesmo gerador que implementa as mesmas funções e usa a mesma gramática mas também porque estes programas possuem outra parte escrita por alunos com conhecimentos e práticas idênticas da linguagem.

A parte implementadas pelos alunos é, nesta amostra, sempre menor que 10% do código total e é nessa parte que reside a diferença que origina a classificação. Os resultados traduzem esta realidade.

Na tabela 6.3.4 estão registados os dados que suportaram estes cálculos.

A utilização desta técnica, com esta amostra, não acrescenta nenhuma nova possibilidade de discriminação entre os compiladores. Indica-nos que, tal como as outras medidas analisadas em separado já o faziam, a amostra é homogénea.

Os resultados suscitam os seguintes comentários:

1. Estes valores indicam que a complexidade não depende das funcionalidades implementadas pelo programa.
2. A tese da independência da complexidade com o tamanho deve ser testada com outros sistemas de classificação. Tal é feito na secção 6.5..

Na secção seguinte descrevemos outra metodologia que usa a análise taxonómica para reconhecer a identidade de um algoritmo.

6.4. Técnicas de reconhecimento da identidade de um algoritmo

O problema de intrusão de vírus em computadores e a consequente modificação não autorizada de programas deu origem ao desenvolvimento de estudos com dois objectivos:

- evitar a actuação de vírus,
- reconhecer que um programa foi alterado por uma actuação de vírus.

Uma das técnicas que permitem o reconhecimento da alteração de um programa foi construída inspirando-se nos processos de defesa imunitária do organismo humano [Forest,S.94; D'Hacseller,76]. Essa técnica permite reconhecer o "*self*" e o "*other*" de um programa.
Define-se "*self*" como um programa que não foi alterado e "*other*" como qualquer programa diferente do "*self*". Se for apresentado como sendo o "*self*", quando detectado como "*other*", indica que houve alteração não autorizada.

Esta técnica é, como tal, um algoritmo de detecção de mudanças.
Nesta dissertação, com o fim de tentar distinguir programas que têm uma medida de estrutura semelhante e que, embora implementem o mesmo tipo de funcionalidade, não o fazem nem na mesma extensão, nem no mesmo modo, usamos a técnica de reconhecimento de identidade.
Como vamos ver de seguida, trata-se de um processo mais permissivo do que, por exemplo, a comparação de ficheiros usando o comando *diff* do sistema operativo Unix. O comando *diff* compara dois ficheiros de texto e retorna localizando as diferenças existentes nos dois ficheiros. Com a técnica de reconhecimento da identidade não é exigida a concordância absoluta, mas sim semelhança de dados de acordo com determinadas regras.

Escolhemos a técnica de reconhecimento de identidade, porque não se baseia, como a maioria das técnicas de detecção de modificações que conhecemos, no conhecimento da identidade dos vírus que podem alterar um programa. Constrói, antes, a identidade (o "*self*") do programa e permite compará-lo com outros que identifica como sendo idênticos ou diferentes. Se o resultado da comparação é o reconhecimento da diferença e se o algoritmo foi aplicado a um programa que se considerava não alterado, podemos concluir que aconteceu uma modificação não autorizada.
A técnica de reconhecimento de identidade detecta se sequências de instruções de um programa foram ou não mudadas, por ter sido alterado um conjunto de seus valores ou por terem sido acrescentados sequências com configuração diferente das iniciais. Este algoritmo não detecta sequências apagadas no programa.

Descrevemos de seguida a estratégia seguida na construção do "*self* " [Forrest,S.94]:
A partir de um programa considerado não corrompido é gerado um conjunto de detectores, que são usados para constantemente monitorizarem o programa e assinalarem quando este é alterado.
Define "*self*" como a sequência que representa o programa a proteger da alteração. Define "*other*" como qualquer outra sequência.
Para construir o conjunto de detectores divide-se a sequência do "*self*" em subsequências de tamanho igual.
Com as subsequências de tamanho igual, forma-se o conjunto, denominado *S*, das sequências do "*self*".

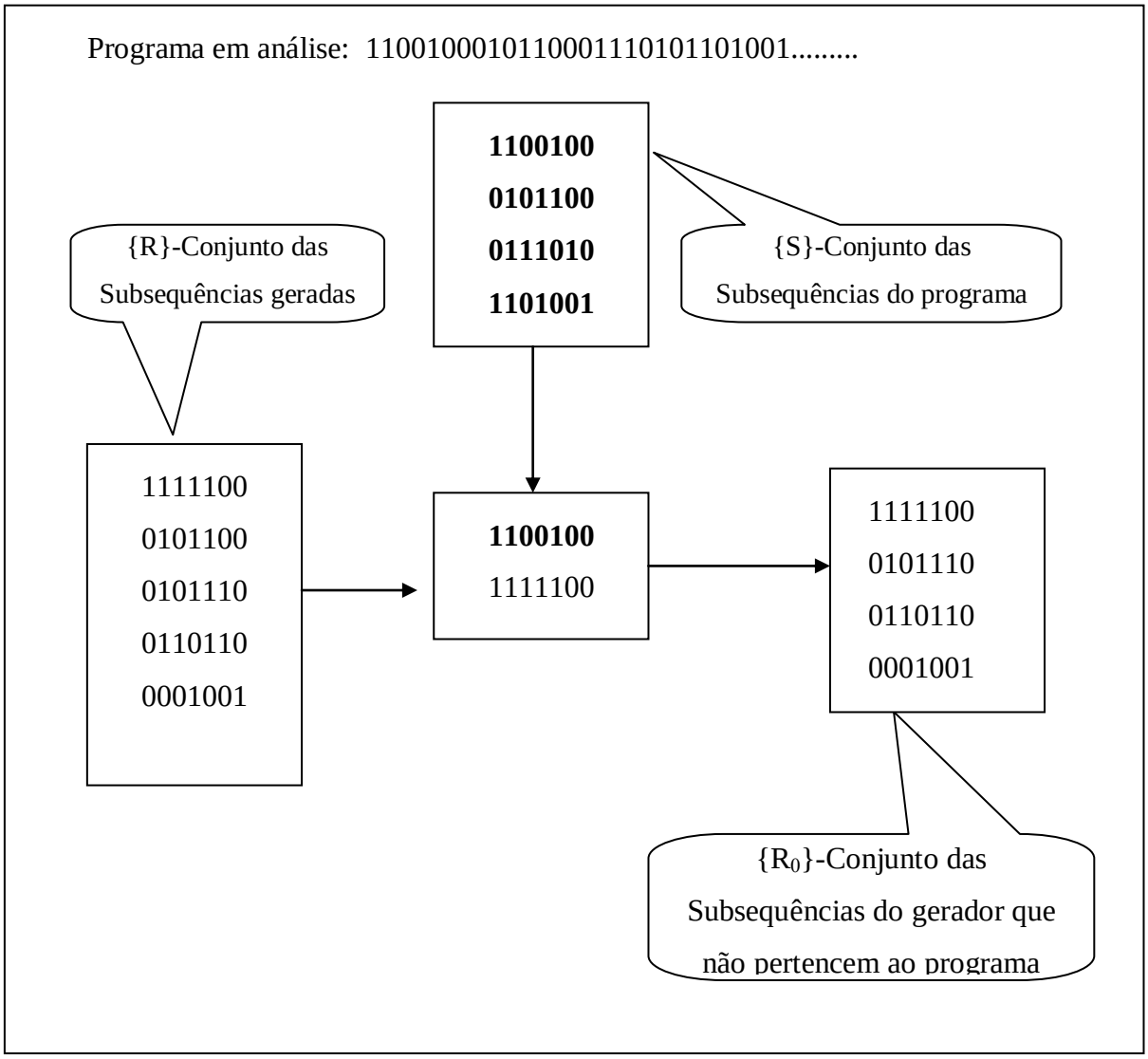


Figura 6.4.1. Construção do "*self*" e do "*other*".

Seguidamente geram-se aleatoriamente sequências que formam um conjunto denominado *Ro*. Esse conjunto é comparado com *S* e construído o conjunto *R* das subsequências de *Ro* que são diferentes de qualquer subsequência de *S*.

O conjunto *R* de sequências é o detector. É usado para ser comparado com o conjunto *S* e fazer a detecção do "*other*". Se é encontrado em *S* alguma sequência que exista em *R* quer dizer que o programa foi modificado e que é um "*other* ".

A regra da coincidência entre duas sequências de *S* e *R* pode ser muito ou pouco exigente. Será muito se se exigir que todos os símbolos de *S* coincidam completamente com todos os

símbolos de *R* e que surjam pela mesma ordem. É a coincidência perfeita.
Pode-se construir a coincidência de grau *r*, exigindo menos. Exige-se que as duas sequências coincidam pelo menos em *r* localizações contínuas. Esta regra pode ser relaxada exigindo cada vez um *r* menor.
Utilizando os conceitos da análise taxonómica pode exigir-se que as sequências coincidam se pertencerem aos mesmos grupos homogéneos.

6.5. Reconhecimento da identidade no primeiro grupo fundamental da amostra

Nesta secção analisamos, recorrendo à técnica de construção de um "*self*" descrita na secção 6.4., se os programas que pertencem aos grupos mais afastados, em termos de funcionalidade, dentro do primeiro grupo fundamental da amostra (grupo com classificação 0 e grupo com a classificação 4) se podem considerar diferentes.

O grupo dos programas do primeiro grupo fundamental da amostra (compiladores) que não implementavam nenhuma funcionalidade, além das já fornecidas como exemplos no enunciado, inclui os programas 1, 3, 7, 10, 21, 25 e 29. Todos eles foram classificados com zero.
O grupo dos compiladores que implementavam todas as funcionalidades, solicitadas pelo enunciado, inclui os programas 2, 6, 28, 32 e 36. Todos eles foram classificados com a classificação máxima.

Utilizamos as sequências numéricas (não obrigatoriamente injectivas), de valores entre -16 e 15, que representam a codificação das sequências das palavras-chave de cada compilador.
A partir de cada sequência, criamos quatro sequências com as subsequências de tamanho dois, três, quatro e cinco.
Exemplo 8.3.1. com a sequência 1, 4, -3, 0, 0, 10,–15, 8, -4, -4, 9, 7,..... construímos sequências:

de dois elementos: 1 4, 4 -3, 3 0, 0 0, 0 10, 10 –15, ...

de três elementos: 1 4 -3, 4 –3 0, 3 0 0, 0 0 10, 0 10 -15, 10 –15 8, ...

de quatro elementos: 1 4 –3 0, 4 –3 0 0, 3 0 0 10, 0 0 10 -15, 0 10 –15 8, 10 –15 8 -4, ...

de cinco elementos: 1 4 –3 0 0, 4 –3 0 0 10, 3 0 0 10 -15, 0 0 10 –15 8, 0 10 –15 8 -4, ... ⊗

Para cada compilador e para cada uma das quatro sequências criadas, calculámos 50 grupos homogéneos usando o programa Cluster listado (anexo 5.3.1) que recebe um ficheiro com as subsequências do tamanho escolhido e o número 50 e retorna a uma lista contendo cada subsequência e o grupo ("*cluster*") em que está incluída, o centro e número de elementos que cada grupo contém.

Para escolher o número de grupos não foi usado nenhum critério, visto este tipo de estudos ser feito com simulações em que se faz variar este número. Todavia, teremos de ter em causa o tamanho da sequência utilizada.

Dos cinquenta grupos obtidos retiramos seis, para análise, porque eram os mais representativos, visto conterem mais de 80% dos elementos.

Estes grupos, definidos pelos seus centros, constituem o "*self"* de cada compilador. A comparação de semelhança será feita comparando as coordenadas dos grupos para cada compilador.

Obtivemos os resultados registados na tabela 6.5.1. para as sequências numéricas de dois elementos:

Tabela 6.5.1. Centros dos Grupo de classificação mínima (0) para sequências de tamanho dois

Compilador1

Coordenadas		Nº
0,84	1,58	409
-14,13	-13,95	132
-13,44	3,00	73
2,80	-13,65	69
-13,56	-0,78	59
12,66	-0,26	35

Compilador 10

Coordenadas		Nº
1,07	1,19	397
-14,33	-14,29	203
-13,53	3,01	78
2,83	-13,68	76
-13,48	-0,83	52
13,19	-0,29	48

Compilador 25

Coordenadas		Nº
0,78	1,71	808
-14,37	-14,16	187
-13,51	3,00	75
2,87	-14,15	110
-13,51	-0,92	92
12,48	-0,25	63

Compilador 3

Coordenadas		Nº
0,80	1,39	430
-14,16	-13,96	149
-13,43	3,00	79
2,82	-13,60	78
-13,59	-0,75	73
-0,68	-13,42	38

Compilador 21

Coordenadas		Nº
0,74	1,78	424
-14,38	-14,31	164
-13,88	2,90	59
2,86	-13,74	70
-13,51	-0,73	49
13,02	-0,53	51

Compilador 29

Coordenadas		Nº
0,00	0,00	143
-14,39	-14,38	245
-13,00	-13,00	96
-1,00	-1,00	172
13,00	13,00	76
3,00	3,00	102

Compilador 7

Coordenadas		Nº
0,73	1,93	492
-14,19	-14,03	236
-13,48	3,00	79
2,85	-13,74	93
-13,52	-0,80	90
13,11	-0,31	54

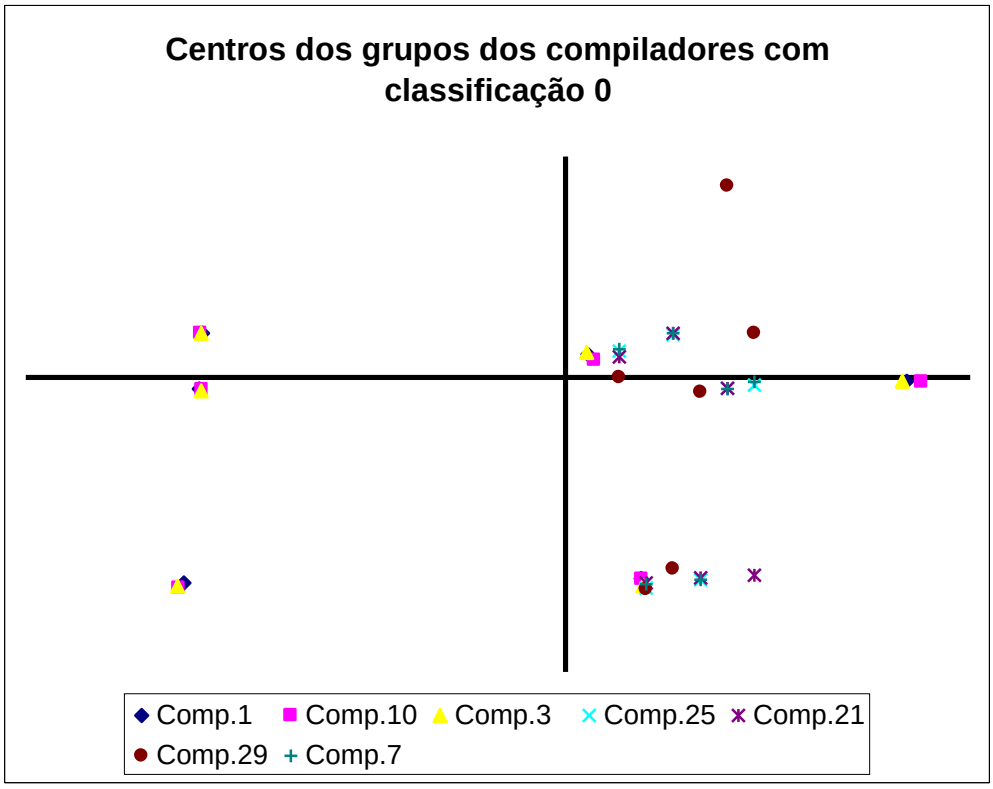


Figura 6.5.1. Coordenadas dos centros dos compiladores com classificação mínima (0)

A representação gráfica dos centros, feita na figura 6.5.1., mostra coincidência grande nos centros dos três grupos com maior número de elementos.

Tabela 6.5.2. Centros dos Grupo de classificação máxima para sequências de tamanho dois

Compilador 36

Coordenadas		Nº
0,90	-0,18	219
-14,07	-14,19	189
-13,67	3,01	99
0,55	-13,88	176
-13,50	-0,75	60
13,13	-0,34	47

Compilador 6

Coordenadas		Nº
0,10	-0,27	0,10
-14,25	-14,19	-14,25
-13,75	2,89	-13,75
2,87	-13,87	2,87
13,44	-1,07	13,44
-0,62	10,44	-0,62

Compilador 28

Coordenadas		Nº
-0,25	1,04	402
-14,36	-14,19	255
-13,52	0,85	157
0,51	-13,90	142
13,00	-1,29	65
7,80	-13,13	40

Compilador 2

Coordenadas		Nº
0,21	1,51	534
-14,38	-14,30	238
-13,54	0,95	140
2,80	-13,64	70
12,64	-0,28	36
7,70	-13,05	37

Compilador 32

Coordenadas		Nº
0,32	1,66	517
-14,33	-14,31	374
-13,63	3,00	78
2,80	-13,67	70
-13,59	-0,91	70
-0,83	-13,81	47

Compilador 34

Coordenadas		Nº
0,61	1,64	482
-14,40	-14,29	203
-13,54	0,92	142
2,80	-13,65	69
12,54	-0,26	39
7,77	-13,16	44

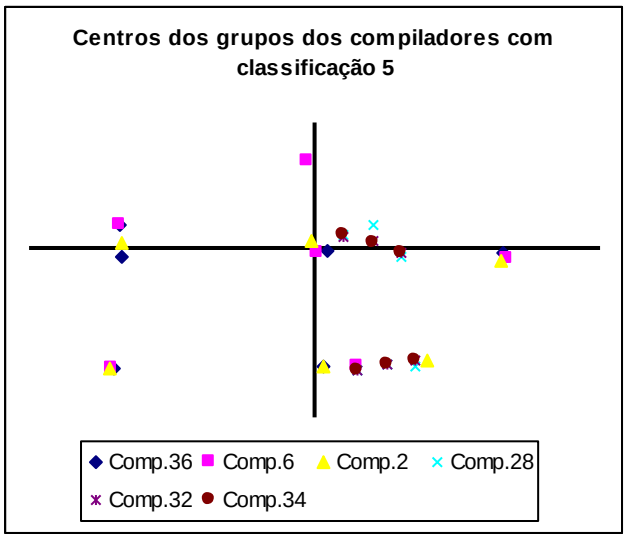


Figura 6.5.2. Coordenadas dos centros dos compiladores com classificação máxima.

A representação gráfica dos centros, feita na figura 6.5.2., mostra coincidência grande nos centros dos três grupos com maior número de elementos.

Apresentam-se de seguida os valores das coordenadas dos centros para um elemento de cada grupo de classificação. O compilador 1 pertence ao grupo de classificação 0 e o compilador 36 pertence ao grupo de classificação 5. Os restantes elementos de cada grupo foram omitidos, uma vez que o padrão é idêntico.
Na tabela 6.5.3. estão representados os centros para as sequências de tamanho três, quatro e cinco do compilador 1. Na tabela 6.5.4. estão representados os centros para as sequências de tamanho três, quatro e cinco do compilador 36

Tabela 6.5.3 Centros do compilador 1 para sequências de tamanho três, quatro e cinco

Compilador 1

Coordenadas			Nº
-1,01	2,41	0,04	562
2,69	-13,36	2,56	98
-14,23	-14,23	-14,19	77
-13,67	2,49	-13,65	69
-14,10	-13,52	0,52	50
13,18	0,00	2,08	38

Compilador 1

Coordenadas				Nº
0,30	2,45	0,25	1,59	660
-13,45	-14,39	-14,34	-14,34	67
3,00	-13,73	2,37	-13,39	62
-14,18	-13,56	1,31	-2,64	45
4,33	7,33	-13,13	-13,29	24
-1,00	-7,00	-15,00	-14,00	23

Compilador 1

Coordenadas					Nº
1,39	-0,65	-1,43	-1,98	-1,56	885
-10,68	-14,47	-13,76	-0,12	-6,44	34
-14,83	-13,92	-0,67	-7,50	-14,83	24
6,88	-13,18	-12,88	4,47	1,35	17
-14,69	-13,92	-14,54	-13,92	0,08	13
5,38	-12,23	4,54	5,85	-12,23	13

Tabela 6.5.4. Centros do compilador 36 para sequências de tamanho três, quatro e cinco

Compilador 36

Coordenadas			Nº
-3,72	-2,07	-2,84	653
0,80	9,95	13,05	20
-10,44	-14,22	-13,83	200
13,25	9,44	12,94	16
-9,95	1,95	11,24	21
13,50	-0,47	-0,03	30

Compilador 36

Coordenadas				Nº
-0,04	0,38	0,16	0,08	1048
-14,47	-14,41	-14,44	-14,36	133
4,38	-14,21	-13,97	-14,09	34
-14,17	-13,57	2,08	-7,08	92
-1,94	-14,20	-13,52	0,70	88
-4,54	2,91	-14,18	-13,71	87

Compilador 36

Coordenadas					Nº
-3,40	-3,35	-3,53	-3,48	-3,45	1043
-15,00	-10,00	15,00	-5,00	-14,00	6
-13,75	-14,50	-10,75	14,25	-3,75	4
15,00	-5,00	-14,00	0,00	-15,00	4
-10,75	15,00	-5,00	-14,00	-15,00	4
12,67	3,33	13,00	-13,00	6,67	3

Podemos reparar que quanto maior é o tamanho da subsequência considerada, maior é a concentração das subsequências num só grupo.

Realizamos os mesmos cálculos aumentando o número de grupos impostos para 80 e para 100 com o objectivo de verificar se se conseguia uma maior discriminação ou se realmente existia homogeneidade.

Para todos os compiladores, a concentração num único grupo, com o aumento do tamanho da subsequência considerada, manteve-se invariável.

De seguida, construímos o "*self*" do grupo de classificação 0 e testámos se poderia ser considerado "*self* " do grupo de classificação 5.

Para construir o "*self*" do grupo com classificação 0, consideramos os três primeiros compiladores e, por média pesada pelo número de elementos do centro, construímos os centros do "*self* ".

Obtemos os valores mostrados na tabela 6.5.5.

Tabela 6.5.5. Centros do "*self*" para o compilador 1 usando os compiladores 1, 3 e 7

Para seq.tam.2	
0,79	1,65
-14,16	-13,99
-7,25	-3,36
-3,39	-8,76
-13,54	0,64
8,86	-4,22

Para seq.tamanho 3		
-1,37	1,84	0,53
-2,86	-13,77	-7,01
-8,44	-13,73	-5,19
-13,54	2,56	-13,71
-6,85	-6,29	-5,71
10,15	5,44	1,12

Para seq.tamanho 4			
-1,61	-0,32	-0,89	-0,54
-11,22	-9,09	-14,22	-14,01
3,10	-13,75	-0,94	-9,98
-12,94	-13,86	-4,01	-4,35
2,10	-4,11	-13,65	-7,23
3,09	-1,64	-10,95	-10,09

Para seq.tamanho 5				
-1,43	-1,81	-2,21	-2,58	-2,22
-10,70	-14,08	-9,43	-3,28	-8,18
-4,56	-12,05	-4,26	-3,47	-8,79
6,23	-10,10	-5,07	6,17	-4,53
-11,78	-6,30	-14,04	-13,57	0,39
4,75	-3,75	7,00	7,00	-12,50

De seguida calculamos as distâncias dos "*selfs*" dos restantes compiladores do grupo de classificação 0 a este "*self*" com este procedimento, se a distância referida for mínima reforçamos a construção do "*self*" do grupo de classificação 0.

Esta distância é uma distância euclidiana normalizada em relação ao código e em relação ao número de coordenadas do "*self* " respectivo.

Os resultados da distância correspondente ao grupo com classificação mínima estão descritos na tabela 6.5.6. e 6.5.7..

Da análise dos dados verificamos que não existe comportamento diferente nas distâncias ao "*self"* do grupo de classificação 0 dos seus próprios elementos e dos elementos do grupo de classificação máxima.

Pode ainda verificar-se que apenas o primeiro grupo e o segundo para cada um dos tamanhos mantém as distâncias em valores maioritariamente iguais ou menores que 0,15 sendo que nos restantes a variação da distância é muito maior. Consideramos, pois, que só estes dois grupos poderão ser característicos desta amostra.

A utilização desta técnica, com esta amostra, não acrescenta nenhuma nova possibilidade de discriminação entre os compiladores. Indica-nos que, tal como as outras medidas analisadas em separado já o faziam, a amostra é homogénea em relação aos dois tipos de estrutura.

Tabela 6.5.6. Distâncias ao "*self* "do grupo de classificação 0 dos elementos do grupo não usados na construção do "*self*".

Distância do compilador 10 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,01	0,05	0,02	0,03
0,01	0,14	0,05	0,09
0,14	0,16	0,14	0,08
0,13	0,00	0,17	0,20
0,02	0,25	0,11	0,16
0,09	0,08	0,15	0,25

Distância do compilador 21 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,00	0,03	0,02	0,01
0,01	0,15	0,21	0,16
0,23	0,15	0,19	0,10
0,25	0,17	0,19	0,17
0,43	0,15	0,10	0,15
0,37	0,19	0,07	0,15

Distância do compilador 25 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,00	0,15	0,05	0,08
0,24	0,15	0,17	0,15
0,21	0,36	0,22	0,18
0,04	0,17	0,16	0,15
0,09	0,31	0,07	0,16
0,16	0,12	0,12	0,11

Distância do compilador 29 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,03	0,03	0,04	0,01
0,01	0,14	0,04	0,15
0,11	0,35	0,13	0,12
0,44	0,25	0,18	0,17
0,22	0,02	0,15	0,17
0,15	0,19	0,15	0,18

Distância do compilador 36 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,03	0,06	0,04	0,02
0,00	0,11	0,25	0,17
0,21	0,28	0,16	0,14
0,25	0,27	0,25	0,14
0,02	0,28	0,22	0,18
0,09	0,14	0,22	0,20

Tabela 6.5.7. Distâncias ao "*self*" do grupo de classificação 0 dos elementos do grupo com classificação máxima

Distância de Compilador 2 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,01	0,03	0,04	0,02
0,01	0,14	0,15	0,15
0,12	0,16	0,18	0,16
0,13	0,16	0,16	0,23
0,41	0,11	0,15	0,23
0,09	0,31	0,14	0,19

Distância de Compilador 6 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,03	0,03	0,02	0,01
0,00	0,14	0,05	0,18
0,24	0,33	0,20	0,16
0,31	0,29	0,16	0,12
0,44	0,24	0,16	0,26
0,38	0,28	0,03	0,12

Distância de Compilador 28 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,02	0,03	0,02	0,02
0,00	0,15	0,05	0,21
0,12	0,20	0,15	0,12
0,10	0,25	0,16	0,21
0,43	0,13	0,11	0,25
0,14	0,35	0,21	0,17

Distância de Compilador 32 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,01	0,02	0,05	0,03
0,01	0,15	0,15	0,10
0,15	0,12	0,16	0,12
0,13	0,25	0,10	0,22
0,03	0,13	0,11	0,15
0,22	0,28	0,07	0,20

Distância de Compilador 34 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,00	0,01	0,01	0,01
0,01	0,15	0,05	0,11
0,12	0,16	0,19	0,08
0,13	0,01	0,17	0,20
0,41	0,23	0,10	0,28
0,09	0,34	0,07	0,12

Distância de Compilador 35 ao "*self*"			
Tamanho da sub sequência			
2	3	4	5
0,01	0,04	0,02	0,01
0,00	0,15	0,05	0,15
0,11	0,16	0,14	0,21
0,25	0,00	0,12	0,21
0,11	0,13	0,10	0,15
0,22	0,11	0,08	0,17

Os resultados suscitam o seguinte comentário:

1. Estes valores reforçam a confirmação da secção anterior que a complexidade não está ligada ao número de funcionalidades existentes no programa.
2. Tal como o indicado no segundo comentário da secção 6.3., a independência da complexidade em relação ao tamanho deve ser testada com sistemas de classificação que venham a ser desenvolvidos e com outras amostras de programas.

6.6. Contribuição deste capítulo para a defesa da tese que é objecto desta dissertação

A contribuição deste capítulo para a defesa da tese que é objecto desta dissertação é a introdução de dois novos métodos de análise, taxonómica e de identidade.

Obtiveram-se valores semelhantes de correlação de gama longa em dois grupos de classificação que a análise de identidade indicou serem homogéneos.

O uso destas técnicas não permitiu, nos programas de compiladores que foram analisados, discriminar os programas com base no número de funcionalidades implementadas.

A utilização da análise taxonómica e de identidade reforça a hipótese de a correlação de gama longa ser a medida mais adequada para a caracterização da estrutura.

7. Dinâmica do processo segundo o modelo da complexidade

...Afinal de contas as equações não lineares não se conseguem resolver; portanto, para quê desperdiçar tempo nesses problemas? ... inevitavelmente, o mundo real iria intrometer-se, e Marcos veria o que alguns anos depois compreendeu serem os sinais do caos... um sistema determinista pode produzir muito mais do que um simples comportamento periódico. Ilhas de ordem podem surgir no meio da desordem...

James Gleick

7.1. Introdução

É objectivo deste capítulo apresentar uma aplicação do cálculo da correlação de gama longa para controlo do processo desenvolvimento de software. Para tal, apresentamos brevemente a equação logística, enquanto exemplo de representação de um sistema dinâmico não linear.
De seguida, aplicamos a referida equação logística enquanto modelo da dinâmica de um processo de desenvolvimento de software real e exemplificamos como é possível determinar o estado do processo de desenvolvimento a partir do valor da correlação de gama longa (obtido para versões sucessivas de programas). A partir destes resultados discutimos a utilidade do modelo referido para controlar o processo de desenvolvimento de software.

7.2. A dinâmica de um sistema complexo

Tal como apresentado no capítulo quatro, de uma maneira geral, o estudo dos sistemas complexos segue duas vias complementares, cada uma delas com objectivos distintos. A primeira está direccionada para a melhoria da capacidade de prever o comportamento dos sistemas complexos, enquanto a segunda está associada ao estudo da criação de estruturas. Foi esta segunda via, ou seja, a caracterização de estruturas, aquela que exploramos ao longo desta dissertação.

As abordagens dinâmica e estrutural da complexidade encontram-se relacionadas, na medida em que, as estruturas criadas pela interacção entre as partes dos sistemas complexos são tanto mais ricas quanto mais o sistema estiver próximo de uma transição para um estado de imprevisibilidade (ou de caoticidade) absoluta. [Vilela Mendes,R.99]. Assim sendo, quanto o estudo da complexidade recai sobre a componente dinâmica dos sistemas, é comum que tenhamos que nos debruçar sobre a noção de caos.

Felizmente este é um campo de estudo onde existem instrumentos matemáticos bem definidos, os quais permitem caracterizar a situação de caoticidade de forma clara e solidamente estabelecida. Tal como já apresentado no capítulo quatro, as grandezas matemáticas correspondentes são os expoentes de Lyapunov e a Entropia de Kolmogorov-Sinai. A propriedade fundamental exibida pelos sistemas caóticos é, como já se viu, a chamada dependência sensível das condições iniciais.

A equação logística

Para modelar matematicamente a dinâmica do processo de desenvolvimento de software, escolhemos a equação logística. A opção por esta equação deve-se a suas características de não-linearidade e à sua capacidade de representação de comportamentos bastante diversificados. A diversidade de comportamentos possíveis de representar com a equação logística depende apenas de modificações no valor de um único parâmetro (*k*);

A equação logística tem sido adoptada como modelo das variações de populações de espécies vivendo num ambiente de recursos limitados.

Na sua forma discreta a equação logística é dada por:

$$x_{t+1} = kx_t(1 - x_t)$$

onde:

x_t representa o valor da população num determinado período (normalizado pelo valor máximo que a população pode atingir). O parêntesis $(1 - x_t)$ faz com que, para valores (da população) próximos de zero, quando existem recursos disponíveis, a população cresça proporcionalmente à população anterior e que, pelo contrário, diminua quando x_t atinge o valor (da população) máximo para a região. A representação gráfica da equação logística é a de uma parábola invertida.

Para os valores de x_t menores que 0 e maiores que 1, x_{t+1} é negativo. Uma população

inexistente não se reproduz e uma população além do seu valor máximo morre por falta de recursos. Interessa, pois, analisar o comportamento da equação dentro do intervalo [0,1].

A constante k, o parâmetro da equação, indica o achatamento da parábola, visto o máximo acontecer no ponto (0.5,k/4), onde a derivada em ordem a x_t é nula. O parâmetro k tem um papel na definição do comportamento da população. O intervalo de variação de k é entre 0 e 4, visto que, acima de 4, o valor máximo da população seria maior do que 1, o que traduziria um valor impossível.

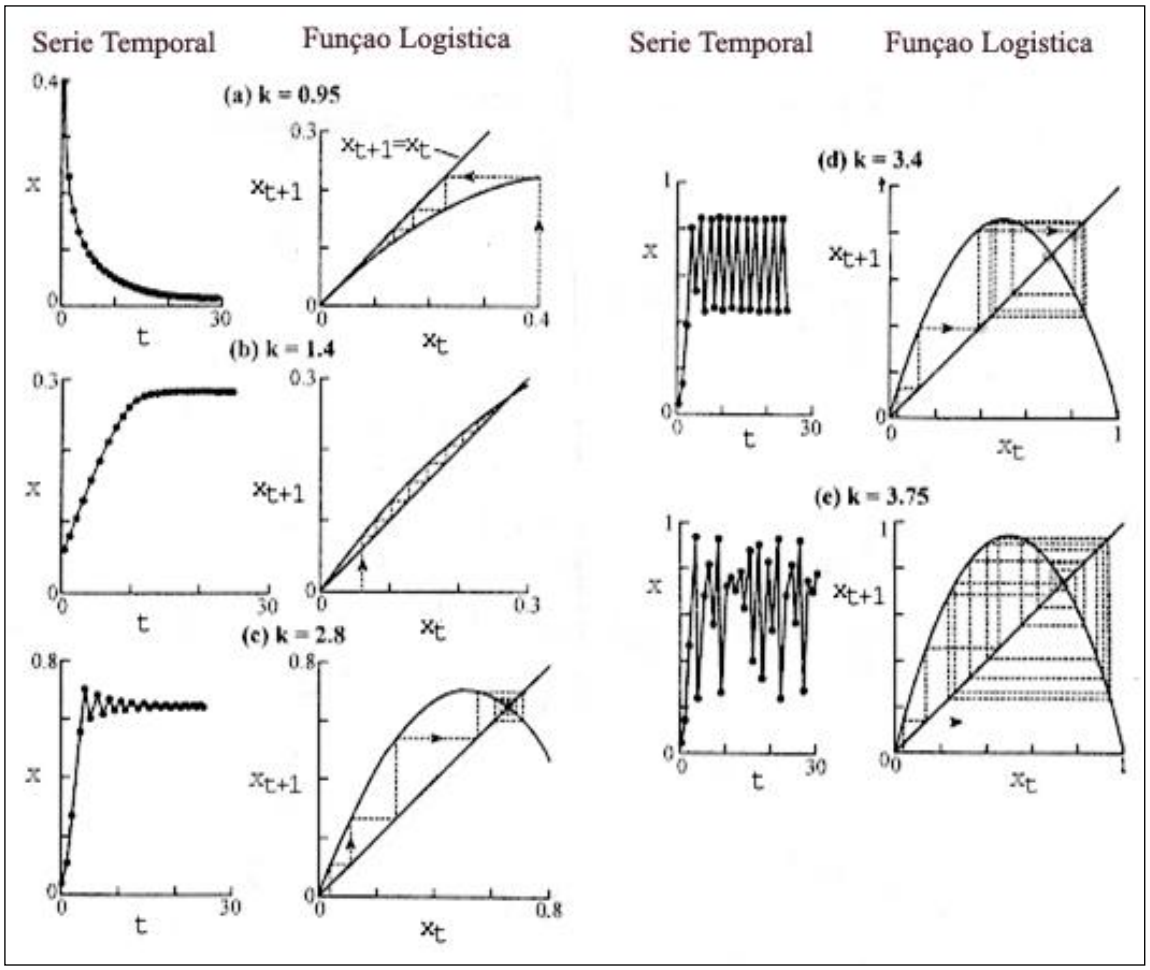


Figura. 7.2.3.1 Séries temporais e equação logística para 5 valores diferentes do parâmetro k.

Com k entre 0 e 4, se tivermos em conta *i)* as séries temporais formadas pelos valores consecutivos da variável *x* e *ii)* a representação de x_{t+1} em equação de x_t (o chamado "espaço

de fase" da equação) identificamos cinco situações diferentes. Estas situações encontram-se apresentadas na Figura 7.3.2.1., sendo aí designadas pelas primeiras cinco letras do alfabeto.

1. Na situação (a), k é igual a 0,95 e o valor inicial da população corresponde a 40% da população máxima (0,40). A série temporal rapidamente se aproxima de zero. Repetindo os cálculos com outros valores iniciais de x_t concluímos que se verifica a mesma aproximação ao valor nulo. O valor inicial (de partida) é irrelevante. Com k≤1 a série temporal é sempre atraída para o valor nulo. Este valor (zero) é designado, nesse caso, por atractor.
2. Na situação (b) k é igual a 1,4 e o valor inicial corresponde a 0,06 (6% da população máxima). Também neste caso, repetindo os cálculos para outros valores iniciais, a série temporal converge sempre para um único valor aproximado de 0,286.
3. Na situação (c), k é igual a 2,8 e para qualquer valor inicial da população a série temporal converge sempre para um único valor aproximado de 0,643. Quando k se encontra entre 1 e 3, quaisquer que sejam os valores iniciais da população x_t, ela é atraída para um estado estável cujo valor é 1-1/k, o ponto de intersecção da parábola com a linha $x_{t+1} = x_t$ [Ott,E.93].
4. Na situação (d), k é igual a 3,4 e para qualquer valor inicial da população a série temporal converge sempre para dois valores próximos de 0,452 e 0,842.
5. Quando k >3,57 os valores da equação tornam-se erráticos no gráfico, como na situação (e), e impossíveis de prever. Com k≥3 aparece inicialmente um atractor periódico constituído por dois pontos que, à medida que k aumenta, se torna sucessivamente em atractor periódico de taxa dupla (4, 8, 16, ... pontos).

Desta descrição concluímos que, naquele modelo, o parâmetro k funciona como parâmetro de controlo da equação logística. Se esta equação for um modelo aceitável para uma qualquer dinâmica no mundo real, o conhecimento do valor tomado por k em cada iteração do sistema pode indicar se o mesmo se encontra num estado estável ou se aproxima de um estado caótico. Uma actuação no sentido de o modificar (quando possível ou quando desejável) permite provocar a manutenção de estados estáveis ou controlar a duração de estados caóticos controlados quando pretendermos introduzir mudanças.

Na secção 7.3, apresentamos as analogias encontradas entre o modelo que a equação logística

formaliza e o processo de desenvolvimento. Na secção 7.4 damos conta da utilização dos valores da correlação de gama longa na modelação de alguns comportamentos expressos pela equação logística.

7.3. A modelação do processo real de desenvolvimento do software

Como referido na secção 4.2., a generalidade dos sistemas complexos pode ser identificada uma série de propriedades comuns. A característica mais frequentemente encontrada em todos estes sistemas é a de serem, em geral, compostos por um grande número de elementos e o seu comportamento apresentar propriedades que não podem ser facilmente deduzidas a partir das propriedades do comportamento dos elementos quando isolados. A emergência de propriedades colectivas, qualitativamente diferentes das individuais e a não-linearidade das interacções entre os elementos são as características fundamentais do comportamento dos sistemas complexos.

Os programas são formados por sub rotinas, módulos e funções hierarquizadas em diferentes níveis que detêm em separado características diferentes do programa no seu conjunto [Kokol,P.00]. Também a interacção entre as suas partes não é linear, uma vez que pequenas alterações numa delas pode provocar efeitos que podem impedir até o funcionamento de outras com elas relacionadas.

Consideramos que o processo ideal se desenvolve de acordo com três fases.

O processo ideal inicia-se com um período de formação de ideias, seguido de um período de convergência de ideias e termina com um período de implementação de uma única ideia. Esta descrição é válida também para a construção das sucessivas versões e está de acordo com o desenvolvimento e controlo do processo real, descrito no capítulo 2.

Consideramos que o produto se comporta, durante as três fases e em relação à criatividade, da forma seguinte: no início do processo, existe uma grande criatividade e grande número de ideias, estes vão diminuindo e estabilizam na fase de implementação de uma única ideia.

Em relação ao processo, como descrevemos na Figura 7.3.1., consideramos que a primeira parte é uma fase de muitas soluções possíveis, o que está de acordo com a descrição que fizemos de sistema caótico.

Nessa fase o processo é muito sensível às condições iniciais: por exemplo, sabemos por experiência, que se um único elemento da equipa de desenvolvimento dispuser de

conhecimento de uma ferramenta de desenvolvimento, constrói soluções diferentes da equipa que não conheça tão bem essa ferramenta.

Seguidamente, o processo necessita de ser controlado, porque só se tornará produtivo, se começar a desenvolver unicamente ideias promissoras. Nessa fase, a analogia estabelece-se com o regime de bifurcação (periódico) podendo estar a equipa, por exemplo a desenvolver vários protótipos e a testar as vantagens de cada um.

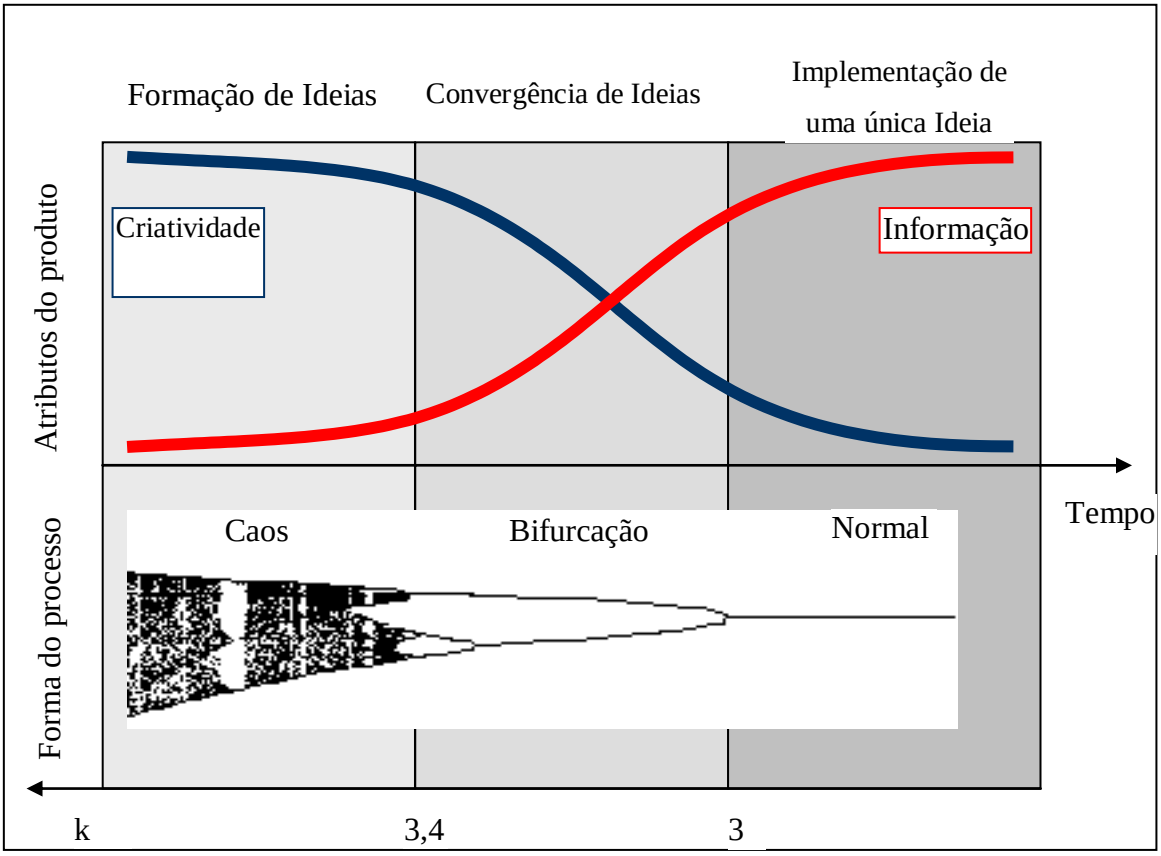


Figura 7.3.1. Comportamentos nas três fases do desenvolvimento do processo ideal.

Finalmente, o processo acaba com desenvolvimento de uma única ideia – a solução escolhida - que funciona como atractor do regime estável [Cardoso,A.01;Kokol,P.00a; Kokol,P.00b].

O processo real não segue a sucessão destas três fases, nem sempre evolui para a solução. Muitas vezes retoma fases anteriores, regredindo afastando-se da solução, e, como tal, é relevante encontrar uma medida cuja variação indique esse facto para poder antecipar as

acções de controlo necessárias a levar o processo para o estado estável.

As leis de Lehman foram descritas informalmente, com base na observação do seu autor. Nesta dissertação, propõe-se contribuir com um estudo experimental de um modelo matemático que possibilite a sua descrição.

7.4. A equação logística como modelo da dinâmica de um processo de desenvolvimento de software real

Descrevemos nesta secção como é possível ajudar a controlar o processo de desenvolvimento utilizando a equação logística. O valor da correlação de gama longa (α) varia em 0,5 (aleatoridade) e 1 (máximo) e os valores de x_t na equação logística variam entre 0 (mínimo) e 1 (máximo). Assim, torna-se necessário normalizar α para

$$\alpha_{norm} = 2|\alpha - 0.5|$$

A equação logística, empregue neste contexto, é então reescrita como

$$k = \frac{\alpha_{t+1}}{\alpha_t(1-\alpha_t)}$$

Retornando os estados do processo (representados na Figura 7.3.1.), estes serão definidos pela tendência de α [Kokol,P.00], quando α é calculado para sucessivas versões de um programa. Identificamos então quatro diferentes situações:

1. **O valor de alfa é estável**, ou seja as sucessivas versões mantém o valor de α sem grandes variações.
O processo encontra-se no estado de implementação de uma única ideia.

2. **O valor de alfa diminui nas sucessivas versões**.
O processo encontra-se no estado de implementação de uma única ideia a regredir para o estado de convergência de ideias, ou no estado de convergência de ideias a regredir para o

estado de formação de ideias.
Se o parâmetro da equação logística - k - é menor que 3 o processo ainda está na fase de implementação de uma só ideia.
Se o parâmetro da equação logística - k - é maior que 3 e menor que 3,6 está na fase de implementação de convergência de ideias .
Em qualquer dos casos o processo encontra-se no estado de implementação de modificações no sistema, ainda não estáveis.

3. O valor de alfa aumenta nas sucessivas versões.
O processo encontra-se no estado de convergência de ideias a evoluir para o estado de implementação de uma única ideia, ou no estado de formação de ideias a evoluir para o estado de convergência de ideias.
Se o parâmetro da equação logística - k - é maior que 3 e menor que 3,6 o processo está a evoluir para a fase de implementação de uma só ideia.
Se o parâmetro da equação logística - k - é maior que 3,6 o processo está a evoluir para a fase de convergência de ideias.
Em qualquer dos casos o processo encontra-se no estado de implementação de modificações no sistema que se estão a tornar estáveis.

4. O valor de alfa oscila nas sucessivas versões
Se o parâmetro da equação logística - k – é menor que 3,6 o processo está na fase de convergência de ideias.
Se o parâmetro da equação logística - k – é maior que 3,6 o processo está na fase de formação de ideias.
O processo encontra-se em estados indefinidos quanto à aproximação da solução

Assim, pelo conhecimento dos valores de α é possível calcular para duas versões sucessivas o valor de k, e saber se a versão mais recente se encontra ou não mais perto da solução. Tal cálculo permite controlar o estado do processo.

Exemplo 7.4.1.: Num sistema real obtivemos, para as três primeiras versões de 12 programas, os valores de α. Trata-se de versões de uma das componentes desenvolvidas ao longo de

aproximadamente três meses, em C para um dos programas constituintes do Office da Microsoft™, num laboratório Universitário. O sigilo contratual impede a indicação de dados mais precisos nesta dissertação. Registámos também a opinião do responsável sobre o seu estado de desenvolvimento. A tabela 7.4.1. mostra a informação obtida.

Tabela 7.4.1. Informação sobre 12 programas de um sistema real.

Nº Prog.	α_1	α_2	α_3	**Causas de alterações**
1	0,77	0,77	0,78	Pequenas modificações localizadas
2	0,77	0,76	0,76	Pequenas modificações localizadas
3	0,82	0,82	0,74	Removidas duas funções
4	0,74	0,74	0,75	Pequenas modificações localizadas
5	0,74	0,74	0,74	Pequenas modificações localizadas
6	0,83	0,77	0,79	Modificação da estrutura por adição de código
7	0,50	0,74	0,74	Detectado e corrigido um importante erro de código
8	0,78	0,78	0,78	Pequenas modificações localizadas
9	0,77	0,77	0,77	Pequenas modificações localizadas
10	0,78	0,78	0,78	Pequenas modificações localizadas
11	0,75	0,74	0,71	Várias instâncias de uma estrutura foram inseridas
12	0,77	0,73	0,70	Várias instâncias de uma estrutura foram inseridas

Na tabela seguinte estão registados: a tendência observada na variação dos três valores de α, o cálculo de k entre a segunda e a terceira versão ($\mathbf{k_{23}}$) e o estado do sistema. Juntámos a coluna do quadro anterior respeitante ao estado do sistema na opinião do responsável pelo desenvolvimento.

Tabela 7.4.2. Cálculo da fase para os 12 programas de um sistema real.

Nº Prog.	**Tendência do α**	**k_{23}**	**Fase de acordo com a tendência de α e k_{23}**	**Causas de alterações**
1	estável	1,79	Implementação de uma única ideia	Pequenas modificações localizadas
2	estável	1,80	Implementação de uma única ideia	Pequenas modificações localizadas
3	a diminuir	2,29	A regredir da fase de Implementação de uma única ideia para a de convergência de ideias	Removidas duas funções
4	estável	2,01	Implementação de uma única ideia	Pequenas modificações localizadas
5	estável	2,06	Implementação de uma única ideia	Pequenas modificações localizadas
6	a oscilar	1,68	Convergência de ideias	Modificação da estrutura por adição de código
7	estável	2,09	Implementação de uma única ideia	Detectado e corrigido um importante erro de código
8	estável	1,76	Implementação de uma única ideia	Pequenas modificações localizadas
9	estável	1,83	Implementação de uma única ideia	Pequenas modificações localizadas
10	estável	1,78	Implementação de uma única ideia	Pequenas modificações localizadas
11	a diminuir	2,36	A regredir da fase de Implementação de uma única ideia para a de convergência de ideias	Várias instâncias de uma estrutura foram inseridas
12	a diminuir	2,44	A regredir da fase de Implementação de uma única ideia para a de convergência de ideias	Várias instâncias de uma estrutura foram inseridas

As duas últimas colunas mostram que a estimativa da fase a partir da tendência de α e do cálculo de k coincide com a informação do técnico de desenvolvimento sobre o estado do sistema.

7.5. Análise de uma amostra real

Descrevemos de seguida o estudo que efectuámos das sucessivas versões de programas de uma amostra. A amostra estudada é constituída por programas da JBoss, uma organização

mundial de software livre com o URl http://www.jboss.org. Além do código, para as várias versões, a JBoss disponibiliza acesso à descrição das modificações da equipa, das metodologias e do código.

Os programas analisados fazem parte do JBoss Application Server, escrito em Java [Lindholm,99]. A correlação de gama longa foi medida nos programas objecto de acordo com a metodologia descrita na secção 5.5..

A utilização da amostra em Java alarga o âmbito da aplicação da métrica de correlação de gama longa a outra linguagem. Devido ao número mais elevado de palavras-chave e à pequena dimensão de alguns programas analisados, optou-se efectuar as medidas no ficheiro objecto de *Bytecodes*. Esta opção não põe em causa os resultados uma vez, conforme observado no capítulo 5, existe uma forte correlação entre os programas fonte e objecto.
Os nomes dos programas e a indicação do número de versões, medidas para cada um, estão indicados na tabela 7.5.1.. Em anexo 7.5.1. estão registadas os gráficos das medições obtidas para cada programa. Entre parêntesis, indicamos à frente de cada programa o tipo de comportamento observado. Utilizamos U para implementação de uma única ideia, E para comportamento evolutivo, partindo de várias ideias e caminhando para uma única ideia e R para comportamento evolutivo, com fases de regressão do número de ideias .

A análise realizada permitiu distinguir os 3 grupos de comportamentos distintos:

1. Comportamento estável, de implementação de uma única ideia (U)

Na amostra existe um exemplo, o programa JDBCKeyGeneratorCreateCommand, que apresenta valores de α praticamente constantes. Foi entregue depois de construídas apenas 3 versões, e embora com um número elevado de linhas de código, cerca de 5600, não aumentou esse número de uma versão para outra. O valor de alfa é alto: 0,81. O valor de k é 1,64 <3 indicando estabilidade. A equipa que o desenvolveu também não foi modificada. Utilizando o processo de análise descrito em na secção 7.4 o processo encontra-se no estado de implementação de uma única ideia.

Tabela 7.5.1.Lista dos programas que constituem a amostra analisada

Versões	Programas
48	StatefulSessionInstanceInterceptor(R)
43	ApplicationMetaData(R)
35	JDBCStartCommand(R)
34	JDBCLoadEntityCommand(R)
29	LRUEnterpriseContextCachePolicy(R), JDBCEJBQLCompiler(E)
24	JDBCCMPFieldMetaData(R)
22	JDBCLoadRelationCommand(R)
21	CMPFilePersistenceManager(R)
19	JDBCAbstractCMPFieldBridge(R)
16	JDBCStopCommand(R)
15	ServerSessionPoolLoader(R)
14	DLQHandler(R), Interceptor(R)
12	ResourceRefMetaData(R)
11	JDBCInsertRelationsCommand(E), JDBCDeleteRelationsCommand(E)
10	BeanCacheMonitor(E), TxEntityMap(R), J2eeApplicationMetaData(E)
9	EnvEntryMetaData(E), JDBCFindEntityCommand(E), J2eeModuleMetaData(E), MappingMetaData(E)
8	ThreadPool(E), QueryMetaData(E), ConnectorFactoryService(E), EntityBridge(E)
7	JDBCAutomaticQueryMetaData(E)
6	ClientUserTransactionObjectFactory(E), RelationMetaData(E), JMXInvokerInterceptor(E)
5	ClientUserTransaction(E), RemoteDeployer(E), PollingClientNotificationListener(E), JDBCValueClassMetaData(E), JDBCQueryCommand(E), InstancePoolContainer(E)
4	JDBCFindByPrimaryKeyQuery(E)
3	AbstractClient(E), JPMStopCommand(E), InstancePoolFeeder(E), JDBCKeyGeneratorCreateCommand (U)

Os valores observados indicam um estado coincidente com o resultado dessa análise.

Nas Figuras 7.5.1. a 7.5.3. estão representados, respectivamente, os gráficos de variação do α, e do número de linhas para as duas versões deste programa.

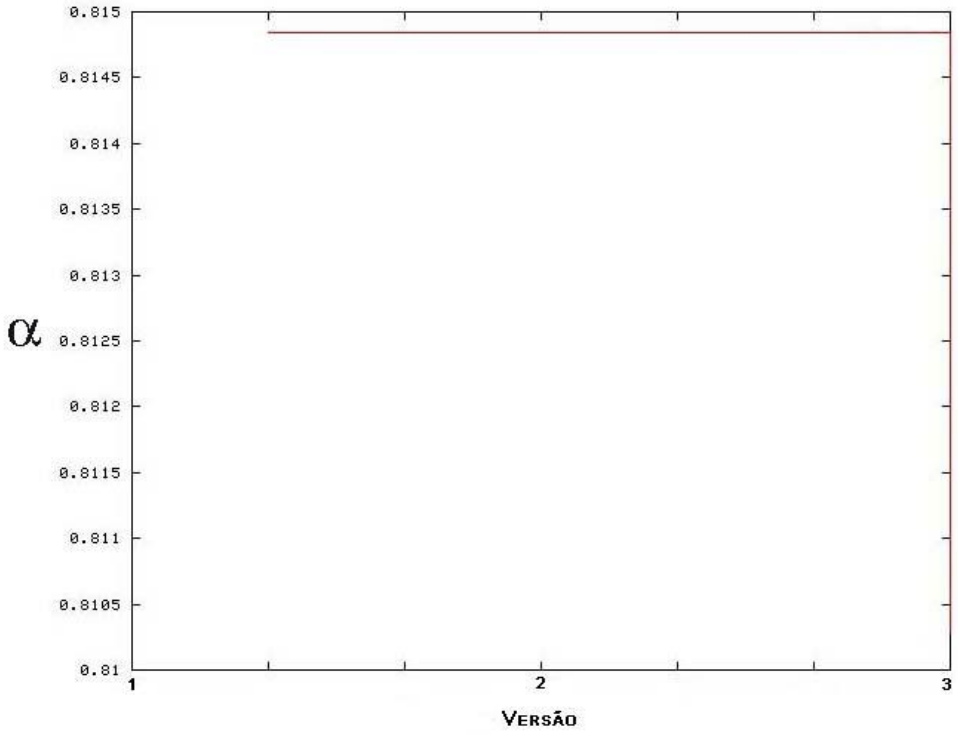


Figura. 7.5.1. α para o programa o primeiro tipo.

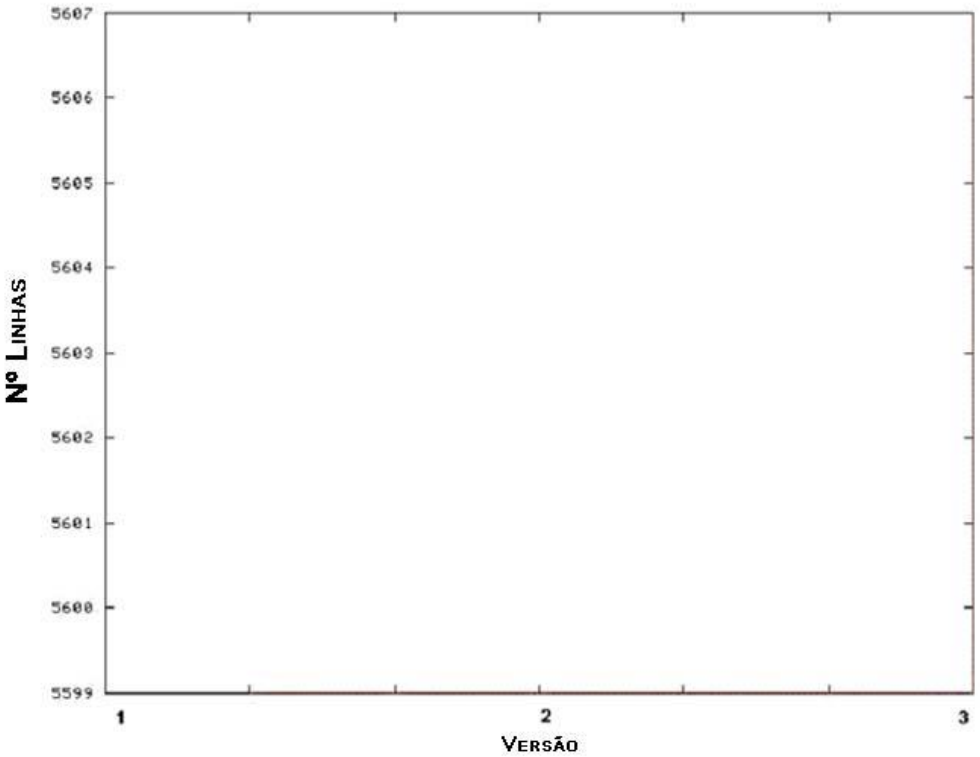


Figura. 7.5.2.número de linhas para o programa o primeiro tipo.

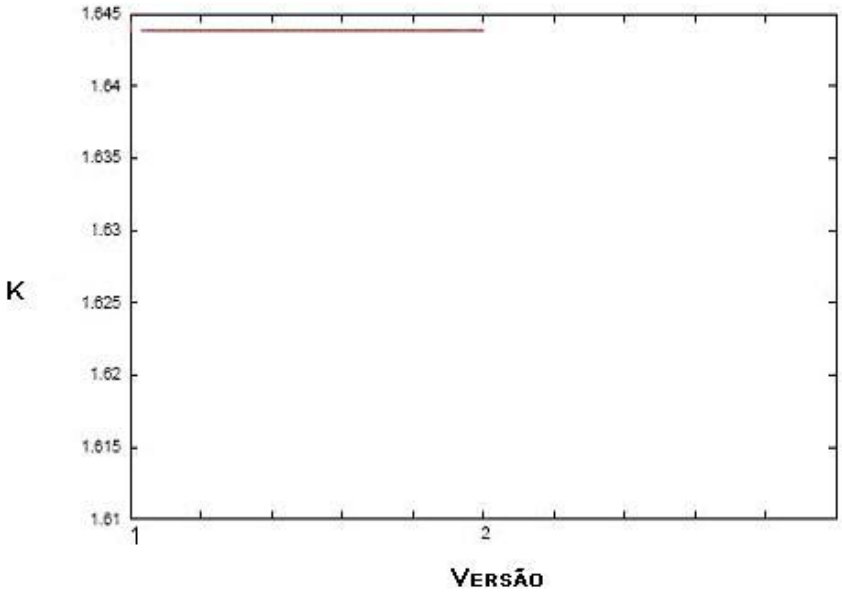


Figura. 7.5.3. k para o programa o primeiro tipo.

2. Comportamento evolutivo, partindo de várias ideias e caminhando para uma única ideia (E)

Foram observados 27 programas, que apresentam valores de α inicialmente baixos. Um deles, o JDBCEJBQLCompiler foi entregue depois de construídas 28 versões, mas os restantes apresentaram um número de versões baixo, menor ou igual a 11. O número de linhas de código aumentou da versão inicial para a versão final. O valor de alfa inicialmente é baixo, perto de 0,6 e apresenta algumas oscilações, aumentando de seguida para valores de cerca de 0,8. Os valores de k são, no início próximos de 3,6 descendo para valores entre 2 e 1,8. A equipa que os desenvolveu é estável e não estão reportadas mudanças de metodologias de desenvolvimento.

Utilizando o processo de análise descrito em na secção 7.4 o processo evolui do estado de implementação de várias ideias para o estado de convergência de ideias e de seguida para o estado de implementação de uma única ideia.

A estabilização do valor de α coincide muito frequentemente com a estabilização do número de linhas do programa.

Os valores observados indicam um estado coincidente com o resultado dessa análise.

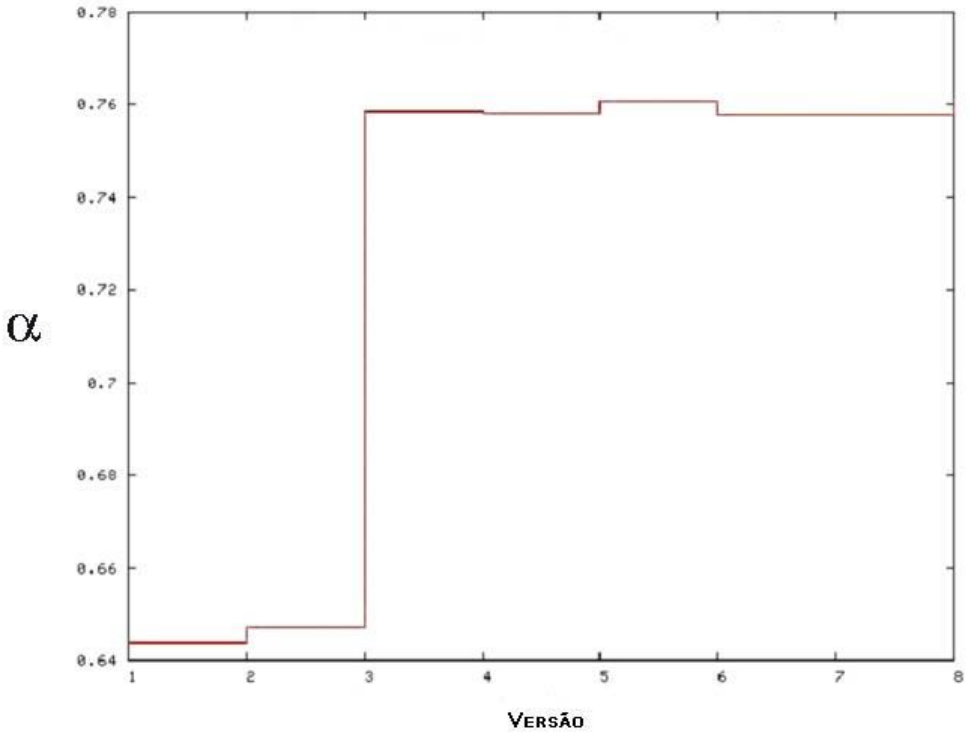


Figura. 7.5.4. α para o programa do segundo tipo.

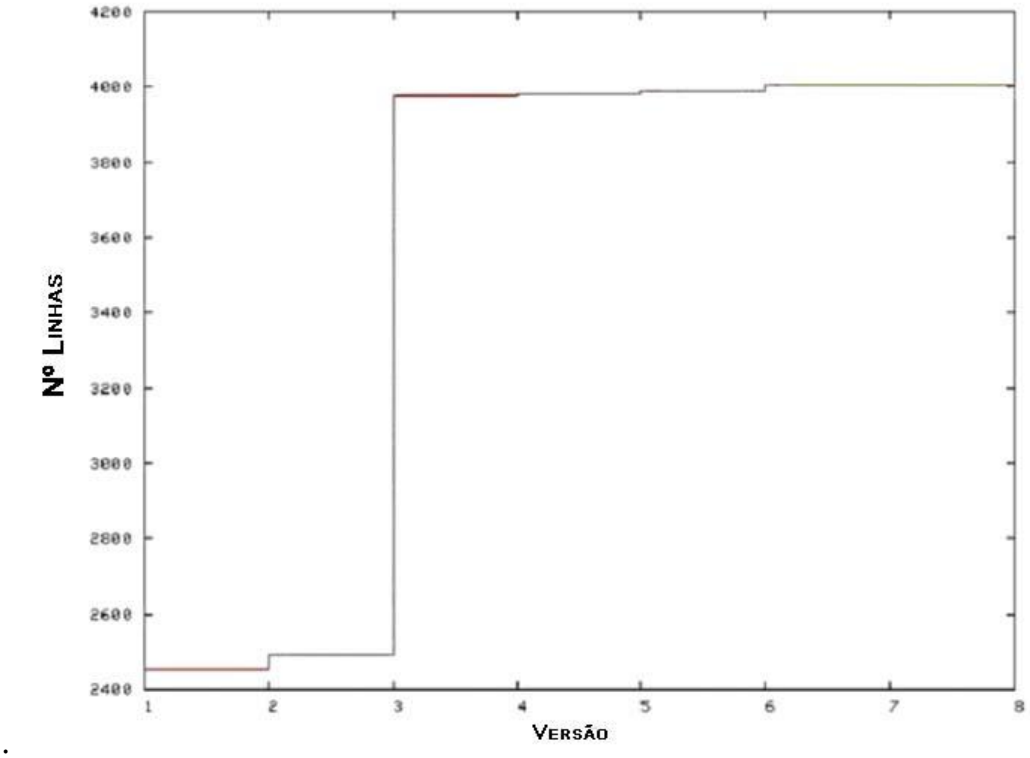


Figura. 7.5.5. número de linhas para o programa do segundo tipo.

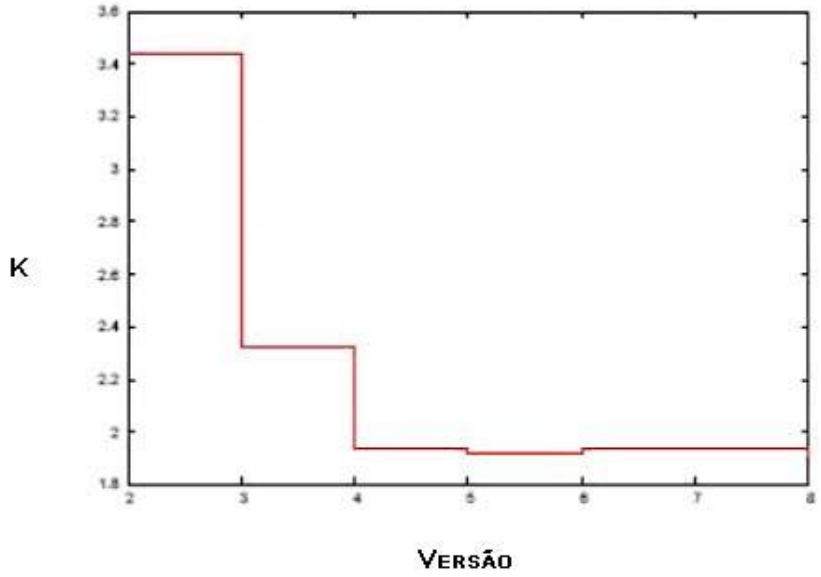


Figura. 7.5.6. k para o programa do segundo tipo.

Nas figuras 7.5.4 a 7.5.6. estão representados os gráficos de variação do α, k e do número de linhas para as duas versões de um programa deste grupo, o ThreadPool com 8 versões.

3. Comportamento evolutivo, com fases de regressão do número de ideias (R):

Foram observados 15 programas, que apresentam valores de α inicialmente baixos. Excluindo 2 programas que foram desenvolvidos respectivamente em 6 e 11 versões, os restantes apresentaram um número de versões alto, maior que 28. O número de linhas de código aumentou da versão inicial para a versão final. O valor de alfa inicialmente é baixo, perto de 0,6 e apresenta oscilações durante todo o processo terminando com valores próximos de 0,8. O parâmetro k oscila de acordo com os estados apresentados na secção 7.4.. A equipa que os desenvolveu por vezes é mudada e estão reportadas alterações de metodologias de desenvolvimento. As versões construídas na altura em que existiram quer alterações da equipa quer alterações de metodologia correspondem às oscilações dos valores de alfa.

Utilizando o processo de análise descrito em na secção 7.4 o processo evolui do estado de implementação de várias ideias para o estado de convergência de ideias e de seguida para o estado de implementação de uma única ideia, mas apresentou nessa evolução episódios de regressão de convergência de ideias para implementação de ideias ou de implementação de uma única ideia para convergência de ideias.

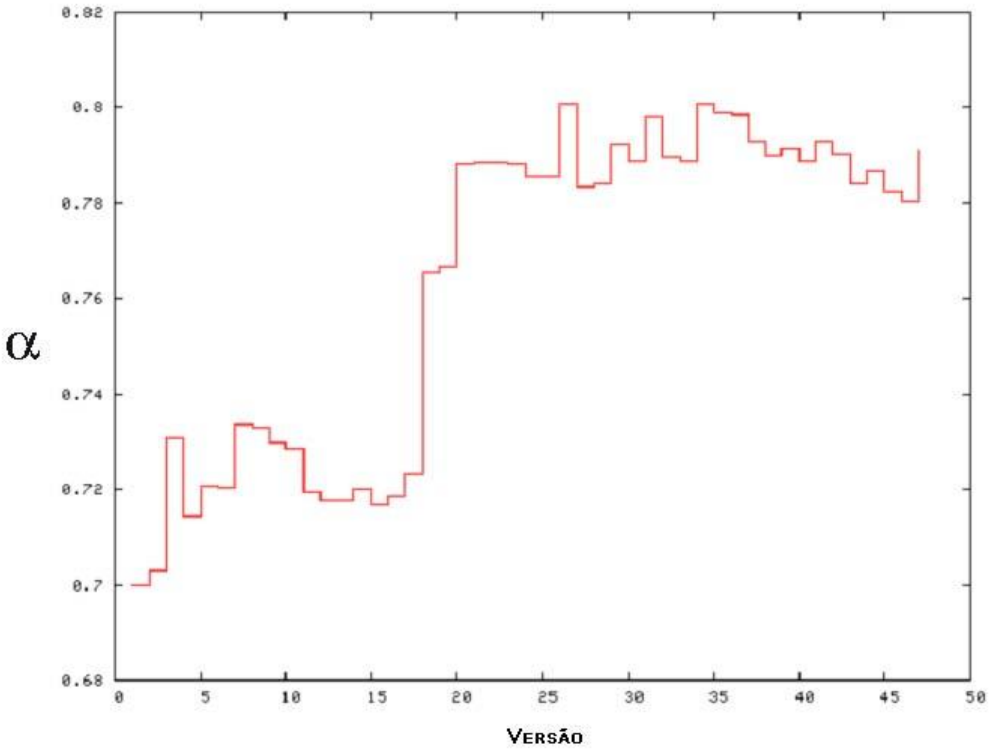


Figura. 7.5.7. α para o programa do terceiro tipo.

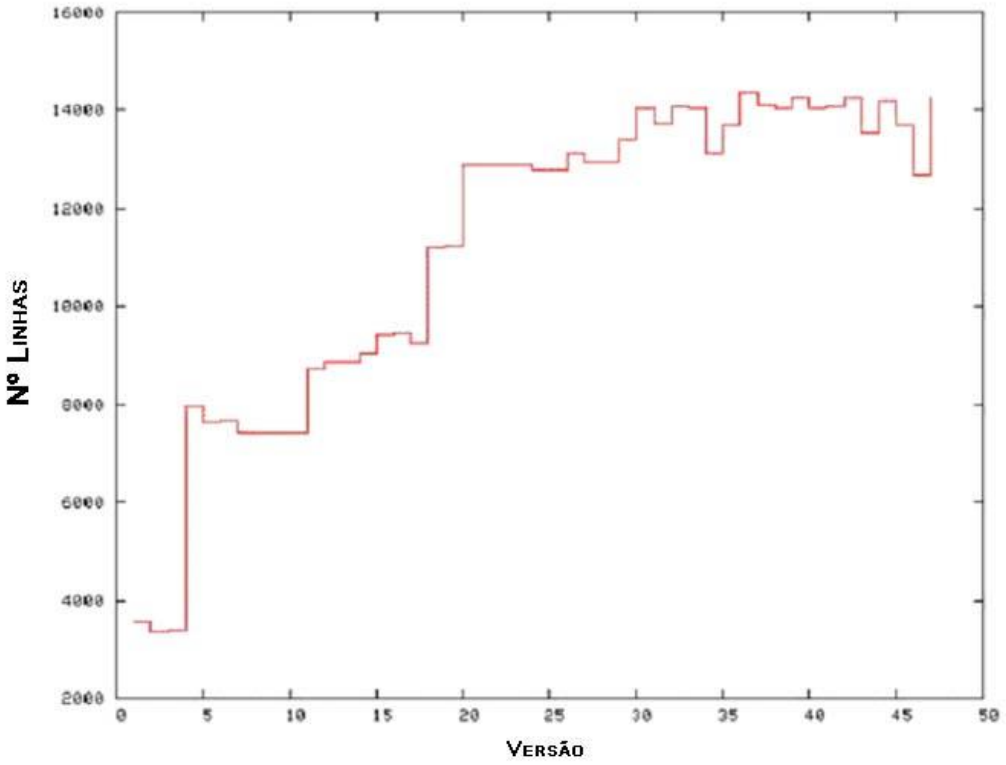


Figura. 7.5.8. numero de linhas para o programa do terceiro tipo.

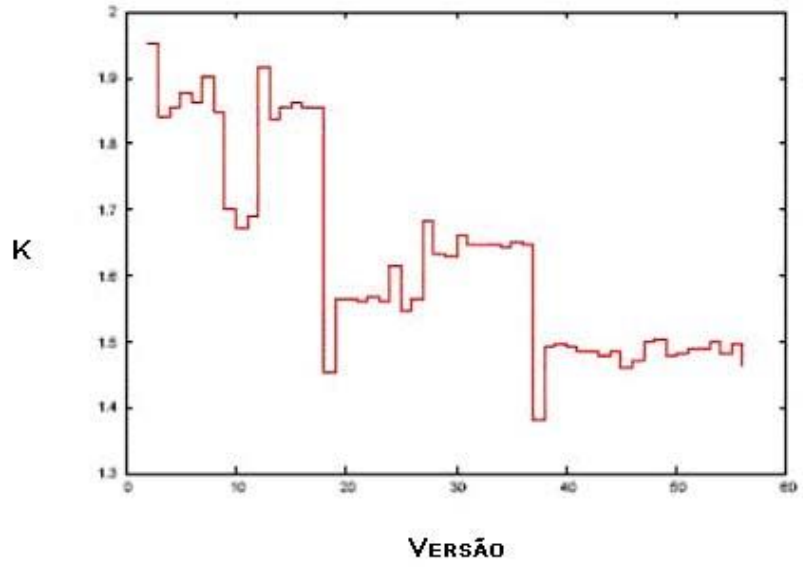


Figura. 7.5.9. k para o programa do terceiro tipo.

Os valores observados indicam um estado coincidente com o resultado dessa análise.

Nas figura 7.5.7 a 7.5.9.. estão representados os gráficos de variação de α, k e do número de linhas para as duas versões de um programa do grupo StatfullSessionInstanceInterceptor.

Os gráficos correspondentes a todos os programas, organizados por grupo, estão transcritos no anexo 7.5.1.

Os resultados indicam pois, a utilidade da utilização da correlação de gama longa no controle da evolução do processo de desenvolvimento do software.

7.6. Contribuição deste capítulo para a defesa da tese que é objecto desta dissertação

A contribuição deste capítulo para a defesa da tese, que é objecto desta dissertação, é a proposta da utilização da equação logística e da correlação de gama longa na modelação do processo de desenvolvimento.

O cálculo da correlação de gama longa para a sucessão de versões obtidas de um mesmo programa e a sua utilização na equação logística tornam possível determinar o comportamento do processo de desenvolvimento e assim sendo ajudar no seu controle.

Esta métrica pode ser calculada automaticamente, para cada versão e possibilita, através de uma única medida de uma série de versões, a obtenção de uma indicação acerca do progresso ou retrocesso do processo.

8. Conclusões e trabalhos futuros

...qual é a utilidade da nossa viagem? Na busca pelas razões profundas de o tempo só fluir num sentido e não noutro, viajámos até aos confins do tempo onde a própria noção de espaço se dissolveu. Aprendemos que as nossas teorias não são ainda adequadas para fornecerem as respostas (...)Devemos agora regressar a casa. O nosso caminho de regresso será muito mais especulativo que a viagem de ida, não existe outro caminho razoável...

Roger Pendrose in Mente Virtual

8.1. Introdução

Neste capítulo resumimos criticamente os objectivos deste trabalho e os contributos da escrita dos capítulos dois a sete desta dissertação para o estabelecimento da tese nela defendida. Enumeramos as conclusões globais da tese. Nas conclusões, destacamos a estratégia que usámos na investigação e os critérios que nortearam este trabalho.

Terminamos este capítulo referindo aspectos que não foram abordados e com a enumeração dos futuros desenvolvimentos que podem alargar o âmbito do trabalho realizado.

8.2. Motivação, objectivos, estratégia e conclusões

A motivação para a escrita desta dissertação foi identificar as razões pelas quais os vastos estudos realizados no âmbito da Engenharia do Software na área das Métricas de software são sistematicamente qualificados de pouco precisos e, como tal, menos úteis, quer pela indústria, quer pela comunidade científica.

Depois de estabelecida a hipótese que poderíamos estar a usar para o produto e para o processo modelos que não incluíssem atributos importantes, ou que analisassem o produto e o

processo de uma perspectiva redutora não tendo em conta interacções entre partes descritas separadamente, foi objectivo desta dissertação conhecer a complexidade da estrutura do produto de software como passo para poder controlar a dinâmica do processo de desenvolvimento do software a partir um modelo matemático.

Para tal, caracterizámos programas codificados em C como sistemas complexos e discutimos o uso de medidas de complexidade na avaliação da complexidade da estrutura desses programas. Verificamos, experimentalmente, as qualidades dessas medidas e escolhemos a correlação de gama longa como uma medida adequada para caracterizar a complexidade da estrutura do produto.

A escolha da medida foi orientada pelos critérios de caracterização global da complexidade do produto. A estratégia seguida para identificar as razões pelas quais os vastos estudos já mencionados são sistematicamente qualificados de pouco precisos, consistiu em estudar os modelos do processo e as métricas do produto disponíveis.
O estudo foi principalmente influenciado pelo trabalho de Many Lehman sobre as duas leis genéricas de evolução de software [Lehman,M.85], a *lei da mudança contínua* (que afirma que um produto em uso ou está em mudança constante ou se torna progressivamente menos útil) e a *lei do aumento da complexidade* (que afirma a mudança contínua introduz sistematicamente complexidade no produto, deteriorando a estrutura do mesmo).

Tendo ainda em conta os enunciados das leis *i) da evolução do produto* (que afirma que o processo de evolução do produto é uma dinâmica auto regulada com tendências estatísticas determináveis e invariantes), *ii)* da *conservação da estabilidade organizacional* (que afirma que durante a vida activa do produto a taxa de actividade global é invariante) *iii) da conservação da familiaridade* (que diz que durante a vida activa do produto o conteúdo de cada nova versão é estatisticamente invariante), optámos por procurar na teoria dos sistemas complexos a caracterização do produto (software) enquanto um sistema complexo e a tentar encontrar invariantes característicos na sua estrutura.

Passando ao estudo do processo, este foi objecto do segundo capítulo, no qual foram formuladas as seguintes observações:

- Ao longo do processo (considerando que o processo engloba a fase de evolução) vão sendo construídos vários modelos da solução (produtos) sob a influência da pressão dos utilizadores e fornecedores. Esta pressão é causada pela necessidade do produto ser mantido sem erros visíveis, adaptado ao meio (ele próprio em constante mudança) e às necessidades de novas utilizações e mudanças tecnológicas que vão surgindo. Assim o processo é dinâmico e adaptativo, apresentando vários estados diferentes ao longo da sua duração.
- A dinâmica do processo é não linear porque cada um dos seus estados é constituído por vários módulos interdependentes e hierarquizados por uma estrutura de controlo. Modificar um módulo pode ter consequências em vários pontos diferentes da estrutura e, como tal, os resultados dessas intervenções não são incrementais mas antes multiplicativos.
- Também a dinâmica organiza-se como foi descrito na secção 2.5.6. em dois processos que actuam de forma diferente: sequencial e iterativa havendo alterações em várias partes distintas do produto conforme o processo de modificação que é activado.

As observações acima apresentadas reforçam o princípio de que os produtos do processo de desenvolvimento de software podem ser modelados enquanto sistemas complexos.
Procurámos, então, indicadores na área das métricas de produto de forma a investigar a possível contribuição destes indicadores para a caracterização da sua complexidade.

Com o estudo das métricas, objecto do terceiro capítulo, concluiu-se que:

- Não conhecemos nenhuma medida que o caracterize completamente o produto. Todas as medidas disponíveis analisam atributos parcelares do produto e é difícil, através delas, caracterizar um produto globalmente, porque, muitas vezes, essas medidas tomam valores contraditórios.
- Também para nenhuma das medidas actualmente existentes existem valores padrões que indiquem a qualidade do produto e a sua utilidade. Não é possível, usando estas medidas, comparar dois produtos desde que sejam representados através de modelos diferentes.

Com o estudo dos indicadores da complexidade da estrutura do produto, objecto do quarto

capítulo, conclui-se que:

- A caracterização da complexidade da estrutura do produto faz–se com a identificação de regularidades
- Existiam várias técnicas de pesquisa de regularidades do produto.

Esta constatação levou a escolher um modelo de produto, os programas fonte codificados em linguagem C, e obter uma amostra representativa para experimentalmente, verificar a existência da regularidade na estrutura do produto de software e analisar o seu comportamento.

No quinto capítulo foi efectuada a análise qualitativa da amostra recolhida. Este conjunto pode ser descrito como constituído por dois grupos fundamentais: o grupo dos *compiladores* e o grupo dos *não compiladores*. Neste último encontram-se programas que implementam diversas funcionalidades. Os programas foram escritos por diferentes programadores. Na sua totalidade, o conjunto de programas corresponde a cerca de 20.000 linhas de código. A variedade dos programas e o tamanho da amostra garantem a sua representabilidade estatística.

O primeiro grupo é mais homogéneo, sendo constituído por 36 programas escritos em linguagem C, que implementam um compilador de uma linguagem estruturada em blocos. Os referidos programas foram construídos por alunos do Instituto Superior Técnico.

Foram ainda considerados três grupos de controlo da amostra, obtidos através de geradores e como tal com características impostas na geração.
Assim trabalhámos com os seguintes grupos:

Grupo A: um conjunto de programas gerados a partir dos programas do primeiro grupo fundamental, sendo cada um dos programas do grupo A o resultado da sequência aleatória das palavras-chave de cada um dos programas do primeiro grupo fundamental.
Grupo B: um conjunto de 36 programas gerados, com o mesmo número de linhas e sem a mesma distribuição de palavras-chave encontradas nos programas do primeiro grupo

fundamental e construídos de forma a garantir em cada um dos programas do grupo B:

- a ausência de erros de sintaxe
- e o não desempenho de qualquer função.

Grupo C: um conjunto de 36 programas gerados a partir do mesmo conjunto de palavras-chave encontradas nos programas do primeiro grupo fundamental e construídos de forma a garantir em cada um dos programas do grupo C:

- a ausência de erros de sintaxe,
- o não desempenho de qualquer função e
- a mesma distribuição de frequências de palavras-chave do segundo grupo fundamental.

A estratégia usada no quinto capítulo foi a de procurar regularidades na estrutura dos programas através da aplicação das medidas descritas no capítulo quarto.
Foi verificado que:

- a classe funcional dos compiladores escritos em linguagem C tem uma distribuição de frequências de palavras-chave, menos dispersa e distinta da distribuição de frequências de palavras-chave característica de um conjunto significativo de outros programas escritos em linguagem C,
- a distribuição dos símbolos menos frequentes dos compiladores escritos em linguagem C é variável e, como tal, não pode ser usada para caracterizar esta classe funcional dos compiladores,
- a lei de Zipf ajustada foi verificada para a distribuição de frequências das palavras-chaves dos compiladores escritos em linguagem C, dos programas com várias funcionalidades e dos programas dos grupos de controlo, não sendo pois de grande utilidade na sua discriminação,
- a medida da entropia de Shannon não discrimina a existência de estrutura sintáctica dos programas escritos em C,
- a correlação de gama longa discrimina a existência de estrutura sintáctica dos programas escritos em C,
- a correlação de gama longa não discrimina diferentes estruturas funcionais dos programas

escritos em C. Embora a correlação de gama longa esteja presente com um valor elevado nos programas com estrutura funcional, também pode surgir em programas sem essa estrutura desde que gerados com limitações que obriguem a uma aproximação a uma estrutura funcional.

Dispomos pois de uma medida que representa a complexidade da estrutura do produto independente da funcionalidade do mesmo.

Tentámos ainda, no capítulo seis separar os programas do primeiro grupo fundamental da amostra (compiladores) em relação ao número de funcionalidades que implementam.

Como a correlação de gama longa não permite tal discriminação efectuámos uma análise taxonómica, cujos resultados evidenciam que o grupo é homogéneo, quando medido pelo conjunto das métricas apresentadas no capítulo 4. O uso dessa técnica não nos permitiu efectuar a classificação pretendida.

Seguidamente, procurámos uma estratégia alternativa. Não utilizámos os resultados das métricas apresentadas no capítulo 4, para caracterizar cada compilador. Usámos uma técnica de identificação de intrusão de vírus que fundamenta a construção do chamado "*self* " de cada compilador. Verificamos que os "*self*s" dos dois grupos mais afastados em termos de funcionalidade dentro da amostra (grupo com classificação 0 e grupo com a classificação 4) não se podem considerar diferentes. Tal como a análise taxonómica, esta técnica não nos permitiu efectuar a distinção entre compiladores com diferentes números de funcionalidades.

Concluímos, então, que as técnicas de agrupamento de dados, não são sensíveis ao número de funcionalidades que um compilador implementa.

Apresentámos no capítulo sete uma aplicação do cálculo da correlação de gama longa para controlo do processo desenvolvimento de software.

Para tal, apresentamos brevemente os conceitos que caracterizavam a dinâmica dos sistemas complexos e analisámos a equação logística, que modela a evolução de populações competindo por recursos limitados.

De seguida, aplicámos esta equação como modelo da dinâmica de um processo de desenvolvimento de software real e exemplificámos como é possível determinar o estado do processo, calculando a correlação de gama longa para a sucessão de versões obtidas.

Finalmente apresentámos resultados obtidos sobre uma amostra de cerca de 43 programas e suas versões, onde se evidencia a utilidade do modelo referido para controlar o processo de

desenvolvimento de software.

Concluímos da utilização da equação logística e da correlação de gama longa para na modelação da dinâmica de um processo desenvolvimento de software, tornam possível determinar o comportamento do processo e assim sendo ajudar no seu controlo.

Esta métrica pode ser calculada automaticamente, para cada versão e possibilita, através de uma única medida de uma série de versões, a obtenção de uma indicação acerca do progresso ou retrocesso do processo.

A inovação desta dissertação sintetiza-se na consideração o produto de software enquanto um sistema complexo e na aplicação de um conjunto de medidas utilizadas na teoria dos sistemas complexos com a finalidade de caracterizar a estrutura de programas fontes codificados numa linguagem de programação. Concluímos que a medida mais adequada para a caracterização do produto é a da correlação de gama longa. Concluímos ainda que esta medida é sensível a modificações na estrutura sintáctica dos programas e à localização (ou concentração) de palavras-chave dentro da especificação dos mesmos. Verificamos também que a medida da correlação de gama longa não permite concluir sobre a existência de diferentes estruturas funcionais (número e diversidade de funcionalidades de cada programa).

Verificámos, finalmente, que a utilização da equação logística na modelação da dinâmica do processo permite ajudar no seu controlo.

8.3. Análise Crítica

É necessário esclarecer as limitações e as fraquezas que alguns assuntos incluídos nesta dissertação apresentam. A identificação desses pontos indica necessidade de mais esforço de investigação que já não foi possível incluir nesta tese. Listamos de seguida as referidas limitações.

Em relação ao processo, a sensibilidade às condições iniciais foi assumida, na secção 2.5.6., por analogia baseada em constatações de uma experiência longa de vinte anos no controlo de projectos de software em actividade comercial. Embora esta verificação experimental tenha indicado o caminho para hipóteses que confirmaram o carácter de sistema complexo ao processo e, como tal, de sistema sensível a condições iniciais esta constatação será mais forte quando forem medidas ao longo da dinâmica do sistema indicadores dessa sensibilidade.

Em relação à correlação de gama longa, o estudo desta medida apenas em duas linguagens é útil como guia de identificação de hipóteses e estabelecimento de regras para obtenção de

amostras de outras linguagens. Os programas construídos para medição são facilmente aplicados a outros conjuntos de palavras-chaves de variadas linguagens. Todavia, o estudo feito sobre apenas duas linguagens representa uma limitação e não permite a visão global da mesma estrutura codificada em várias linguagens.

Na análise da estrutura, esta restrição foi imposta pela necessidade de analisar uma amostra constituída por programas que tivessem a mesma funcionalidade, fossem escritos de forma controlada mas por equipas diferentes e fossem avaliados segundo um mesmo critério. O uso de uma amostra deste tipo permitiu controlar o tipo de estrutura que medimos sem restringir a diversidade dos programas medidos.

Na análise da dinâmica, esta restrição foi imposta pela disponibilidade de documentação de processos reais com tamanho suficiente.

Também o uso de uma amostra de uma determinada classe funcional de programas, os compiladores, restringe o âmbito da formulação das hipóteses a essa classe funcional.

Esta restrição foi imposta pela impossibilidade de encontrar uma amostra que implementasse outra funcionalidade e os programas que a constituíssem fossem escritos de forma controlada, por equipas diferentes e conhecêssemos a avaliação da sua funcionalidade segundo um mesmo critério. A segunda amostra analisada representa, todavia, a confirmação da possibilidade de utilização desta medida para outras linguagens distintas do C.

Na secção 8.4 são listadas algumas propostas de investigação, orientadas à resolução dos problemas levantados nesta secção.

8.4. Trabalhos futuros

Ao escrever a última secção desta dissertação temos a consciência de que o que foi feito é só o início de um trabalho. Todavia, agora conseguimos avaliar o esforço que terá de ser realizado futuramente. Passamos a descrever as linhas que serão a orientação do nosso trabalho de investigação nos anos futuros.

8.4.1.Dinâmica do processo

O grupo de investigação a que pertenço estuda, desde algum tempo, um modelo da dinâmica do processo. Embora já tenhamos uma casuística analisada com alguma dimensão muito há ainda a estudar nesse campo.

Será necessário usar a correlação de gama longa como medida global caracterizadora do estado do sistema (da complexidade da estrutura do produto) e verificar como essa

complexidade da estrutura evolui ao longo de vários processos de actualização de sistemas.
Já iniciámos esses estudos procurando amostras de processos disponíveis escritos em várias linguagens. Actualmente dispomos de um processo constituído pelas várias versões do Linux e estamos a analisar como foram obtidas. Esse estudo consiste em obter informações sobre as equipas construtoras, sobre o que levou à introdução das modificações, acrescentos de funcionalidade, sobre os ambientes de trabalho e indicadores de qualidade de cada versão. O projecto está a decorrer sob a designação ISoQ-Improving Software Quality by Mastering Complexity, número SloP-10/01-04, com financiamento da FCT.

Consideramos ainda necessário analisar outros processos, que não a construção de um sistema operativo, e estamos a realizar a avaliação dos critérios de escolha desse processo.
Com uma equipa de investigação constituída pelo professor Peter Kokol da universidade de Maribor na Eslovénia e com o Professor Gustavo Crespo do IST Portugal estamos a definir metodologias que permitam medir a correlação de gama longa em diversos modelos do produto. Estudamos a forma de medir a correlação de gama longa (a invariância partilhada por todos os níveis da estrutura) no conjunto dos requisitos que definem o sistema. Nesta dissertação construímos a metodologia para medir a complexidade da estrutura dos programas fonte. A equipa coordenada pelo Professor Peter Kokol formulou uma metodologia para medir a complexidade da estrutura dos programas objecto. Neste trabalho já utilizámos essa metodologia que descrevemos na secção 5.5.
Será agora necessário unificar os três modelos: requisitos, os programas fonte e programas objecto de vários processos e investigar como é que a complexidade da estrutura da descrição do problema descrita em modelos diferentes evolui. Nomeadamente, é importante investigar se a complexidade obtida pelos requisitos é ou não determinante para a complexidade dos modelos da solução, posteriores, construídos com base nos requisitos. Tal constatação seria de grande utilidade porque permitiria medir a estrutura para a qual caminhava a solução numa fase inicial do processo.
A comprovar-se tal hipótese seria possível usar esta medida no apoio às tomadas de decisão durante o processo de desenvolvimento de produtos comerciais.
Nesta área existe também algum trabalho já publicado. Foi incluído numa uma colectânea de artigos, um capítulo sobre a evolução da complexidade nas versões de produtos gerados na indústria [Cardoso, A.04]. Noutro artigo descreve-se como a evolução é modelada pela

equação logística, sendo identificados vários casos de comportamento evolutivo nomeadamente caótico e estável.[Kokol,P.00].

8.4.2.Linguagens de programação

Não foi analisada uma amostra de programas com a mesma funcionalidade mas que implementassem outra funcionalidade que não fosse um compilador. Para podermos concluir com alguma validade se a correlação medida para programas escritos numa linguagem, está dependente do tipo de funcionalidade teríamos de estudar amostras de tamanho equivalente de programas construídos pelas mesmas equipas, em que cada amostra correspondesse à implementação de uma funcionalidade diferente.

Ficou também por analisar se há, ou não, um valor característico de cada linguagem correspondente a uma boa estruturação e se esse valor difere de linguagem para linguagem. Para tal teríamos de dispor de amostras de programas fontes codificados em linguagens diferentes que implementassem o mesmo conjunto de requisitos e que as várias equipas tivessem conhecimento idêntico de estruturas de programação e da linguagem que utilizassem.

Os programas são, preferencialmente, escolhidos no mundo académico. É de toda a conveniência confirmar os resultados, descritos nesta dissertação, em sistemas desenvolvidos no mundo comercial.

9. Bibliografia

[Abreu,F.94]

Abreu, F. B.; *A Qualidade na Produção do Software*, INA, 1994.

[Albrecht,A.79]

Albrecht,A.J.;*Measuring application development productivity*, IBM Applications Development Joint SHARE/GUIDE Symposium, 1979.

[Arabi,L.96]

Arabi,L. and all; *Clustering and Classification*, World Scientific, 1996.

[Ashkenazy Y.01]

Y. Ashkenazy, P. Ch. Ivanov, S. Havlin, C.-K. Peng, A. L. Goldberger, and H. E. Stanley, *Magnitude and Sign Correlations in Heartbeat Fluctuations,* Phys. Rev. Lett. 86, 1900-1901, 2001.

[Balcázar, J.88]

Balcázar, J. L. Díaz, J. and Gabarró J.; *Structural Complexity I* , Springer Verlag, 1988.

[Bache,R.90]

Bache, R.; *Graph Theory Models of Software*, PhD thesis, South Bank University, London, 1990.

[Bache,R.90A]

Bache, R. and Mullerburg, M.; *Measures of testability as a basis for quality assurance*, Software Engineering Journal, 5(2), 86-92, 1990.

[Balzer, R.81]

Balzer, R..;*Transformational implementation, an example,* Transactions on Software Engineering, 7 (1) 3-4, 1981.

[Barabási,A.00]

Barabási,A.; *Scale-free Characteristics of Random Networks: the Topology of the World Wide Web*, Physica A 281, 69-77, 2000.

[Berg,K.95]

Van den Berg, K.G. and van den Broek, P.M.; *Static analysis of functional programmes*, Information and Software Technology, 37(4), 213-224, 1995.

[Bieman,N.90]

Bieman,N.;*A Tool for Estimating Software Testing Requirements*, Microcomputer Applications, 9(3), 72-79, 1990.

[Boehm,B.81]

Boehm,B.; *Software Engineering*, Prentice -Hall, Englewood Cliffs, 1981.

[Bohem,B.88]

Bohem,B.W.; *A spiral model of software development and enhancement*, IEEE Computer 21(5), 61-72, 1988.

[Booch,G.95]

Booch,G.; *Managing the Object Oriented Project*, Addison Wesley, 1995.

[Bőrger,E.89]

Bőrger,E.; *Computability, Complexity, Logic*, North Holland ,Amsterdam, 1989.

[Briand,L.95]

Briand, L. et all; On *Application of Measurement Theory in Software Engineering*, ISERN Tecnical Report nº95-04, 1995.

[Briand,L.97]

Briand, L. et all; *Qualitative Analysis for Maintenance Process*, NASA ,1997.

[Buldyrev, S. V.93]

S. V. Buldyrev, A. L. Goldberger, S. Havlin, C.-K. Peng, M. Simons, and H. E. Stanley, *Generalized Levy Walk Model for DNA Nucleotide Sequences*, Phys. Rev. E 47, 4514-4523, 1993.

[Buldyrev, S. V.95]

S. V. Buldyrev, A. L. Goldberger, S. Havlin, R. N. Mantegna, M. E. Matsa, C.-K. Peng, M. Simons, and H. E. Stanley, *Long-Range Correlation Properties of Coding and Noncoding DNA Sequences: GenBank Analysis*, Phys. Rev. E 51, 5084-5091, 1995.

[Caldas,J.91]

Caldas,J.M.; *Cinco métodos de classificação na óptica da optimização combinatória*, Documento de trabalho 2-91 do CEMAPRE, Instituto Superior de Economia e Gestão, Universidade Técnica de Lisboa, 1991.

[Campos,L.99]

Campos,L et all; *Programação em Visual Basic*, FCA Editora de Informática, 1999.

[Cardoso, A.00]

Cardoso, A.I., and Crespo, R.G., Kokol, P.; *Two different views about Software Complexity*; Proceedings of European Software Control and Metrics Conference, 433-438, Shaker, Munique, 2000.

[Cardoso, A.00a]

Kokol,P.;Podgorelec,V. Cardoso,A.; Assessing the state of the software process using chaos theory; Software Engineering notes. vol.25, Pag. 41-44 ACM Press, 2000.

[Cardoso, A.01]

Cardoso, A.I., Crespo, R.G., Kokol, P.; *An Alternative Way to Measure Software – A Measure from Complex System Theory*, Proceedings of World Multiconference on System Cybernectics and Informatics, IFSR and IEEE Computer Society, Orlando, 213-216, 2001.

[Cardoso, A.04]

Cardoso,A.I.,Kokol,P.,Lenic,M.,Crespo,R.G; *Complexity-based Evaluation of Systems Evolution in Advances in UML/XML based Software Evolution*, IRM Press,2004.

[Chaitin,G.66]

Chaitin, Gregory J.; On *The Length Of Programs For Computing Finite Binary Sequences*, Journal of the ACM, 13, 547-569, 1966.

[Cohen,B.86]

Cohen, B., Harwood W. T., Jackson M. I.: *The specification of complex systems*, Addison Wesley, 1986.

[Conte,S.86]

Conte S.D., Dunsmore H.F., Shen V.Y.; *Software engineering metrics and models*, Benjamin/Cummings, Menlo Park, 1986.

[Coulter,N.83]

Coulter,N.; *Software Science and Cognitive Psychology*, IEEE Transactions on Software Engineering, SE9 (2), 166-171, 1983.

[D'Haeseleer,P.76]

D'Haeseleer,P.; *An Immunological Approach to Change Detection: Algorithms, Analysis and Implications*, Proceddings of IEEE Symposiumon Security and Privacy, 43-52, 1996.

[Dorogovtsev,S.00]

Dorogovtsev,S. et all,; *Scalling Behaviour of Developing and Decaying Networks*, arXiv:cond-mat / 0005050, 2, May 2000.

[Dromey,R.96]

Dromey, R. Geoff; *Cornering the chimera*, IEEE Software, 13(1), 33-43, 1996.

[Edmonds,B.00]

Edmonds,B.; *Syntactic Measures of Complexity*, Doctoral Thesis, University of Manchester, Manchester, UK. 1999.

[Elliot,J.88]

Elliot, J.J.; *Data complexity aspects of software*, Alvey Project SE/69, PRRM South Bank Polytechnic, London, 1988.

[Fenton,N.86]

Fenton, N. and Pfleeger E.; *Axiomatic Approach to Software Metrication through Program Decomposition*, Computer Journal, 29(4), 330-339, 1986.

[Fenton,N.97]

Fenton, N. and Pfleeger E.; *Software Metrics A Rigorous & Pratical Approach*, PWS Publishing Company, Boston, 1997.

[Forrest,S.94]

Forrest,S. et all; *Self –Nonself Discrimination in a computer*, National Science Fundation, IRI 9157644, 1994.

[Gabaix,X.03]

X. Gabaix, P. Gopikrishnan, V. Plerou, and H. E. Stanley, *A Theory of Power-Law Distributions in Financial Market Fluctuations,* Nature 423, 267-270, 2003.

[Gabaix,X.03a]

X. Gabaix, P. Gopikrishnan, V. Plerou, and H. E. Stanley, *A Theory of Large Fluctuations in Stock Market Activity*, MIT Department of Economics Working Paper Series: Working Paper 03-30, 16 August 2003.

[Garnett,P.97]

Garnett,P. W.; *Chaos Theory Tamed*, Joseph Henry Press,1997.

[Gell-Mann,95]

Gell-Mann,M.; *What is Complexity*, Complexity, John Willey and Sons, Inc, Vol1, nº1, 16-19 1995.

[German Ministry of Defense92]

German Ministry of Defense; *V-model: Software Lifecycle process model*, General Reprint Nº250, 1992.

[Gopikrishnan, P.99]

P. Gopikrishnan, V. Plerou, L. A. N. Amaral, M. Meyer, and H. E. Stanley, *Scaling of the Distributions of Fluctuations of Financial Market Indices*, Phys. Rev. E 60, 5305-5316, 1999.

[Grady,R.92]

Grady,R.B. et all; *Practical Software Metrics for Project Management and Project Improvement*, Prentice Hall,1992.

[Halstead,M.77]

Halstead, M. H.; *Elements of Software Science*, Prentice-Hall, Inc., New York, 1977.

[Harel,D.92]

Harel,D.; *Algorithmics*, Addison-Wesley, Reading, MA, 1992.

[Hartigan, J.75]

Hartigan, J.; *Clustering Algorithms*, John Wiley and Sons, New York, 1975.

[Havlin,S.95]

S. Havlin, S. N. Buldyrev, A. L. Goldberger, R. N. Mantegna, C.-K. Peng, M. Simons, H. E. Stanley; *Statistical properties of DNA sequences, Fractal Reviews, in the Natural and Applied Science* . Ed., M. M. Novak. Chapman & Hall, London , 1995.

[Hill,P.99]

Hill, P.; Edited and compiled by: *Software Project Estimation: A Workbook for Macro-Estimation of Software Development Effort and Duration*, ISBSG, 1999.

[Humphrey,W.01]

Winning with Software: An Executive Strategy, Addison Wesley Professional, 2001.

[IEEE83]

Standart Glossary of Software Engineering Terminology, IEEE Std.723, 1983.

[IEEE,90]

IEEE Standart Glossary of Software Engineering Terminology, IEEE inc.,1990.

[INSEAD,01]

2000 ESA/INSEAD Data Analysis Report, Center for Research in Information Systems Excellence (RISE), May 21, 2001.

[Jacobson,I.99]

Jacobson, I., G. Booch, and J. Rumbaugh,; *The Unified Software Development Process*.,USA: Addison Wesley Longman, Inc., 1999.

[Jain,A.88]

Jain,A. and all; *Algorithm for clustering data*, Prentice Hall, 1988.

[Kernighan,B.88]

Kernighan,B. and all; *The C Programming Language*, Prentice Hall, 1988.

[Kirkman,G.02]

The Global Information Technology Report 2001-2002, Center for International Development at Harvard University, 2002.

[Kitchenham,B.98]

Kitchenham,B.; *The Certainty of Uncertainty*, Proceedings Fesma 98, 17-25, 1998.

[Kokol,P.99]
Kokol P et all; *Computer and natural language texts – a comparison based on long range correlations*, Journal of the American Society for Information Science, 1295-1301, 1999.
[Kokol,P.00]
Kokol, P., Podgorelec V., Cardoso, I., Dion, F.; *Assesing the State of the Software Process Development Using the Chaos Theory*, Software Engineering Notes, ACM Press, 25, (3) 41-43, 2000.
[Kokol,P.01]
Kokol P., Brest J, Vumer V.; *Long Range Correlation in Computer Programs*, Cybernetics and Systems, 43-57, 2001.
[Kolmogorov,A.58]
Kolmogorov,A.N.;*A new metric invariant of transitive dynamical systems and automorphisms in Lebesgue spaces*, Dokl Acad Nauk SSSR 119: 861-864, 1958.
[Lehman,M.85]
M.M.Lehman and all; *Program Evolution*, Academic Press, London, 1985.
[Li,W.97]
Li, W.; *The Study of Correlation Structures of DNA Sequences: A Critical Review*. published Computer & Chemistry, 342-356, 1997.
[Lindholm,99]
Lindholm, T. and Yellin, F.; *The Java(TM) Virtual Machine Specification*, Addison-Wesleyl, 1999.
[Mantegna, R. N.95]
R. N. Mantegna, S. V. Buldyrev, A. L. Goldberger, S. Havlin, C.-K. Peng, M. Simons, and H. E. Stanley, *Systematic Analysis of Coding and Noncoding DNA Sequences Using Methods of Statistical Linguistics*, Phys. Rev. E 52, 2939-2950, 1995.
[Matia,K.03]
K. Matia, Y. Ashkenazy, and H. E. Stanley, *Multifractal Properties of Price Fluctuations of Stocks and Commodities*, Europhys. Lett. 61, 422-428, 2003.
[MatLab,01]
MATLAB Notebook Version 1.5; www.mathworks.com.
[McCabe,J.89]
McCabe,T.J. and all; *A complexity measure.*IEEE transactions on Software Engineering, 2 (4): 308-320, 1985.

[Mills,H.86]

Mills,H.D.and all; *Strutured Programing:retrospect and prospect*, IEEE Software, 3(6), 58-66, 1986.

[Morowitz,H.88]

Morowitz,H.; *The Emergence of Complexity*, Complexity 1(1): 4, 1995.

[Mossa,S.02]

S. Mossa, M. Barthelemy, H. E. Stanley, and L. A. N. Amaral, *Incomplete Information and the Growth of Scale-Free Networks: The `Cost of Information'*, Phys. Rev. Lett. 88, 138701-1, 138701-4, 2002.

[Naur,P.69].

Naur,P.e all; *Software Engineering, Report of Nato Science Committee, in Garmish Conference*, Scientific Affairs Division Nato Brussels, 1969.

[Paulk,91]

Paulk and all; *Capability Maturity Model for Software*, CMU/SEI-91-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1991.

[Peng,C.92]

C-K. Peng, S.V. Buldyrev, A.L. Goldberger, S. Havlin, F. Sciortino, M. Simon, and H.E. Stanley; *Long-range Correlations in Nucleotide Sequences* Nature, 356, 168-170, March 12, 1992.

[Peng, C. K.93]

C. K. Peng, S. V. Buldyrev, A. L. Goldberger, S. Havlin, M. Simons, and H. E. Stanley, *Finite Size Effects on Long-Range Correlations: Implications for Analyzing DNA Sequences*, Phys. Rev. E 47, 3730-3733, 1993.

[Peng C. K.94]

C. K. Peng, S. V. Buldyrev, S. Havlin, M. Simons, H. E. Stanley and A. L. Goldberger, *Mosaic Organization of DNA Nucleotides*, Phys. Rev. E 49, 1685-1689, 1994.

[Pfleeger,S.98]

Pfleeger,S.; *Software Engineering: Theory and Practice*, Prentice Hall, 1998.

[Pfleeger,S.98a]

Pfleeger,S.L.and McGowan, C.; *Software Metrics in the process maturity frame work*, Journal of Systems and Software, 12, 255-261, 1998.

[Plerou,V.00]

V. Plerou, P. Gopikrishnan, B. Rosenow, L. A. N. Amaral, and H. E. Stanley, *Econophysics: Financial Time Series from a Statistical Physics Point of View*, Physica A 379, 443-456, 2000.

[Pressman,R.94]

Pressman, R. S.; *Software Engineering, A Practitioner's Approach*, McGraw Hill, 1994.

[Reifer,D.98]

Reifer,D.J.; *Software Management*, fifth edition, IEEE Computer Society Press, 1998.

[Royce,W.87]

Royce,W.W.; *Managing the development of large software systems: Concepts and technologies*, Proceedings of ICSE1987, 328-339, 1987.

[Schenkel,A.92]

Schenkel A, Zhang J, Zhang Y.; *Long range correlations in human writings*, Fractals 1(1):47-55, 1993.

[Schildt,H.86]

Schildt,Herbet; *C Avançado*, MacGraw-Hill, 1986.

[Schulte-Frohlinde,V.02]

V. Schulte-Frohlinde, Y. Ashkenazy, A. L. Goldberger, P. Ch. Ivanov, M. Costa, A. Morley-Daview, H. E. Stanley, and L. Glass, *Complex Patterns of Abnormal Heartbeats*, Phys. Rev. E 66, 031901, 2002.

[Shanon,C.49]

Shanon,C.; *A mathematical theory of communication*, Bell Syst .Tech. Journal 27, 379-423, 1949.

[Shepperd, M.93]

Shepperd, M.J. and Ince, D.; *Derivation and Validation of Software Metrics*, Clarendon Press, Oxford, UK, 1993.

[Sommerville,I.96]

Ian Sommerville; *Software Engineering*, Addison-Wesley Publishing Company, 1996.

[SPC,95]

Software Produtivity Consortio; Extend of reuse, NASA, 1995.

[Stanley,H.E. 00]

H. Eugene Stanley; *Exotic statistical physics: Applications to biology, medicine, and economics* Physica A: Statistical Mechanics and its Applications 285(1-2), 1-17, 2000.

[Stanley, H.E.02]

H. E. Stanley, L. A. N. Amaral, S. V. Buldyrev, P. Gopikrishnan, V. Plerou, and M. A. Salinger,; *Self-Organized Complexity in Economics and Finance*, Proc. Natl. Acad. Sci. 99-Supp, 2561-2565, 2002.

[Stanley,H.E. 94]

H. E. Stanley, S. V. Buldyrev, A. L. Goldberger, S. Havlin, R. N. Mantegna, S. M. Ossadnik, C.-K. Peng, F. Sciortino, and M. Simons; *Fractals in biology and medicine - Diffusion Processes: Experiment, Theory, Simulations , Proceedings*, Kudowa, Poland; Lecture Notes in Physics 438, 147, 1994.

[Stanley, H.E.99]

H. E. Stanley, L. A. N. Amaral, D. Canning, P. Gopikrishnan, Y. Lee, and Y. Liu, *Econophysics: Can Physicists Contribute to the Science of Economics*? Physica A 269, 156-169, 1999.

[Suki,B.03]

B. Suki,A. M. Alencar, U. Frey, P. Ch. Ivanov, S. V. Buldyrev, A. Majumdar, H. E. Stanley, C. A. Dawson, G. S. Krenz, and M. Mishima, *Fluctuations, Noise, and Scaling in the Cardio-Pulmonary System*, Fluctuations and Noise Letters 3, R1-R25, 2003.

[Tarski,W.99]

Tarski,W.et all; *Specification of computer Programs*, Addison-Wesley, 1999.

[Vilela Mendes,R.99]

Vilela Mendes,R.; *Medidas de complexidade e auto organização*, Colóquio Ciências Fundação Gulbenkian, 1999.

[Vilela Mendes,R.03]

Vilela Mendes, R., Araújo, T., Louçã F.; *Reconstructing an economic space from a market metric*ics : Physica, A, 323, 635-655, 2003.

[Wegner,P.95]

Wegner P.; *Symposium on Computational Complexity and the Nature of Computer Science*, Israel M (Eds.): Computing Surveys, 27(1), 5-62, 1995.

[Weyuker,E.88]

Weyuker, E.K.; *Evaluation software complexity measures*, IEEE Transactions on Software Engineering, SE-14(9), 1357-1365, 1988.

[Whitty,R.90]
Whihy, R.W.and Lockhart, R.; Structural Metrics, Esprit 2/Cosmos document GC/WPI/REP/7.3, Goldsmith College, London, 1990.
[Woodward,M.93]
Woodward,M.R.; *Difficulties using cohesion and coupling as quality indicators*, Software Quality Journal, 2(2), 109-128, 1993.
[Yang,H.03]
Successful Evolution of Software Systems, Artech House, 2003.
[Yourdan,E.79]
Yourdon, E. and Constantine, L.L.; *Structured Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.
[Zang,C.91]
Zang,C.; *Complexity and 1/f Noise: a Phase Space Approach*, J. Phys. I1, 971-993, 1991.
[ZIF,00].
The Sciences of complexity: From Mathematics to a sustainable worl, Program definition document, http:www.uni-bielefeld.de/complexity/introduction.html
[Zuse,H.91]
Zuse, H.; *Software Complexity: Measures and Methods*, De Gruyter, Berlin, 1991.
[Zuse,H.98]
Zuse H.; *History of Software Measurement*, De Gruyter, Berlin, 1998.

Anexo 1. da Secção 5.2.

(ANEXO 5.2.1.)

O Enunciado do problema

Anexo 2. da Secção 5.2.

(ANEXO 5.2.2)

A “Makefile” de um compilador da amostra analisada

Makefile Compilador 1

```
CC = gcc

all:
   rm -rf a.out core *.o zeta lex.yy.c zeta.tab.c zeta.tab.h zeta.output
   bison -d -v zeta.y
   flex zeta.l
   $(CC) -g -o zeta hash.c c3e.c defs.c lex.yy.c zeta.tab.c -lfl

clean:
   rm -rf a.out core *.o zeta lex.yy.c zeta.tab.c zeta.tab.h zeta.output c3e.out
```

Anexo 1. da Secção 5.3.

(ANEXO 5.3.1)

Programas utilizados

1. **Keywords**
    i. **Keywords.l**
    ii. **Keywordsdlg.ccp**
    iii. **Keywords.ccp**
2. **Lcr**
3. **Ale**
4. **Gramática dos geradores**
5. **Gerador1**
6. **Gerador2**
7. **Clusters**

1. Keywords

1.1 Keywords.l

```
%{
#include <stdio.h>

FILE* istream = NULL;
FILE* ostream = NULL;

void comments();

%}

%%

"/*"   { comments();               }
\;     { fprintf( ostream, "%s\n", yytext ); }
\,     { fprintf( ostream, "%s\n", yytext ); }
\=     { fprintf( ostream, "%s\n", yytext ); }
\{     { fprintf( ostream, "%s\n", yytext ); }
\(     { fprintf( ostream, "%s\n", yytext ); }


asm      { fprintf( ostream, "%s\n", yytext ); }
bad_cast { fprintf( ostream, "%s\n", yytext ); }
bad_typeid { fprintf( ostream, "%s\n", yytext ); }
bool   { fprintf( ostream, "%s\n", yytext ); }
catch    { fprintf( ostream, "%s\n", yytext ); }
const_cast { fprintf( ostream, "%s\n", yytext ); }
delete   { fprintf( ostream, "%s\n", yytext ); }
dynamic_cast { fprintf( ostream, "%s\n", yytext ); }
except   { fprintf( ostream, "%s\n", yytext ); }
explicit { fprintf( ostream, "%s\n", yytext ); }
finally    { fprintf( ostream, "%s\n", yytext ); }
friend   { fprintf( ostream, "%s\n", yytext ); }
inline   { fprintf( ostream, "%s\n", yytext ); }
mutable    { fprintf( ostream, "%s\n", yytext ); }
namespace  { fprintf( ostream, "%s\n", yytext ); }
new      { fprintf( ostream, "%s\n", yytext ); }
operator { fprintf( ostream, "%s\n", yytext ); }
private    { fprintf( ostream, "%s\n", yytext ); }
protected  { fprintf( ostream, "%s\n", yytext ); }
public   { fprintf( ostream, "%s\n", yytext ); }
reinterpret_cast { fprintf( ostream, "%s\n", yytext ); }
template { fprintf( ostream, "%s\n", yytext ); }
this   { fprintf( ostream, "%s\n", yytext ); }
throw    { fprintf( ostream, "%s\n", yytext ); }
try      { fprintf( ostream, "%s\n", yytext ); }
type_info  { fprintf( ostream, "%s\n", yytext ); }
typeid   { fprintf( ostream, "%s\n", yytext ); }
typename { fprintf( ostream, "%s\n", yytext ); }
virtual    { fprintf( ostream, "%s\n", yytext ); }

auto   { fprintf( ostream, "auto\n" );     }
break    { fprintf( ostream, "break\n" );    }
case   { fprintf( ostream, "case\n" );     }
char   { fprintf( ostream, "char\n" );     }
const    { fprintf( ostream, "const\n" );    }
continue { fprintf( ostream, "continue\n" );   }
default    { fprintf( ostream, "default\n" );    }
do     { fprintf( ostream, "do\n" );     }
double   { fprintf( ostream, "double\n" );   }
```

```
else   { fprintf( ostream, "else\n" );     }
enum   { fprintf( ostream, "enum\n" );     }
extern   { fprintf( ostream, "extern\n" );   }
float    { fprintf( ostream, "float\n" );    }
for      { fprintf( ostream, "for\n" );      }
goto   { fprintf( ostream, "goto\n" );     }
if     { fprintf( ostream, "if\n" );     }
int      { fprintf( ostream, "int\n" );      }
long   { fprintf( ostream, "long\n" );     }
register { fprintf( ostream, "register\n" );   }
return   { fprintf( ostream, "return\n" );   }
short    { fprintf( ostream, "short\n" );    }
signed   { fprintf( ostream, "signed\n" );   }
sizeof   { fprintf( ostream, "sizeof\n" );   }
static   { fprintf( ostream, "static\n" );   }
struct   { fprintf( ostream, "struct\n" );   }
switch   { fprintf( ostream, "switch\n" );   }
typedef    { fprintf( ostream, "typedef\n" );    }
union    { fprintf( ostream, "union\n" );    }
unsigned { fprintf( ostream, "unsigned\n" );   }
void   { fprintf( ostream, "void\n" );     }
volatile { fprintf( ostream, "volatile\n" );   }
while    { fprintf( ostream, "while\n" );    }

.      ;

%%

void comments()
{
  register int c;
  for ( ; ; )
  {
    while ( ( c = input() ) != '*' && c != EOF );
    if ( c == '*' )
    {
      while ( ( c = input() ) == '*' );
            if ( c == '/' ) break;    /* found the end */
    }
    if ( c == EOF ) unput( c );
  }
}

int yywrap()
{
  return 1;
}
```

1.2 Keywordsdlg.ccp

```
// KeywordsDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Keywords.h"
#include "KeywordsDlg.h"
#include "DirDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
  CAboutDlg();

// Dialog Data
  //{{AFX_DATA(CAboutDlg)
  enum { IDD = IDD_ABOUTBOX };
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAboutDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
  //}}AFX_VIRTUAL

// Implementation
protected:
  //{{AFX_MSG(CAboutDlg)
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
  //{{AFX_DATA_INIT(CAboutDlg)
  //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAboutDlg)
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
  //{{AFX_MSG_MAP(CAboutDlg)
    // No message handlers
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CKeywordsDlg dialog
```

```
CKeywordsDlg::CKeywordsDlg(CWnd* pParent /*=NULL*/)
  : CResizingDialog(CKeywordsDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CKeywordsDlg)
  mFile = _T("");
  mDirectory = _T("");
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

  SetControlInfo( IDC_TITLE,   RESIZE_HORIZONTAL | ANCHORE_TOP );
  SetControlInfo( IDC_VERSION, RESIZE_HORIZONTAL | ANCHORE_TOP );

  SetControlInfo( IDC_FILE,        RESIZE_HORIZONTAL | ANCHORE_TOP );
  SetControlInfo( IDC_BROWSE_FILE, ANCHORE_RIGHT | ANCHORE_TOP );

  SetControlInfo( IDC_DIRECTORY,        RESIZE_HORIZONTAL | ANCHORE_TOP );
  SetControlInfo( IDC_BROWSE_DIRECTORY, ANCHORE_RIGHT | ANCHORE_TOP );

  SetControlInfo( IDC_GENERATE,  RESIZE_HORIZONTAL | ANCHORE_TOP );
  SetControlInfo( IDOK,          ANCHORE_RIGHT | ANCHORE_TOP );
}

void CKeywordsDlg::DoDataExchange(CDataExchange* pDX)
{
  CResizingDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CKeywordsDlg)
  DDX_Control(pDX, IDC_VERSION, mVersionCtrl);
  DDX_Control(pDX, IDC_TITLE, mTitleCtrl);
  DDX_Text(pDX, IDC_FILE, mFile);
  DDX_Text(pDX, IDC_DIRECTORY, mDirectory);
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CKeywordsDlg, CResizingDialog)
  //{{AFX_MSG_MAP(CKeywordsDlg)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  ON_BN_CLICKED(IDC_GENERATE, OnGenerate)
  ON_BN_CLICKED(IDC_BROWSE_FILE, OnBrowseFile)
  ON_BN_CLICKED(IDC_BROWSE_DIRECTORY, OnBrowseDirectory)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CKeywordsDlg message handlers

BOOL CKeywordsDlg::OnInitDialog()
{
  CResizingDialog::OnInitDialog();

  // Add "About..." menu item to system menu.

  // IDM_ABOUTBOX must be in the system command range.
  ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
  ASSERT(IDM_ABOUTBOX < 0xF000);

  CMenu* pSysMenu = GetSystemMenu(FALSE);
  if (pSysMenu != NULL)
  {
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
```

```
      pSysMenu->AppendMenu(MF_SEPARATOR);
      pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
  }

  // Set the icon for this dialog.  The framework does this automatically
  //  when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);    // Set big icon
  SetIcon(m_hIcon, FALSE);   // Set small icon

  // TODO: Add extra initialization here
  mTitleCtrl.SetFontBold( true );
  mTitleCtrl.SetFontSize( 18 );
  mTitleCtrl.SetFontName( "Arial" );


  return TRUE;  // return TRUE  unless you set the focus to a control
}

void CKeywordsDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_ABOUTBOX)
  {
    CAboutDlg dlgAbout;
    dlgAbout.DoModal();
  }
  else
  {
    CResizingDialog::OnSysCommand(nID, lParam);
  }
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CKeywordsDlg::OnPaint()
{
  if (IsIconic())
  {
    CPaintDC dc(this); // device context for painting

    SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

    // Center icon in client rectangle
    int cxIcon = GetSystemMetrics(SM_CXICON);
    int cyIcon = GetSystemMetrics(SM_CYICON);
    CRect rect;
    GetClientRect(&rect);
    int x = (rect.Width() - cxIcon + 1) / 2;
    int y = (rect.Height() - cyIcon + 1) / 2;

    // Draw the icon
    dc.DrawIcon(x, y, m_hIcon);
  }
  else
  {
    CResizingDialog::OnPaint();
  }
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CKeywordsDlg::OnQueryDragIcon()
{
```

```
  return (HCURSOR) m_hIcon;
}

void CKeywordsDlg::OnBrowseFile()
{
  UpdateData(true);

  CFileDialog dialog( true, NULL, NULL,
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "Source Files (*.c;*.cpp;*.cc;*.cxx)|*.c; *.cpp; *.cc; *.cxx|All Files
(*.*)|*.*||" );

  if ( dialog.DoModal() != IDOK ) return;
  mFile = dialog.GetPathName();

  if ( mDirectory.GetLength() == 0 )
  {
    mDirectory = mFile;
    mDirectory = mDirectory.Left( mFile.ReverseFind( '\\' ) );
  }

  UpdateData(false);
}

void CKeywordsDlg::OnBrowseDirectory()
{
  UpdateData(true);

  CDirDialog dialog;
  dialog.m_strTitle  = "Select Destination Directory:";
  dialog.m_strSelDir = mDirectory;
  if ( dialog.DoBrowse() != IDOK ) return;
  mDirectory = dialog.m_strPath;

  UpdateData(false);
}

extern "C" void yyrestart( FILE* );
extern "C" int  yylex();

extern "C" FILE* istream;
extern "C" FILE* ostream;

/*
FILE* istream = NULL;
FILE* ostream = NULL;
*/

void CKeywordsDlg::OnGenerate()
{
  UpdateData(true);

  if ( mFile.GetLength() == 0 )
  {
    MessageBox(
      "Please specify source file.",
      "Error", MB_OK | MB_ICONERROR );
    return;
  }

  if ( mDirectory.GetLength() == 0 )
  {
    MessageBox(
      "Please specify destination directory.",
      "Error", MB_OK | MB_ICONERROR );
```

```
    return;
  }

  if ( istream != NULL ) fclose( istream );
  if ( ( istream = fopen( mFile, "r" ) ) == NULL )
  {
    MessageBox(
      "Cannot open source file.",
      "Error", MB_OK | MB_ICONERROR );
    return;
  }

  if ( ostream != NULL ) fclose( ostream );
  if ( mDirectory.GetAt( mDirectory.GetLength() - 1 ) == '\\' )
  {
    mDirectory = mDirectory.Left( mDirectory.GetLength() - 1 );
  }
  CString destination = mDirectory;
  destination += mFile.Right( mFile.GetLength() - mFile.ReverseFind( '\\' ) );
  destination += ".kwd";
  if ( ( ostream = fopen( destination, "w" ) ) == NULL )
  {
    MessageBox(
      "Cannot open destination file.",
      "Error", MB_OK | MB_ICONERROR );

    fclose( istream ); istream = NULL;

    return;
  }

  // Parsing file
  rewind( istream );
  yyrestart( istream );
  yylex();

  fclose( istream ); istream = NULL;
  fclose( ostream ); ostream = NULL;

  MessageBox( "Done !", "Information", MB_OK | MB_ICONINFORMATION );
}
```

1.3 Keywords.ccp

```
// Keywords.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "Keywords.h"
#include "KeywordsDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CKeywordsApp

BEGIN_MESSAGE_MAP(CKeywordsApp, CWinApp)
  //{{AFX_MSG_MAP(CKeywordsApp)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //    DO NOT EDIT what you see in these blocks of generated code!
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CKeywordsApp construction

CKeywordsApp::CKeywordsApp()
{
  // TODO: add construction code here,
  // Place all significant initialization in InitInstance
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CKeywordsApp object

CKeywordsApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CKeywordsApp initialization

BOOL CKeywordsApp::InitInstance()
{
  AfxEnableControlContainer();

  // Standard initialization
  // If you are not using these features and wish to reduce the size
  //  of your final executable, you should remove from the following
  //  the specific initialization routines you do not need.

#ifdef _AFXDLL
  Enable3dControls();      // Call this when using MFC in a shared DLL
#else
  Enable3dControlsStatic();// Call this when linking to MFC statically
#endif

  CKeywordsDlg dlg;
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();
  if (nResponse == IDOK)
  {
    // TODO: Place code here to handle when the dialog is
```

```
    //  dismissed with OK
  }
  else if (nResponse == IDCANCEL)
  {
    // TODO: Place code here to handle when the dialog is
    //  dismissed with Cancel
  }
  // Since the dialog has been closed, return FALSE so that we exit the
  //  application, rather than start the application's message pump.
  return FALSE;
}
```

2. Lcr

```
clear
%compil é o programa aberto para leitura do programa cujo nome está em argumento do fopen
%linha é a variável para onde é lida cada linha do ficheiro
%numpal é o numero de palavras chavedo linguagem em análise que se encontram arquivadas
%na matriz palavras chave
%cp é o contador de ocorrência de palavras chave
%numcat é ocomprimento de cada palavra chave
%codpalres é a matriz onde são guardados os códigos atribuídos ás palavras chave
compil=fopen('total1.txt','r');
linha=fgetl(compil);
%inicialização das variáveis
numpal=32;
cp=zeros(1,numpal);
palavra chave=['auto    ';'break  ';'case    ';'char    ';'const   ';
  'continue';'default ';'do      ';'double  ';'else    ';'enum    ';
  'extern  ';'float   ';'for     ';'goto    ';'if      ';'int     ';
  'long    ';'register';'return  ';'short   ';'signed  ';'sizeof  ';
  'static  ';'struct  ';'switch  ';'typedef ';'union   ';'unsigned';
  'void    ';'volatile';'while   '];
numcat=[4 5 4 4 5 8 7 2 6 4 4 6 5 3 4 2 3 4 8 6 5 6 6 6 6 6 7 5 8 4 8 5];
codpalres=[-16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15];
k=1;

%identificação da palavra chave guardada em linha
while (linha~=-1)
  i=1;
  while (i<(numpal+1))&(strcmp(linha,palavra-chave(i,1:numcat(i)))==0)
    i=i+1;
  end
%indicação se em linha está uma palavra que não é palavra chave
  if i == numpal+1
    clc
    fprintf('%s: esta palavra não está no código',linha)
    pause
   else
    %contagem da ocorrência de palavra chave
       cp(i)=cp(i)+1;
    %criação da matriz fic com os codigos das palavras
    fic(k)=codpalres(i);
   k=k+1;
  end
    linha=fgetl(compil);
end

%calculo das várias posições do passeio aleatório
p=1;
posi(1)=fic(1);
while(p~=k-1)
  posi(p+1)=posi(p)+fic(p+1);
  p=p+1;
end
%calculo da long range correlation
%o comprimento do ficheiro é k-1
%a variável dist fica com as sucessivas distâncias
%a variavel dist2 fica com os quadrados da distancia
%o F(l)é calculado depois de cada ciclo e só com os valores das distancias correspondentes a esse ciclo
l=1;
n=1;
while(l~=k-2)
  contd=0;
  while(n~=k-l)
    dist(n)=posi(n+l)-posi(n);
```

```
    dist2(n)=dist(n)^2;
    n=n+1;
  end
  F(l)=mean(dist2(1:((k-1)-l)))-(mean(dist(1:((k-1)-l)))^2);
  l=l+1;
  n=1;
end
%impressão do gráfico do alfa
b=0;
s=1;
while (s~=k-2)
  f1(s)=sqrt(F(s));
  b(s)=s;
  s=s+1;
end
loglog(f1)
%calculo do alfa
%r é o indice do ponto máximo da função
%só uso os pontos de 10 a r para o cálculo do declive
[log(s),r]=max(log(f1));
%obtenho os logaritmos
f2=lo10(f1);
f3=lo10(b);
%restrinjo a string até um terço do máximo e depois do 50 máximo
t=round(r/5)
f4(1:(t-49))=f2(50:t);
f5(1:(t-49))=f3(50:t);
r
%obtenho os coeficientes da recta o primeiro será o expoente da power law
R=polyfit(f5,f4,1)
fclose(compil);
```

4. Ale

```
clear
%compil é o programa aberto para leitura do programa cujo nome está em argumento do fopen
%linha é a variável para onde é lida cada linha do ficheiro
%numpal é o numero de palavras chavedo linguagem em análise que se encontram arquivadas
%na matriz palavras chave
%cp é o contador de ocorrência de palavras chave
%numcat é ocomprimento de cada palavra chave
%codpalres é a matriz onde são guardados os códigos atribuídos ás palavras chave
compil=fopen('total.txt','r');
linha=fgetl(compil);
%inicialização das variáveis
numpal=32;
cp=zeros(1,numpal);
palavrachave=['auto ';'break ';'case ';'char ';'const '; 'continue';'default ';'do ';'double ';'else ';'enum '; 'extern ';'float ';'for ';'goto ';'if ';'int ';'long ';'register';'return ';'short ';'signed ';'sizeof '; 'static ';'struct ';'switch ';'typedef ';'union ';'unsigned'; 'void ';'volatile';'while '];
numcat=[4 5 4 4 5 8 7 2 6 4 4 6 5 3 4 2 3 4 8 6 5 6 6 6 6 6 7 5 8 4 8 5];
codpalres=[-16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15];
k=1;
%identificação da palavra reservada guardada em linha
while (linha~=-1)
  i=1;
  while (i<(numpal+1))&(strcmp(linha,palavrareserv(i,1:numcat(i)))==0)
    i=i+1;
  end
%indicação se em linha está uma palavra que não é palavra reservada
  if i == numpal+1
    clc
    fprintf('%s: esta palavra não está no código',linha)
    pause
   else
    %contagem da ocorrência de palavra reservada
       cp(i)=cp(i)+1;
    %criação da matriz fic com os codigos das palavras
    fic(k)=codpalres(i);
   k=k+1;
  end
    linha=fgetl(compil);
end
%o vector fic(k)contem a sequencia numérica corespondente ás palavras
%chave de um programa.
%o seu tamanho é k-1
%gero um vector com permutação aleatórias dos k-1 indices

%repito o processo 3 vezes.
u=1;
while u<4
indice=randperm(k-1);
u=u+1;
end
%com esses novos indices reconstru-o
m=1;
while m<k
  ficale(m)=fic(indice(m));
  m=m+1;

end

%calculo das várias posições do passeio aleatório
p=1;
posi(1)=ficale(1);
while(p~=k-1)
```

```
  posi(p+1)=posi(p)+ficale(p+1);
  p=p+1;
end
%calculo da long range correlation
%o comprimento do ficheiro é k-1
%a variável dist fica com as sucessivas distâncias
%a variavel dist2 fica com os quadrados da distancia
%o F(l)é calculado depois de cada ciclo e só com os valores das distancias correspondentes a esse ciclo
l=1;
n=1;
while(l~=k-2)
  contd=0;
  while(n~=k-l)
    dist(n)=posi(n+l)-posi(n);
    dist2(n)=dist(n)^2;
    n=n+1;
  end
  F(l)=mean(dist2(1:((k-1)-l)))-(mean(dist(1:((k-1)-l)))^2);
  l=l+1;
  n=1;
end
%impressão do gráfico do alfa
b=0;
s=1;
while (s~=k-2)
  f1(s)=sqrt(F(s));
  b(s)=s;
  s=s+1;
end
loglog(f1)
%calculo do alfa
%r é o indice do ponto máximo da função
%só uso os pontos de 10 a r para o cálculo do declive
[log(s),r]=max(log(f1));
%obtenho os logaritmos
f2=lo10(f1);
f3=lo10(b);
%restrinjo a string até um terço do máximo e depois do 50 máximo
t=round(r/5)
f4(1:(t-49))=f2(50:t);
f5(1:(t-49))=f3(50:t);
r
%obtenho os coeficientes da recta o primeiro será o expoente da power law
R=polyfit(f5,f4,1)
fclose(compil);
```

5. Gramática do gerador

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELIPSIS RANGE

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%{ /* RGC 2001.4.6 */
unsigned snumb,fnumb;
%}

%start file
%%

primary_expr
    : identifier
    | CONSTANT
    | STRING_LITERAL
    | '(' expr ')'
    ;

postfix_expr
    : primary_expr
    | postfix_expr '[' expr ']'
    | postfix_expr '(' ')'
    | postfix_expr '(' argument_expr_list ')'
    | postfix_expr '.' identifier
    | postfix_expr PTR_OP identifier
    | postfix_expr INC_OP
    | postfix_expr DEC_OP
    ;

argument_expr_list
    : assignment_expr  { /*RGC 2001.4.6*/ snumb; }
    | argument_expr_list ',' assignment_expr  { /*RGC 2001.4.6*/ snumb; }
    ;

unary_expr
    : postfix_expr
    | INC_OP unary_expr
    | DEC_OP unary_expr
    | unary_operator cast_expr
    | SIZEOF unary_expr
    | SIZEOF '(' type_name ')'
    ;

unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;
```

```
cast_expr
    : unary_expr
    | '(' type_name ')' cast_expr
    ;

multiplicative_expr
    : cast_expr
    | multiplicative_expr '*' cast_expr
    | multiplicative_expr '/' cast_expr
    | multiplicative_expr '%' cast_expr
    ;

additive_expr
    : multiplicative_expr
    | additive_expr " multiplicative_expr
    | additive_expr '-' multiplicative_expr
    ;

shift_expr
    : additive_expr
    | shift_expr LEFT_OP additive_expr
    | shift_expr RIGHT_OP additive_expr
    ;

relational_expr
    : shift_expr
    | relational_expr '<' shift_expr
    | relational_expr '>' shift_expr
    | relational_expr LE_OP shift_expr
    | relational_expr GE_OP shift_expr
    ;

equality_expr
    : relational_expr
    | equality_expr EQ_OP relational_expr
    | equality_expr NE_OP relational_expr
    ;

and_expr
    : equality_expr
    | and_expr '&' equality_expr
    ;

exclusive_or_expr
    : and_expr
    | exclusive_or_expr '^' and_expr
    ;

inclusive_or_expr
    : exclusive_or_expr
    | inclusive_or_expr '|' exclusive_or_expr
    ;

logical_and_expr
    : inclusive_or_expr
    | logical_and_expr AND_OP inclusive_or_expr
    ;

logical_or_expr
    : logical_and_expr
    | logical_or_expr OR_OP logical_and_expr
    ;

conditional_expr
```

```
    : logical_or_expr
    | logical_or_expr '?' logical_or_expr ':' conditional_expr
    ;

assignment_expr
    : conditional_expr
    | unary_expr assignment_operator assignment_expr
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expr
    : assignment_expr
    | expr ',' assignment_expr
    ;

constant_expr
    : conditional_expr
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : CHAR
```

```
    | SHORT
    | INT
    | LONG
    | SIGNED
    | UNSIGNED
    | FLOAT
    | DOUBLE
    | CONST
    | VOLATILE
    | VOID
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union identifier '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union identifier
    ;

struct_or_union
    : STRUCT
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : type_specifier_list struct_declarator_list ';'
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expr
    | declarator ':' constant_expr
    ;

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM identifier '{' enumerator_list '}'
    | ENUM identifier
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    ;

enumerator
    : identifier
    | identifier '=' constant_expr
    ;

declarator
    : declarator2
```

```
    | pointer declarator2
    ;

declarator2
    : identifier
    | '(' declarator ')'
    | declarator2 '[' ']'
    | declarator2 '[' constant_expr ']'
    | declarator2 '(' ')'
    | declarator2 '(' parameter_type_list ')'
    | declarator2 '(' parameter_identifier_list ')'
    ;

pointer
    : '*'
    | '*' type_specifier_list
    | '*' pointer
    | '*' type_specifier_list pointer
    ;

type_specifier_list
    : type_specifier
    | type_specifier_list type_specifier
    ;

parameter_identifier_list
    : identifier_list
    | identifier_list ',' ELIPSIS
    ;

identifier_list
    : identifier
    | identifier_list ',' identifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list ',' ELIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : type_specifier_list declarator
    | type_name
    ;

type_name
    : type_specifier_list
    | type_specifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | abstract_declarator2
    | pointer abstract_declarator2
    ;

abstract_declarator2
    : '(' abstract_declarator ')'
    | '[' ']'
```

```
    | '[' constant_expr ']'
    | abstract_declarator2 '[' ']'
    | abstract_declarator2 '[' constant_expr ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | abstract_declarator2 '(' ')'
    | abstract_declarator2 '(' parameter_type_list ')'
    ;

initializer
    : assignment_expr
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | initializer_list ',' initializer
    ;

statement
    : labeled_statement    { /*RGC 2001.4.6*/ snumb; }
    | compound_statement   { /*RGC 2001.4.6*/ snumb; }
    | expression_statement { /*RGC 2001.4.6*/ snumb; }
    | selection_statement  { /*RGC 2001.4.6*/ snumb; }
    | iteration_statement  { /*RGC 2001.4.6*/ snumb; }
    | jump_statement       { /*RGC 2001.4.6*/ snumb; }
    ;

labeled_statement
    : identifier ':' statement
    | CASE constant_expr ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : ';'
    | expr ';'
    ;

selection_statement
    : IF '(' expr ')' statement
    | IF '(' expr ')' statement ELSE statement
    | SWITCH '(' expr ')' statement
    ;

iteration_statement
```

```
    : WHILE '(' expr ')' statement
    | DO statement WHILE '(' expr ')' ';'
    | FOR '(' ';' ';' ')' statement
    | FOR '(' ';' ';' expr ')' statement
    | FOR '(' ';' expr ';' ')' statement
    | FOR '(' ';' expr ';' expr ')' statement
    | FOR '(' expr ';' ';' ')' statement
    | FOR '(' expr ';' ';' expr ')' statement
    | FOR '(' expr ';' expr ';' ')' statement
    | FOR '(' expr ';' expr ';' expr ')' statement
    ;

jump_statement
    : GOTO identifier ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expr ';'
    ;

file
    : external_definition
    | file external_definition
    ;

external_definition
    : function_definition
    | declaration
    ;

function_definition
    : declarator function_body { /* RGC 2001.4.6 */ fnumb; }
    | declaration_specifiers declarator function_body { /* RGC 2001.4.6 */ fnumb; }
    ;

function_body
    : compound_statement
    | declaration_list compound_statement
    ;

identifier
    : IDENTIFIER
    ;
%%

#include <stdio.h>

extern char yytext[];
extern int column;

yyerror(s)
char *s;
{
    fflush(stdout);
    printf("\n%*s\n%*s\n", column, "^", column, s);
}

int main()
{
    int yyparse();

   /*RGC 2001.4.6*/ yyparse();
                printf("***Instrucoes=%d\n",snumb);
                printf("***Funcoes=%d\n",fnumb);
```

```
    /* RGC 2001.4.6 return(yyparse()); */
}
```

6. Gerador1

```
#include <stdio.h>
#include <stdlib.h>

FILE *fp;       /* descritor ficheiro saida */
int fnumb;      /* numero de funcoes que falta gerar */
int lnumb;      /* numero de linhas que falta gerar */

void expr(), expr1(), expr2();

void out(char *s) {
  fprintf(stderr,"%s\n",s); exit(EXIT_FAILURE); }

int layer(int size) {
/* devolve numero aleatorio entre 0 e size-1 */
  int trand = rand();
  int layer = (RAND_MAX+1) / size;
  int result = 0;

  while (trand>layer) {trand=trand-layer; result++;}
  return result; }

void id() {
/* imprime no ficheiro um identificador */
  fprintf(fp,"i"); }

void tip() {
/* imprime no ficheiro um tipo de dados válido */
  switch(layer(15)) {
    case 0: fprintf(fp,"char "); break;
    case 1: fprintf(fp,"short "); break;
    case 2: fprintf(fp,"int "); break;
    case 3: fprintf(fp,"long "); break;
    case 4: fprintf(fp,"signed "); break;
    case 5: fprintf(fp,"unsigned "); break;
    case 6: fprintf(fp,"float "); break;
    case 7: fprintf(fp,"double "); break;
    case 8: fprintf(fp,"const "); break;
    case 9: fprintf(fp,"volatile "); break;
    case 10: fprintf(fp,"void "); break;
    case 11:
      switch(layer(2)) {
        case 0: fprintf(fp,"struct "); break;
        case 1: fprintf(fp,"union "); break;
        default: out("Erro typ-1"); }
      switch(layer(2)) {
        case 0: id(); fprintf(fp," "); break;
        case 1: break;
        default: out("Erro tip-2"); }
      switch(layer(2)) {
        case 0: fprintf(fp,"{");tip(); fprintf(fp,"}"); break;
        case 1: break;
        default: out("Erro tip-3"); }
      break;
    case 12: fprintf(fp,"enum ");
      switch(layer(2)) {
        case 0: id(); fprintf(fp," "); break;
        case 1: break;
        default: out("Erro tip-4"); }
      switch(layer(2)) {
        case 0: fprintf(fp,"{");id(); fprintf(fp,"}"); break;
```

```
          case 1: break;
          default: out("Erro tip-5"); }
        break;
     case 13: id(); fprintf(fp," "); break;
     case 14: tip(); fprintf(fp," *"); break;
     default: out("Erro tip-6"); }
}

void declaration(int locs) {
/* imprime no ficheiro uma sequencia de declaracoes */
int trand;

  /* identifica numero de declaracoes */
  int dnumb = locs/10;
  switch(layer(5)) {
    case 0: dnumb=dnumb/3; break;
    case 1: dnumb=dnumb/2; break;
    case 2: break;
    case 3: dnumb=dnumb*2; break;
    case 4: dnumb=dnumb*3; break;
    default: out("Erro locs-1\n"); }

  for(;dnumb>0;dnumb--) {
    lnumb--;
    if (rand()<=RAND_MAX*.5) { /*storage*/
       switch(layer(5)) {
          case 0: fprintf(fp,"typedef "); break;
          case 1: fprintf(fp,"extern "); break;
          case 2: fprintf(fp,"static "); break;
          case 3: fprintf(fp,"auto "); break;
          case 4: fprintf(fp,"register "); break;
          default: out("Erro locs-2"); } }
    else tip(); /*type*/
    id(); fprintf(fp,";\n"); }
  fprintf(fp,"\n"); }

void expr() {
/* imprime no ficheiro uma expressao */
   switch(layer(9)) {
      case 0:
      case 1: fprintf(fp,"i"); break;
      case 2:
      case 3: fprintf(fp,"1"); break;
      case 4: expr1(); break;
      case 5:
      case 6: expr2(); break;
      case 7: fprintf(fp,"("); expr(); fprintf(fp,")"); break;
      case 8: fprintf(fp,"("); tip(); fprintf(fp,")"); expr(); break;
      default: out("Erro expr"); }
}
void expr1() {
/* imprime no ficheiro uma expressao unnaria */
   switch(layer(8)) {
      case 0: fprintf(fp,"-"); expr(); break;
      case 1: fprintf(fp,"!"); expr(); break;
      case 2: fprintf(fp,"*"); expr(); break;
      case 3: fprintf(fp,"~"); expr(); break;
      case 4: expr(); fprintf(fp,"++"); break;
      case 5: expr(); fprintf(fp,"--"); break;
      case 6: fprintf(fp,"&"); id(); break;
      case 7: fprintf(fp,"*"); id(); break;
      default: out("Erro expr1"); exit(1); }
}
void expr2() {
```

```
/* imprime no ficheiro uma expressao binaria */
  switch(layer(13)) {
    case 0: expr(); fprintf(fp," + "); expr(); break;
    case 1: expr(); fprintf(fp," - "); expr(); break;
    case 2: expr(); fprintf(fp," * "); expr(); break;
    case 3: expr(); fprintf(fp," / "); expr(); break;
    case 4: expr(); fprintf(fp," % "); expr(); break;
    case 5: expr(); fprintf(fp," %% "); expr(); break;
    case 6: expr(); fprintf(fp," // "); expr(); break;
    case 7: expr(); fprintf(fp," == "); expr(); break;
    case 8: expr(); fprintf(fp," > "); expr(); break;
    case 9: expr(); fprintf(fp," >= "); expr(); break;
    case 10: expr(); fprintf(fp," < "); expr(); break;
    case 11: expr(); fprintf(fp," <= "); expr(); break;
    case 12: expr(); fprintf(fp," != "); expr(); break;
    default: out("Erro expr2"); } }

void attrib() {
/* imprime no ficheiro uma atribuicao: 1 em 6 casos e' multipla */
  id(); fprintf(fp,"="); expr();
  switch(layer(6)) {
     case 0:
     case 1:
     case 2:
     case 3:
     case 4: break;
     case 5: fprintf(fp,","); attrib(); break;
     default: out("Erro attrib\n"); } }

void stmt() {
/* imprime no ficheiro uma instrucao */

   lnumb--;
   switch(layer(6)) {
     case 0: /* labeled statement */
       switch(layer(3)) {
         case 0: fprintf(fp,"   "); id(); fprintf(fp,":"); stmt(); lnumb--; break;
         case 1: fprintf(fp,"   case 0:"); stmt(); lnumb--; break;
         case 2: fprintf(fp,"   default:"); stmt(); lnumb--; break;
         default: out("Erro stmt-0\n"); }
       break;
     case 1: /* compound statement */
       fprintf(fp,"   {");
       switch(layer(2)) {
         case 0: break;
         case 1: declaration(2); break;
         default: out("Erro stmt-1-1"); }
       switch(layer(2)) {
         case 0: break;
         case 1: stmt(10); break;
         default: out("Erro stmt-1-2"); }
       fprintf(fp,"   }");
       break;
     case 2: /* expression statement */
       switch(layer(6)) {
         case 0:
         case 1:
         case 2:
         case 3:
         case 4: fprintf(fp,"   "); attrib(); fprintf(fp,";\n"); break;
         case 5: fprintf(fp,"   ;"); break;
         default: out("Erro stmt-2"); }
       break;
     case 3: /* selection statemenet */
```

```
        switch(layer(2)) {
          case 0: fprintf(fp,"   if("); expr(); fprintf(fp,")");
                  stmt(); fprintf(fp,"\n");
                  switch(layer(2)) {
                    case 0: break;
                    case 1: fprintf(fp,"      else ");
                            stmt(); lnumb--; fprintf(fp,"\n"); break;
                    default: out("Erro stmt-3-1"); }
                  break;
          case 1: fprintf(fp,"   switch("); expr(); fprintf(fp,") {");
                  stmt(); lnumb--; fprintf(fp,"}\n"); break;
          default: out("Erro stmt-3-2"); }
        break;
      case 4: /* iteration statement */
        switch(layer(3)) {
          case 0: fprintf(fp,"   while("); expr(); fprintf(fp,")"); stmt(); break;
          case 1: fprintf(fp,"   do "); stmt(); fprintf(fp," while("); expr(); fprintf(fp,")\n"); break;
          case 2: fprintf(fp,"   for(");
            switch(layer(2)) {
              case 0: fprintf(fp,";"); break;
              case 1: expr(); fprintf(fp,";"); break;
              default: out("Erro stmt-4-1"); }
            switch(layer(2)) {
              case 0: fprintf(fp,";"); break;
              case 1: expr(); fprintf(fp,";"); break;
              default: out("Erro stmt-4-2"); }
            switch(layer(2)) {
              case 0: fprintf(fp,")"); break;
              case 1: attrib(); fprintf(fp,")"); break;
              default: out("Erro stmt-4-3"); }
            stmt(); break;
          default: out("Erro stmt-3-2"); }
        break;
      case 5: /* jump statement */
        switch(layer(4)) {
          case 0: fprintf(fp,"   goto "); id(); fprintf(fp,";\n"); break;
          case 1: fprintf(fp,"   continue;\n"); break;
          case 2: fprintf(fp,"   break;\n"); break;
          case 3: fprintf(fp,"   return ");
            switch(layer(2)) {
              case 0: break;
              case 1: expr(); break;
              default: out("Erro stmt-5-1"); }
            fprintf(fp,";\n"); break;
          default: out("Erro stmt-4"); }
        break;
      default: out("Erro stmt"); }}

void func(int locs) {
/* imprime no ficheiro uma funcao */
int startn=lnumb;

   /* cabecalho */
   tip(); fprintf(fp," nome() {\n");
   lnumb--;

    /* declaracoes locais */
   declaration(locs);

    /* instrucoes */
   while( (startn-lnumb)<locs ) stmt();
   fprintf(fp,"}\n\n"); }

main(int argc, char *argv[]) {
```

```
int i;

if (argc!=4 )    out("gerador ficheiro nlinhas nfunctions");

/* abre ficheiro saida */
fp=fopen(argv[1],"w");
if (fp==NULL)    out("Erro na escrita do ficheiro");
/* obtem numero de linhas a gerar */
lnumb=atoi(argv[2]);
srand(lnumb % RAND_MAX);
/* obtem numero de funcoes a gerar */
fnumb=atoi(argv[3]);
if (fnumb<1)     out("Deve haver, pelo menos, uma funcao\n");

/* gera declaracoes globais */
declaration(lnumb/atoi(argv[3]));

/* gera funcoes */
while (lnumb>0) {
  func(atoi(argv[2])/fnumb); }

fclose(fp);
 }
```

7. Gerador2

```
#include <stdio.h>
#include <stdlib.h>
#define TRUE  1
#define FALSE 0
#define DEBUG 1

FILE *fp,       /* descritor ficheiro saida */
     *ins;      /* ficheiro com contador das palavras chave */
int fnumb;      /* numero de funcoes que falta gerar */

int TOK_auto,
    TOK_break,
    TOK_case,
    TOK_char,
    TOK_const,
    TOK_continue,
    TOK_default,
    TOK_do,
    TOK_double,
    TOK_else,
    TOK_enum,
    TOK_extern,
    TOK_float,
    TOK_for,
    TOK_goto,
    TOK_if,
    TOK_int,
    TOK_long,
    TOK_register,
    TOK_return,
    TOK_short,
    TOK_signed,
    TOK_sizeof,
    TOK_static,
    TOK_struct,
    TOK_switch,
    TOK_typedef,
    TOK_union,
    TOK_unsigned,
    TOK_void,
    TOK_volatile,
    TOK_while,
    NUMBTOK_decl,    /* numero de tokens de declaracoes */
    NUMBTOK_stmt;    /* numero de tokens de instrucoes */

void expr(), expr1(), expr2(), declaration();

void out(char *s) {
   fprintf(stderr,"%s\n",s); exit(EXIT_FAILURE); }

int layer(int size) {
/* devolve numero aleatorio entre 0 e size-1 */
   int trand = rand();
   int layer = RAND_MAX/size+1;
   int result = 0;

   while (trand>layer) {trand=trand-layer; result++;}
   return result; }

void id() {
/* imprime no ficheiro um identificador */
```

```
  fprintf(fp,"i"); }

void tip(int rest) {
/* imprime no ficheiro um tipo de dados valido */
int aux;
int level=layer(15);

    switch(level) {
    case 0: if(TOK_volatile>0) {
            TOK_volatile--;fprintf(fp,"volatile ");return; }
    case 1: if(TOK_char>0) {
            TOK_char--;fprintf(fp,"char ");return; }
    case 2: if(TOK_short>0) {
            TOK_short--;fprintf(fp,"short ");return; }
    case 3: if(TOK_int>0) {
            TOK_int--;fprintf(fp,"int ");return; }
    case 4: if(TOK_long>0) {
            TOK_long--;fprintf(fp,"long ");return; }
    case 5: if(TOK_signed>0) {
            TOK_signed--;fprintf(fp,"signed ");return; }
    case 6: if(TOK_unsigned>0) {
            TOK_unsigned--;fprintf(fp,"unsigned ");return; }
    case 7: if(TOK_float>0) {
            TOK_float--;fprintf(fp,"float ");return; }
    case 8: if(TOK_double>0) {
            TOK_double--;fprintf(fp,"double ");return; }
    case 9: if(TOK_const>0) {
            TOK_const--;fprintf(fp,"const ");return; }
    case 10: if(TOK_void>0) {
            TOK_void--;fprintf(fp,"void ");return; }
    case 11:
      switch(layer(2)) {
        case 0: if(TOK_struct>0) {
            TOK_struct--;fprintf(fp,"struct ");break; }
              else break;
        case 1: if(TOK_union>0) {
            TOK_union--;fprintf(fp,"union ");break; }
              else break;
        default: out("Erro typ-1"); }
        if(TOK_struct==00 && TOK_union==0) return;
      switch(layer(2)) {
        case 0: id();fprintf(fp," ");break;
        case 1: break;
        default: out("Erro tip-2"); }
      switch(layer(6)) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4: fprintf(fp,"{");
                declaration(rest,layer(10));
                fprintf(fp,"}\n"); break;
        case 5: break;
        default: out("Erro tip-3"); }
      break;
    case 12: if(TOK_enum>0) {
              TOK_enum--;fprintf(fp,"enum ");
              switch(layer(2)) {
                case 0: id(); fprintf(fp," "); break;
                case 1: break;
                default: out("Erro tip-4"); }
              switch(layer(2)) {
                case 0: fprintf(fp,"{");id();fprintf(fp,"}");break;
                case 1: break;
```

```
                default: out("Erro tip-5"); }
              return; }
     case 13: fprintf(fp,"nametype "); break;
     case 14: tip(rest); fprintf(fp," *"); break;
     default: out("Erro tip-6"); }
}

int toks() {
 /* determina o numero de palavras chave de declaracoes que falta gerar */
 return TOK_auto+
    TOK_char+ TOK_const+
    TOK_double+
    TOK_enum+ TOK_extern+
    TOK_float+
    TOK_int+
    TOK_long+
    TOK_register+
    TOK_short+ TOK_signed+ TOK_static+ TOK_struct+
    TOK_typedef+
    TOK_union+ TOK_unsigned+
    TOK_void+ TOK_volatile; }

void basic_decl(int rest) {
  if (rand()<=RAND_MAX*.2) { /*storage*/
    switch(layer(5)) {
      case 0: if(TOK_auto>0) {
                TOK_auto--;fprintf(fp,"auto ");break; }
      case 1: if(TOK_register>0) {
                TOK_register--;fprintf(fp,"register ");break; }
      case 2: if(TOK_typedef>0) {
                TOK_typedef--;fprintf(fp,"typedef ");break; }
      case 3: if(TOK_extern>0) {
                TOK_extern--;fprintf(fp,"extern ");break; }
      case 4: if(TOK_static>0) {
                TOK_static--;fprintf(fp,"static ");break; }
              else break;
      default: out("Erro locs-2\n"); } }
   else tip(rest); /*type*/

   fprintf(fp,"name;\n"); }

void declaration(int rest, int locs) {
/* imprime no ficheiro uma sequencia de locs declaracoes */

 /* identifica numero de declaracoes */
 int dnumb = locs;

 switch(layer(3)) {
   case 0: dnumb=dnumb/2; break;
   case 1: break;
   case 2: dnumb=dnumb*2; break;
   default: out("Erro locs-1\n"); }

 for(;dnumb>0;dnumb--) basic_decl(rest);

   /* na ultima funcao, despeja o resto de declaracoes */
   if (rest==1) {
     while(toks()>0) basic_decl(rest); }

 fprintf(fp,"\n"); }

void parameters(int rest, int decls) {
/* imprime lista de parametros */
```

```
  if (decls==0) return;
  while (decls>1) {
     tip(rest); fprintf(fp," par,"); decls--;
  }
  tip(rest); fprintf(fp," par");
}

void expr(int rest) {
/* imprime no ficheiro uma expressao */

   switch(layer(10)) {
      case 0:
      case 1: fprintf(fp,"i"); break;
      case 2:
      case 3: fprintf(fp,"1"); break;
      case 4: expr1(); break;
      case 5:
      case 6: expr2(rest); break;
      case 7: switch(layer(2)) {
         case 0: if(TOK_sizeof>0) {
            TOK_sizeof--;fprintf(fp,"sizeof("); break; }
         case 1: fprintf(fp,"func("); break;
         default: out("Erro expr-funcao"); }
         expr(rest);fprintf(fp,")"); break;
      case 8: fprintf(fp,"("); expr(rest); fprintf(fp,")"); break;
      case 9: fprintf(fp,"("); tip(rest); fprintf(fp,")"); expr(rest); break;
      default: out("Erro expr"); }
}
void expr1(int rest) {
/* imprime no ficheiro uma expressao unaria */
   switch(layer(8)) {
      case 0: fprintf(fp,"-"); expr(rest); break;
      case 1: fprintf(fp,"!"); expr(rest); break;
      case 2: fprintf(fp,"*"); expr(rest); break;
      case 3: fprintf(fp,"~"); expr(rest); break;
      case 4: expr(rest); fprintf(fp,"++"); break;
      case 5: expr(rest); fprintf(fp,"--"); break;
      case 6: fprintf(fp,"&"); id(); break;
      case 7: fprintf(fp,"*"); id(); break;
      default: out("Erro expr1"); exit(1); }
}
void expr2(int rest) {
/* imprime no ficheiro uma expressao binaria */
   expr(rest);
   switch(layer(13)) {
      case 0: fprintf(fp," + "); break;
      case 1: fprintf(fp," - "); break;
      case 2: fprintf(fp," * "); break;
      case 3: fprintf(fp," / "); break;
      case 4: fprintf(fp," % "); break;
      case 5: fprintf(fp," %% "); break;
      case 6: fprintf(fp," // "); break;
      case 7: fprintf(fp," == "); break;
      case 8: fprintf(fp," > "); break;
      case 9: fprintf(fp," >= "); break;
      case 10: fprintf(fp," < "); break;
      case 11: fprintf(fp," <= "); break;
      case 12: fprintf(fp," != "); break;
      default: out("Erro expr2"); }
   expr(rest); }

void attrib(int rest) {
/* imprime no ficheiro uma atribuicao: 1 em 6 casos e' multipla */
   id(); fprintf(fp,"="); expr(rest);
```

```
  switch(layer(6)) {
     case 0:
     case 1:
     case 2:
     case 3:
     case 4: break;
     case 5: fprintf(fp,","); attrib(rest); break;
     default: out("Erro attrib\n"); } }

void stmt(int rest) {
/* imprime no ficheiro uma instrucao */
int i;

  switch(layer(6)) {
    case 0: /* labeled statement */
      switch(layer(3)) {
        case 0: fprintf(fp,"   "); id(); fprintf(fp,":");
              stmt(rest); break;
        case 1: if(TOK_case>0) {
                 TOK_case--; fprintf(fp,"   case 0:");
                 stmt(rest); break; }
        case 2: if(TOK_default>0) {
                 TOK_default--; fprintf(fp,"   default:");
                 stmt(rest); break; }
              else break;
        default: out("Erro stmt-0"); }
      break;
    case 1: /* compound statement */
      fprintf(fp,"   {");
      switch(layer(2)) {
        case 0: break;
        case 1: declaration(rest,2); break;
        default: out("Erro stmt-1-1"); }
      switch(layer(2)) {
        case 0: break;
        case 1: for(i=0;i<layer(20);i++) stmt(rest); break;
        default: out("Erro stmt-1-2"); }
      fprintf(fp,"   }");
      break;
    case 2: /* expression statement */
      switch(layer(6)) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4: fprintf(fp,"   "); attrib(rest);
              fprintf(fp,";\n"); break;
        case 5: fprintf(fp,"   ;"); break;
        default: out("Erro stmt-2"); }
      break;
    case 3: /* selection statemenet */
      switch(layer(2)) {
        case 0: if(TOK_switch>0) {
                  TOK_switch--;
                  fprintf(fp,"   switch("); expr(rest);
                  fprintf(fp,") {");
                  stmt(rest); fprintf(fp,"}\n"); break; }
        case 1: if(TOK_if>0) {
                  TOK_if--;
                  fprintf(fp,"   if("); expr(rest);
                  fprintf(fp,")");
                  stmt(rest); fprintf(fp,"\n");
                  switch(layer(2)) {
                    case 0: break;
```

```
                    case 1: if(TOK_else>0) {
                              TOK_else--;
                              fprintf(fp,"      else ");
                              stmt(rest); fprintf(fp,"\n");
                              break; }
                            else break;
                    default: out("Erro stmt-3-1"); } }
                  break;
          default: out("Erro stmt-3-2"); }
        break;
      case 4: /* iteration statement */
        switch(layer(3)) {
          case 0: if(TOK_while>0 && (TOK_while>TOK_do)) {
if(DEBUG) printf(" 0:TOK_while==%d,TOK_do==%d\n",TOK_while,TOK_do);
                  TOK_while--;
                  fprintf(fp,"   while("); expr(rest);
                  fprintf(fp,")"); stmt(rest); break; }
          case 1: if(TOK_do>0) {
if(DEBUG) printf(" 1:TOK_while==%d,TOK_do==%d\n",TOK_while,TOK_do);
                  TOK_do--; TOK_while--;
                  fprintf(fp,"   do "); stmt(rest);
                  fprintf(fp," while(");
                  expr(rest); fprintf(fp,")\n"); break; }
          case 2: if(TOK_for>0) {
                    TOK_for--; fprintf(fp,"   for(");
            switch(layer(2)) {
              case 0: fprintf(fp,";"); break;
              case 1: expr(rest); fprintf(fp,";"); break;
              default: out("Erro stmt-4-1"); }
            switch(layer(2)) {
              case 0: fprintf(fp,";"); break;
              case 1: expr(rest); fprintf(fp,";"); break;
              default: out("Erro stmt-4-2"); }
            switch(layer(2)) {
              case 0: fprintf(fp,")"); break;
              case 1: attrib(rest); fprintf(fp,")"); break;
              default: out("Erro stmt-4-3"); }
            stmt(rest); break; }
            else break;
          default: out("Erro stmt-3-2"); }
        break;
      case 5: /* jump statement */
        switch(layer(4)) {
          case 0: if(TOK_goto>0) {
                    TOK_goto--;
                    fprintf(fp,"   goto "); id();
                    fprintf(fp,";\n"); break; }
          case 1: if(TOK_continue>0) {
                    TOK_continue--;
                    fprintf(fp,"   continue;\n"); break; }
          case 2: if(TOK_break>0) {
                    TOK_break--; fprintf(fp,"   break;\n"); break; }
          case 3: if(TOK_return>0) {
                    TOK_return--; fprintf(fp,"   return ");
                    switch(layer(2)) {
                      case 0: break;
                      case 1: expr(rest); break;
                      default: out("Erro stmt-5-1"); }
                    fprintf(fp,";\n"); break; }
                  else break;
          default: out("Erro stmt-4"); }
        break;
      default: out("Erro stmt"); }}
```

```
void func(int rest,int ndecl,int nstmt) {
/* imprime no ficheiro uma funcao com ndecl declaracoes e nstmt
   instrucoes, rest indica quantas funcoes falta gerar. */

   /* cabecalho */
      /* tipo */
      tip(rest);
      /* nome da funcao */
      if(rest>1) fprintf(fp," nome(");
      else       fprintf(fp," main(");
      /* parametros */
      parameters(rest,layer(6));
      fprintf(fp,") {\n");

    /* declaracoes locais */
   declaration(rest,ndecl);

    /* instrucoes */
   while( nstmt>0 ) { stmt(rest); nstmt--; }
   fprintf(fp,"}\n\n"); }

main(int argc, char *argv[]) {
int aux;

if (argc!=3)    out("gerador ficheiro nfunctions");

/* abre ficheiro frequencias */
ins=fopen("tokens.txt","r");
if (ins==NULL)
    out("Erro na abertura do ficheiro de frequencias [tokens.txt]");

NUMBTOK_decl=NUMBTOK_stmt=0;
fscanf(ins,"auto=%d\n",&TOK_auto); NUMBTOK_decl+=TOK_auto;
fscanf(ins,"break=%d\n",&TOK_break); NUMBTOK_stmt+=TOK_break;
fscanf(ins,"case=%d\n",&TOK_case); NUMBTOK_stmt+=TOK_case;
fscanf(ins,"char=%d\n",&TOK_char); NUMBTOK_decl+=TOK_char;
fscanf(ins,"const=%d\n",&TOK_const); NUMBTOK_decl+=TOK_const;
fscanf(ins,"continue=%d\n",&TOK_continue); NUMBTOK_stmt+=TOK_continue;
fscanf(ins,"default=%d\n",&TOK_default); NUMBTOK_stmt+=TOK_default;
fscanf(ins,"do=%d\n",&TOK_do); NUMBTOK_stmt+=TOK_do;
fscanf(ins,"double=%d\n",&TOK_double); NUMBTOK_decl+=TOK_double;
fscanf(ins,"else=%d\n",&TOK_else); NUMBTOK_stmt+=TOK_else;
fscanf(ins,"enum=%d\n",&TOK_enum); NUMBTOK_decl+=TOK_enum;
fscanf(ins,"extern=%d\n",&TOK_extern); NUMBTOK_decl+=TOK_extern;
fscanf(ins,"float=%d\n",&TOK_float); NUMBTOK_decl+=TOK_float;
fscanf(ins,"for=%d\n",&TOK_for); NUMBTOK_stmt+=TOK_for;
fscanf(ins,"goto=%d\n",&TOK_goto); NUMBTOK_stmt+=TOK_goto;
fscanf(ins,"if=%d\n",&TOK_if); NUMBTOK_stmt+=TOK_if;
fscanf(ins,"int=%d\n",&TOK_int); NUMBTOK_decl+=TOK_int;
fscanf(ins,"long=%d\n",&TOK_long); NUMBTOK_decl+=TOK_long;
fscanf(ins,"register=%d\n",&TOK_register); NUMBTOK_decl+=TOK_register;
fscanf(ins,"return=%d\n",&TOK_return); NUMBTOK_stmt+=TOK_return;
fscanf(ins,"short=%d\n",&TOK_short); NUMBTOK_decl+=TOK_short;
fscanf(ins,"signed=%d\n",&TOK_signed); NUMBTOK_decl+=TOK_signed;
fscanf(ins,"sizeof=%d\n",&TOK_sizeof); NUMBTOK_stmt+=TOK_sizeof;
fscanf(ins,"static=%d\n",&TOK_static); NUMBTOK_decl+=TOK_static;
fscanf(ins,"struct=%d\n",&TOK_struct); NUMBTOK_decl+=TOK_struct;
fscanf(ins,"switch=%d\n",&TOK_switch); NUMBTOK_stmt+=TOK_switch;
fscanf(ins,"typedef=%d\n",&TOK_typedef); NUMBTOK_decl+=TOK_typedef;
fscanf(ins,"union=%d\n",&TOK_union); NUMBTOK_decl+=TOK_union;
fscanf(ins,"unsigned=%d\n",&TOK_unsigned); NUMBTOK_decl+=TOK_unsigned;
fscanf(ins,"void=%d\n",&TOK_void); NUMBTOK_decl+=TOK_void;
fscanf(ins,"volatile=%d\n",&TOK_volatile); NUMBTOK_decl+=TOK_volatile;
fscanf(ins,"while=%d\n",&TOK_while); NUMBTOK_stmt+=TOK_while;
```

```
fclose(ins);

/* abre ficheiro saida */
fp=fopen(argv[1],"w");
if (fp==NULL)   out("Erro na escrita do ficheiro");
srand((NUMBTOK_decl+NUMBTOK_stmt) % RAND_MAX);

/* obtem numero de funcoes a gerar */
fnumb=atoi(argv[2]);
if (fnumb<1)    out("Deve haver, pelo menos, uma funcao\n");

/* gera declaracoes globais */
declaration(fnumb+1,NUMBTOK_decl/(fnumb+1));

/* gera funcoes */
aux=fnumb;
while (fnumb>0) {
   func(fnumb,NUMBTOK_decl/(aux+1),NUMBTOK_stmt/aux);
   fnumb--; }

fclose(fp); }
```

8. Clusters

```
clear all
%Le e carrega variaA com o ficheiro com as 5 medidas de cada compilador
load todos5.txt
variaA=todos5;
% N´umero de “clusters”
nclust=5;
% cria 5 “clusters” usando com distancia eucliadiana
fT=clusterdata(variaA, nclust);
or k=1:nclust
    aux=T==k;
    auxg=[aux,aux,aux,aux,aux,aux];
    au=sum (aux);
    nel(k)=au;
    centro(k,:)=sum((auxg.*variaA))./au;
end
%Cria ficheiros com o centro centrotodos5
%                   o numero de elementos neltodos5
%                   a identificaçao do cluster “clusters”todos5
%                   os elementos parestodos5
centro
nel
T
variaA
save 'centrotodos5' centro -ascii -tabs
nel=nel';
save 'neltodos5' nel -ascii -tabs
save 'clustertodos5' T -ascii -tabs
save 'parestodos5' variaA -ascii -tabs
```

Anexo 1. da Secção 5.4.

(ANEXO 5.4.1)

Quadros das medidas dos compiladores

Valores da Entropia de Shanon do primeiro grupo fundamental da amostra

Compilador	Entropia	Compilador	Entropia
1	3,86290	**19**	3,91421
2	3,89682	**20**	3,80925
3	3,83133	**21**	3,83729
4	3,97526	**22**	3,88917
5	3,47483	**23**	3,71407
6	3,90197	**24**	3,47134
7	3,95532	**25**	3,65931
8	3,82718	**26**	3,75582
9	3,92959	**27**	3,93302
10	3,85659	**28**	3,80853
11	3,63097	**29**	3,87188
12	3,87809	**30**	3,65416
13	4,55525	**31**	3,30668
14	3,55883	**32**	3,83449
15	3,92847	**33**	3,75263
16	3,79657	**34**	3,94267
17	3,68436	**35**	3,91800
18	3,91400	**36**	3,78091

Valores da Entropia de Shanon do segundo grupo fundamental da amostra

Programa	Entropia	Programa	Entropia	Programa	Entropia
Vários	2,64338	frame	2,70742	plot2	2,57899
Matrizes	2,77477	getopt	2,45307	plot3	2,65915
Calculo Num	2,76619	gif	3,24908	scanner	3,10856
Obscuro	3,71034	hash	3,19796	~string	3,20944
Compilador	3,69904	include	2,99683	stype	2,99207
Primeiro	1,95027	list	2,88239	test	2,87372
color	2,85407	matrix	2,69966	tree	2,93787
cpp	3,21327	memory	3,00457	void	2,27875
expre	2,55593	misc	2,61749	wrapfunct	2,84304
fio	3,48306	naming	2,6633	ban3	2,53186
font	1,94105	pixmap	2,75022	ban2	2,53186
				ban1	2,53186

Valores da Entropia de Rényi do primeiro grupo fundamental da amostra

	alfa			
Compilador	**1/8**	**1/3**	**3**	**8**
1	4,5030	4,3650	-0,003	-0,00000001
2	4,5455	4,4853	-0,005	-0,00000002
3	4,6609	4,5968	-0,004	-0,00000001
4	4,7043	4,7458	-0,008	-0,00000007
5	4,0636	4,1384	-0,010	-0,00000022
6	4,6567	4,7152	-0,008	-0,00000007
7	4,6686	4,6483	-0,006	-0,00000007
8	4,6074	4,5684	-0,006	-0,00000006
9	4,6177	4,5890	-0,005	-0,00000002
10	4,5665	4,5548	-0,006	-0,00000005
11	4,5078	4,5074	-0,028	-0,00001122
12	4,5927	4,6284	-0,009	-0,00000019
13	4,6245	4,6564	-0,036	-0,00002732
14	4,4334	4,4041	-0,012	-0,00000105
15	4,6288	4,6247	-0,006	-0,00000004
16	4,5763	4,5848	-0,005	-0,00000002
17	4,5007	4,4780	-0,010	-0,00000064
18	4,6265	4,6159	-0,006	-0,00000004
19	4,6235	4,6050	-0,008	-0,00000022
20	4,5762	4,5813	-0,008	-0,00000010
21	4,6182	4,5986	-0,006	-0,00000008
22	4,5897	4,5366	-0,013	-0,00000265
23	4,2507	4,2583	-0,011	-0,00000081
24	4,5772	4,4983	-0,011	-0,00000064
25	4,5064	4,4958	-0,008	-0,00000012
26	4,6128	4,5712	-0,005	-0,00000003
27	4,6071	4,5741	-0,008	-0,00000024
28	4,6530	4,6129	-0,007	-0,00000010
29	4,4497	4,4511	-0,012	-0,00000081
30	4,4539	4,3631	-0,018	-0,00000256
31	4,6251	4,6122	-0,007	-0,00000008
32	4,5103	4,5024	-0,009	-0,00000034
33	4,6335	4,6295	-0,006	-0,00000008
34	4,8532	5,3876	-0,253	-0,00279300
35	4,6415	4,5833	-0,007	-0,00000009
36	4,5948	4,5481	-0,008	-0,00000010

Correlação de gama longa nos 36 programas

Programa	α com Lcr	α com ale	Programa	α com Lcr	α com ale
1	0,8906	0,4682	**19**	0,8067	0,4965
2	0,8141	0,4810	**20**	0,7813	0,5041
3	0,8880	0,5117	**21**	0,7620	0,5137
4	0,8327	0,4805	**22**	0,8261	0,5033
5	0,8265	0,5024	**23**	0,7810	0,4938
6	0,7041	0,4473	**24**	0,7871	0,4729
7	0,8326	0,5012	**25**	0,8595	0,5025
8	0,8326	0,5041	**26**	0,8647	0,4960
9	0,7944	0,4886	**27**	0,8798	0,4991
10	0,8926	0,4388	**28**	0,7892	0,4987
11	0,7355	0,4933	**29**	0,8792	0,5005
12	0,8544	0,5088	**30**	0,8080	0,5018
13	0,8428	0,4539	**31**	0,8368	0,4590
14	0,7579	0,4750	**32**	0,7968	0,4459
15	0,7467	0,4712	**33**	0,8637	0,4984
16	0,8997	0,4388	**34**	0,8401	0,4981
17	0,7665	0,4162	**35**	0,6841	0,4908
18	0,8853	0,4614	**36**	0,7702	0,4776

Correlação de gama longa para programas gerados com o mesmo número de linhas

Programa	Correlação de gama	Programa	Correlação de gama
1	0,5667	**19**	0,4965
2	0,5004	**20**	0,4111
3	0,4937	**21**	0,3676
4	0,3192	**22**	0,5031
5	0,2416	**23**	0,4788
6	0,4473	**24**	0,4729
7	0,3012	**25**	0,2537
8	0,5041	**26**	0,4591
9	0,4289	**27**	0,4103
10	0,4388	**28**	0,5033
11	0,3255	**29**	0,4988
12	0,1793	**30**	0,4297
13	0,3927	**31**	0,2537
14	0,2011	**32**	0,5991
15	0,1233	**33**	0,3103
16	0,4388	**34**	0,3693
17	0,4183	**35**	0,2505
18	0,2441	**36**	0,4023

Correlação de gama longa para o Grupo C.

Programa	Correlação de gama longa	Programa	Correlação de gama longa
1	0,5762	**19**	0,5031
2	0,5801	**20**	0,6001
3	0,557	**21**	0,7084
4	0,6214	**22**	0,7084
5	0,5625	**23**	0,7006
6	0,5397	**24**	0,6489
7	0,6029	**25**	0,7101
8	0,5553	**26**	0,7584
9	0,5989	**27**	0,7666
10	0,5358	**28**	0,6446
11	0,5044	**29**	0,6496
12	0,6268	**30**	0,5661
13	0,6072	**31**	0,4569
14	0,5823	**32**	0,5529
15	0,6075	**33**	0,5311
16	0,8173	**34**	0,6254
17	0,5677	**35**	0,5814
18	0,7407	**36**	0,6738

Anexo 2. da Secção 5.4.

(ANEXO 5.4.2)

Entropias locais

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 31

31 Compilador	31 Grupo C	Médias 1	Médias 2	
div 2				
2,5677	3,0234			
1,6609	1,3267	2,1143	2,1751	.
div 3				
2,2907	2,3001			
1,6044	1,5420			
1,2357	1,3214	1,7103	1,7212	.
div 4				
2,4188	2,3762			
1,6044	1,5900			
1,1941	2,0785			
0,9761	0,5420	1,5484	1,6467	.
div 5				
2,4557	2,0021			
1,4386	1,3210			
1,1609	1,0012			
0,9589	0,8665			
0,8162	0,5014	1,3661	1,1384	.
div 6				
2,4636	2,4531			
1,1116	1,1154			
1,1772	1,0023			
0,9554	0,7104			
0,8161	0,5420			
0,7066	0,6549	1,2051	1,0797	.
div 7				
2,3938	0,0987			
1,2367	1,8032			
1,1863	1,0123			
0,8823	0,3983			
0,7986	0,6521			
0,7211	0,9834			
0,6245	0,6502	1,1205	0,7997	.

div 8			
2,4564	0,3421		
1,3270	2,0021		
1,1127	1,0031		
0,9428	0,6342		
0,7901	0,6832		
0,6786	0,6823		
0,6345	0,5376		
0,5514	2,2540	1,0617	1,0173

.

div 9			
2,5366	0,9832		
1,3708	0,4538		
0,8143	0,5342		
0,8665	0,5231		
0,6789	0,9004		
0,7103	0,5003		
0,6173	0,4421		
0,5696	0,3315		
0,4915	0,3223	0,9618	0,5545

.

div 10			
2,5905	2,0010		
1,3429	0,3943		
0,7481	0,4390		
0,9732	2,0321		
0,7946	0,5166		
0,6644	0,4321		
0,6126	1,4650		
0,5653	0,4856		
0,5265	0,3204		
0,4405	0,4102	0,9259	0,8496

.

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 8

8Compilador	8 grupo C	Médias 1	Médias 2
div 2			
2,6253	2,5158		
1,5874	1,5777	2,1064	2,0468
div 3			
2,5277	2,7141		
1,6766	1,5751		
1,2170	1,1006	1,8071	1,7966
div 4			
2,5047	2,1340		
1,6254	1,4931		
1,1323	1,1753		
0,9739	0,8635	1,5591	1,4165
div 5			
2,4011	1,7270		
1,5887	0,9367		
1,2641	0,9804		
0,9697	0,7042		
0,8127	0,7185	1,4073	1,0134
div 6			
2,4242	2,0085		
1,5630	1,0081		
1,2053	0,7529		
0,8208	0,9001		
0,8385	0,6751		
0,6975	0,6392	1,2582	0,9973
div 7			
2,3299	2,2548		
1,4726	0,9011		
1,1744	0,6982		
0,9962	0,7247		
0,7509	0,7223		
0,7229	0,6001		
0,6075	0,5429	1,1506	0,9206

div 8			
2,1277	1,3757		
1,3728	0,6930		
1,1565	0,6824		
0,9687	0,6361		
0,6925	0,3170		
0,7015	0,5849		
0,6360	0,5232		
0,5524	0,4879	1,0260	0,6625
div 9			
1,9501	2,0067		
1,3069	0,3756		
1,1614	0,7983		
0,9536	0,4879		
0,8382	0,6026		
0,5573	0,4965		
0,6403	0,5641		
0,5706	0,4763		
0,4963	0,3458	0,9416	0,6838
div 10			
19488,0000	23685,0000		
13515,0000	0,3466		
11773,0000	0,8134		
0,8992	0,5613		
0,8214	0,5637		
0,6309	0,5604		
0,5689	0,5378		
0,5774	0,4694		
0,5155	0,4409		
0,4547	0,4035	4478,0468	2368,9697

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 11

11Compilador	11 grupo C	Médias 1	Médias 2
div 2			
2,6806	2,7272		
1,5839	1,4490	2,1323	2,0881
div 3			
2,6406	2,6109		
1,6590	1,6490		
1,1597	0,9997	1,8198	1,7532
div 4			
2,5008	1,2306		
1,6200	0,9004		
1,1814	1,0045		
0,9182	0,6235	1,5551	0,9398
div 5			
2,4384	1,1036		
1,6309	1,0063		
1,2775	0,7301		
0,9142	0,6923		
0,7676	0,5355	1,4057	0,8136
div 6			
2,5017	1,3353		
1,6603	1,0030		
1,1936	0,6002		
0,9470	0,6452		
0,8125	0,5021		
0,6607	0,3315	1,2960	0,7362
div 7			
2,5719	1,3005		
1,5954	0,8093		
1,1768	0,7011		
1,0139	0,5767		
0,6778	0,3041		
0,7168	0,5021		
0,5790	0,2011	1,1902	0,6278

div 8			
2,6304	1,3002		
1,4200	0,6750		
1,1557	0,7332		
0,9803	0,4576		
0,8235	0,4387		
0,6696	0,5001		
0,6176	0,2231		
0,5241	0,3321	1,1027	0,5825
div 9			
2,6801	1,6501		
1,2753	0,3831		
1,1761	0,8732		
0,9224	0,4076		
0,8280	0,3856		
0,5809	0,0231		
0,6122	0,2561		
0,5643	0,2216		
0,4675	0,3052	1,0119	0,5006
div 10			
2,6995	1,0650		
1,1818	0,3221		
1,2251	0,6651		
0,9190	0,4357		
0,8087	0,3281		
0,7330	0,6531		
0,5513	0,3251		
0,5703	0,3221		
0,5062	0,2547		
0,4245	0,3002	0,9619	0,4671

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 33

33Compilador	33 grupo C	Médias 1	Médias 2
div 2			
2,6904	2,6837		
1,5680	1,5476	2,1292	2,1157
div 3			
2,5859	2,5965		
1,6152	1,6239		
1,1617	1,1579	1,7876	1,7928
div 4			
2,5650	1,8730		
1,6365	0,6543		
1,1758	0,9834		
0,9083	0,8732	1,5714	1,0960
div 5			
2,6795	1,0943		
1,5264	0,9843		
1,1482	0,6542		
0,9625	0,5001		
0,7405	0,3427	1,4114	0,7151
div 6			
2,6863	1,8430		
1,3970	0,4878		
1,1813	0,3427		
0,9148	0,6521		
0,8006	0,5982		
0,6413	0,3075	1,2702	0,7052
div 7			
2,6839	2,4320		
1,2092	0,7843		
1,1849	0,6544		
0,9185	0,4396		
0,8152	0,3064		
0,6848	0,4342		
0,5806	0,3065	1,1539	0,7653

div 8			
2,5338	2,0043		
1,1215	0,4985		
1,1223	0,8543		
0,8835	0,4532		
0,7314	0,5872		
0,6917	0,4983		
0,6026	0,5983		
0,5162	0,0293	1,0254	0,6904
div 9			
2,4644	1,0003		
1,1696	0,3916		
1,1073	0,6970		
0,9704	0,5642		
0,7795	0,4521		
0,6983	0,4000		
0,6001	0,4210		
0,5301	0,4100		
0,4615	0,3287	0,9757	0,5183
div 10			
2,4581	1,7632		
1,3168	0,0873		
0,9962	0,3065		
0,9265	0,7634		
0,6900	0,4893		
0,5910	0,4576		
0,6302	0,2874		
0,5412	0,4537		
0,4846	0,3003		
0,4216	0,1435	0,9056	0,5052

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 10

10Compilador	10 grupo D	Médias 1	Médias 2
div 2			
2,6021	2,6157		
1,5960	1,6319	2,0991	2,1238
div 3			
2,5100	2,5630		
1,6937	1,4996		
1,2038	1,1983	1,8025	1,7536
div 4			
2,4585	2,6001		
1,6243	1,1823		
1,0758	1,9463		
0,9863	0,8845	1,5362	1,6533
div 5			
2,3681	2,2816		
1,5896	1,4343		
1,2692	1,1178		
0,8330	0,8600		
0,8212	1,0101	1,3762	1,3408
div 6			
2,4004	2,0021		
1,5204	1,4205		
1,1837	1,0019		
0,9132	0,9531		
0,7824	0,6914		
0,7158	0,4076	1,2527	1,0794
div 7			
2,2345	3,0113		
1,3844	1,0024		
1,1394	1,1054		
0,9856	0,8448		
0,5977	0,7112		
0,7102	0,5015		
0,6385	0,5321	1,0986	1,1012

div 8			
2,0510	1,7851		
1,3009	0,3826		
1,1573	1,2191		
0,9506	0,8738		
0,7927	0,4985		
0,5400	0,0124		
0,6216	0,4008		
0,5794	0,0325	0,9992	0,6506
div 9			
1,8946	1,5282		
1,2112	1,3004		
1,1927	1,6989		
0,8747	0,0871		
0,8124	0,7617		
0,5637	0,4969		
0,5579	0,5402		
0,5596	0,2031		
0,5229	0,6411	0,9100	0,8064
div 10			
1,9410	1,0029		
1,3479	1,2774		
1,1847	0,8564		
0,9148	2,5004		
0,7979	0,6016		
0,6972	0,4269		
0,4474	0,3984		
0,5462	0,2271		
0,5053	0,3415		
0,4831	0,3400	0,8866	0,7973

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 28

28 Compilador	28 grupo D	Médias Comp.	Médias Grup.D
1582	1550		
div 2			
2,6186	2,5965		
1,6144	1,6184	2,1165	2,1075
div 3			
2,4160	2,4460		
1,6087	1,6165		
1,1854	1,1987	1,7367	1,7537
div 4			
2,5033	2,5310		
1,6464	1,6446		
1,2150	1,2184		
0,9311	0,9301	1,5740	1,5810
div 5			
2,5990	2,5083		
1,5463	1,5353		
1,1783	1,1755		
0,9552	0,9992		
0,8044	0,8064	1,4166	1,4049
div 6			
2,6654	2,6457		
1,2279	1,2387		
1,1966	1,1889		
0,9591	0,9561		
0,8002	0,8102		
0,7001	0,7006	1,2582	1,2567
div 7			
2,6991	2,6989		
1,1548	1,1520		
1,2269	1,2776		
0,9151	0,9051		
0,8023	0,8039		
0,6748	0,6648		
0,6174	0,6192	1,1558	1,1602

div 8			
2,7061	2,7039		
1,1324	1,1248		
1,1444	1,1488		
0,9363	0,9443		
0,7970	0,7893		
0,7088	0,7156		
0,6033	0,6143		
0,5566	0,5607	1,0731	1,0752
div 9			
2,6491	2,6118		
1,1124	1,1231		
0,9766	0,9816		
0,9114	0,9204		
0,7445	0,7511		
0,6945	0,6910		
0,6182	0,6202		
0,5559	0,5614		
0,5020	0,5091	0,9738	0,9744
div 10			
2,6204	2,6239		
1,1892	1,1807		
0,9037	0,9093		
0,9852	0,9749		
0,7939	0,7459		
0,6902	0,6812		
0,6034	0,6452		
0,5120	0,5090		
0,5111	0,5112		
0,4545	0,4498	0,9264	0,9231

Valores médias locais do grupo de controlo C em relação ao primeiro grupo (compiladores) 39

39 Compilador	39 grupo D	Médias Comp.	Médias Comp.
1582	1550		
div 2			
2,2532	2,1943		
1,6844	1,6043	1,9688	1,8993
div 3			
2,4872	2,4398		
1,6263	1,6300		
1,0841	1,0843	1,7325	1,7180
div 4			
2,3922	2,3820		
1,6464	1,6398		
1,2803	1,2795		
0,9311	0,8120	1,5625	1,5283
div 5			
2,3947	2,3876		
1,5291	1,5197		
1,1962	1,1897		
0,9318	0,9301		
0,7684	0,7643	1,3640	1,3583
div 6			
2,2901	2,2903		
1,3188	1,3034		
1,1534	1,1493		
0,9723	0,6571		
0,6817	0,5316		
0,6713	0,6700	1,1813	1,1003
div 7			
2,0353	2,0303		
1,1930	1,0870		
1,1958	1,1854		
0,9377	0,9250		
0,8395	0,7965		
0,5633	0,5321		
0,6020	0,5932	1,0524	1,0214

div 8			
1,8899	1,8798		
1,2886	1,2311		
1,1948	1,1932		
0,9191	0,9176		
0,8018	0,8067		
0,6740	0,5632		
0,5593	0,5400		
0,5437	0,3002	0,9839	0,9290
div 9			
1,9434	1,8745		
1,3775	1,2375		
1,1038	1,1037		
0,9325	0,8943		
0,7897	0,7698		
0,6950	0,5320		
0,5442	0.7439		
0,4535	0,3201		
0,4958	0,4843	0,9262	0,9020
div 10			
1,9802	1,9745		
1,4792	1,4743		
0,9448	0,9443		
0,9044	0,8911		
0,7610	0,6500		
0,6913	0,7102		
0,5892	0,6065		
0,4612	0.4312		
0,4215	0,4118		
0,4563	0,4459	0,8689	0,9010

Valores das entropias médias locais para os compiladores 8, 10, 11, 31 e 33

	8	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,1064	2,0468
3	1,8071	1,7966
4	1,5591	1,4165
5	1,4073	1,0134
6	1,2582	0,9973
7	1,1506	0,9206
8	1,0260	0,6625
9	0,9416	0,6838
10	4,4780	2,3689

	10	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,1064	2,0468
3	1,8071	1,7966
4	1,5591	1,4165
5	1,4073	1,0134
6	1,2582	0,9973
7	1,1506	0,9206
8	1,0260	0,6625
9	0,9416	0,6838
10	0,8946	0,7066

	11	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,1323	2,0881
3	1,8198	1,7532
4	1,5551	0,9398
5	1,4057	0,8136
6	1,296	0,7362
7	1,1902	0,6278
8	1,1027	0,5825
9	1,0119	0,5006
10	0,9619	0,4671

	31	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,0991	2,1238
3	1,8025	1,7536
4	1,5362	1,6533
5	1,3762	1,3408
6	1,2527	1,0794
7	1,0986	1,1012
8	0,9992	0,6506
9	0,9100	0,8064
10	0,8866	0,7973

	33	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,1292	2,1157
3	1,7876	1,7928
4	1,5714	1,0960
5	1,4114	0,7151
6	1,2702	0,7052
7	1,1539	0,7653
8	1,0254	0,6904
9	0,9757	0,5183
10	0,9056	0,5052

Valores das entropias médias locais para os compiladores 28 e 39

	28	
p	HM(p) do Compilador	HM(p) do Grupo C
2	2,1165	2,1075
3	1,7367	1,7537
4	1,5740	1,5810
5	1,4166	1,4049
6	1,2582	1,2567
7	1,1558	1,1602
8	1,0731	1,0752
9	0,9738	0,9744
10	0,9264	0,9231

	39	
p	HM(p) do Compilador	HM(p) do Grupo C
2	1,6844	1,6043
3	1,0841	1,0843
4	0,9311	0,8120
5	0,7684	0,7643
6	0,6713	0,6700
7	0,6020	0,5932
8	0,5437	0,3002
9	0,4958	0,4843
10	0,4563	0,4459

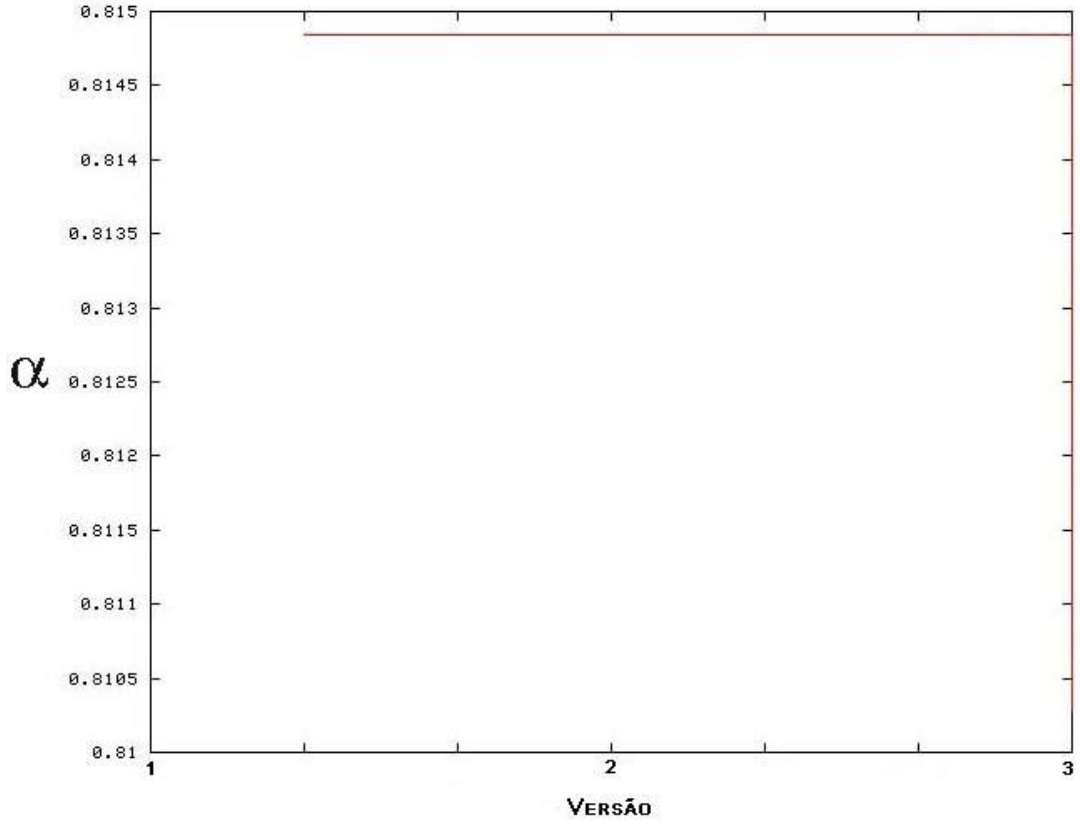
Anexo 1. da Secção 7.5.

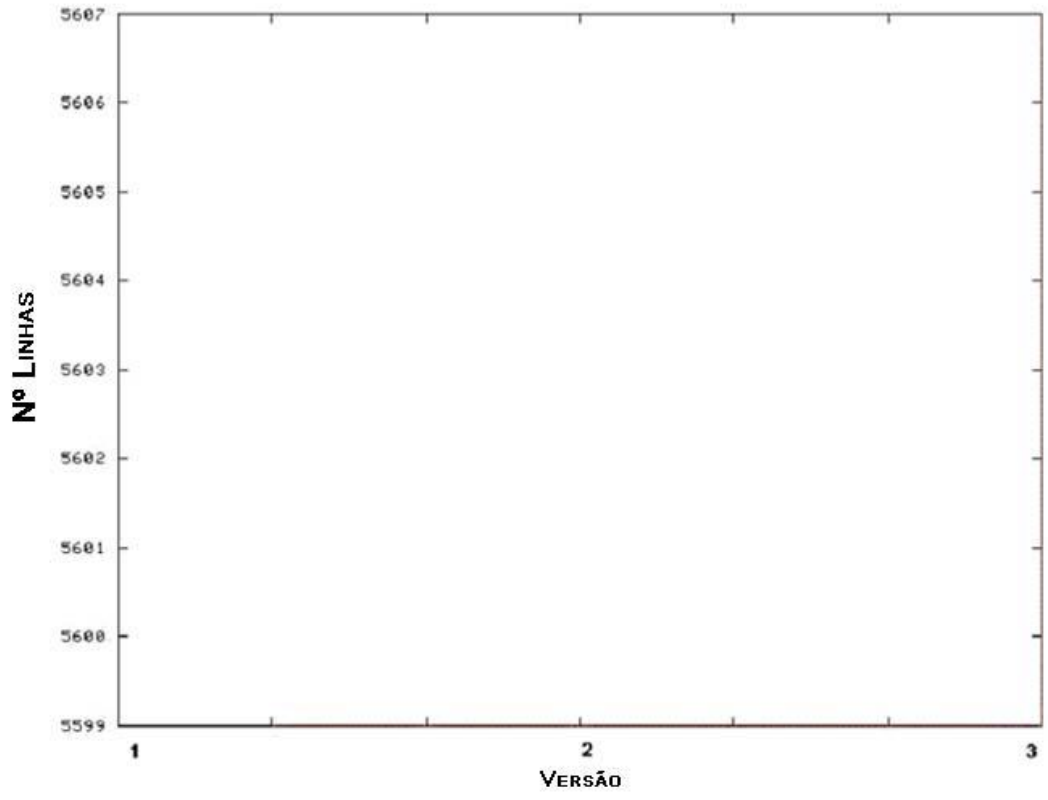
(ANEXO 7.5.1)

Gráficos da amostra codificada em Java

1º GRUPO

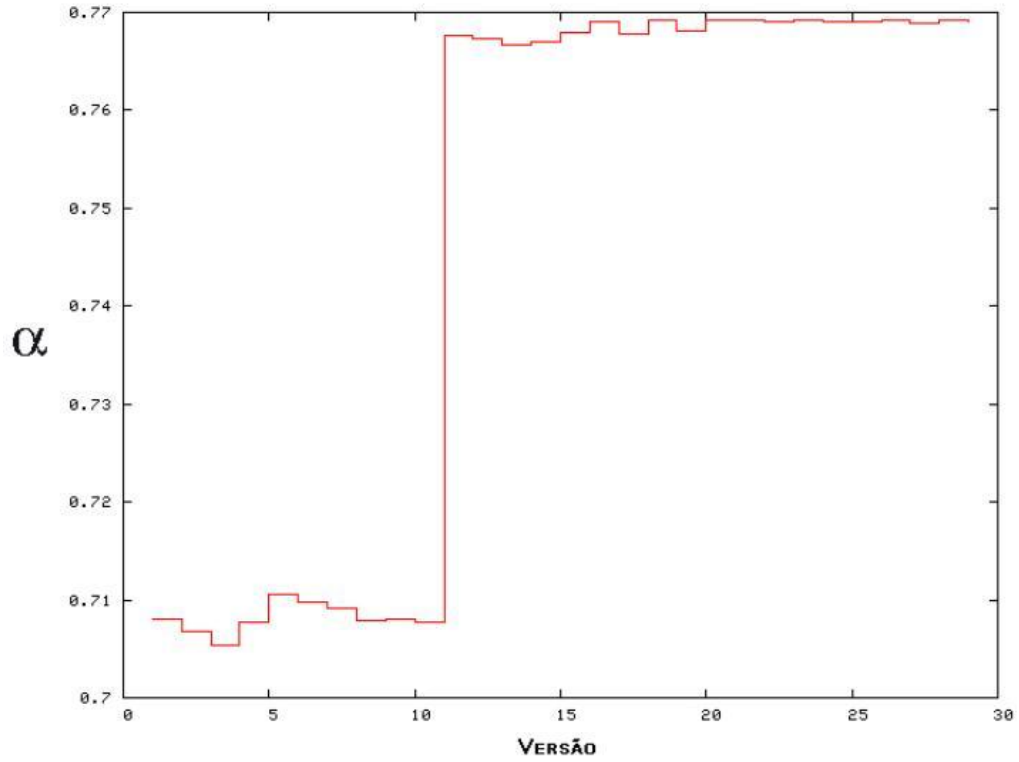
JDBCKeyGeneratorCreateCommand.java

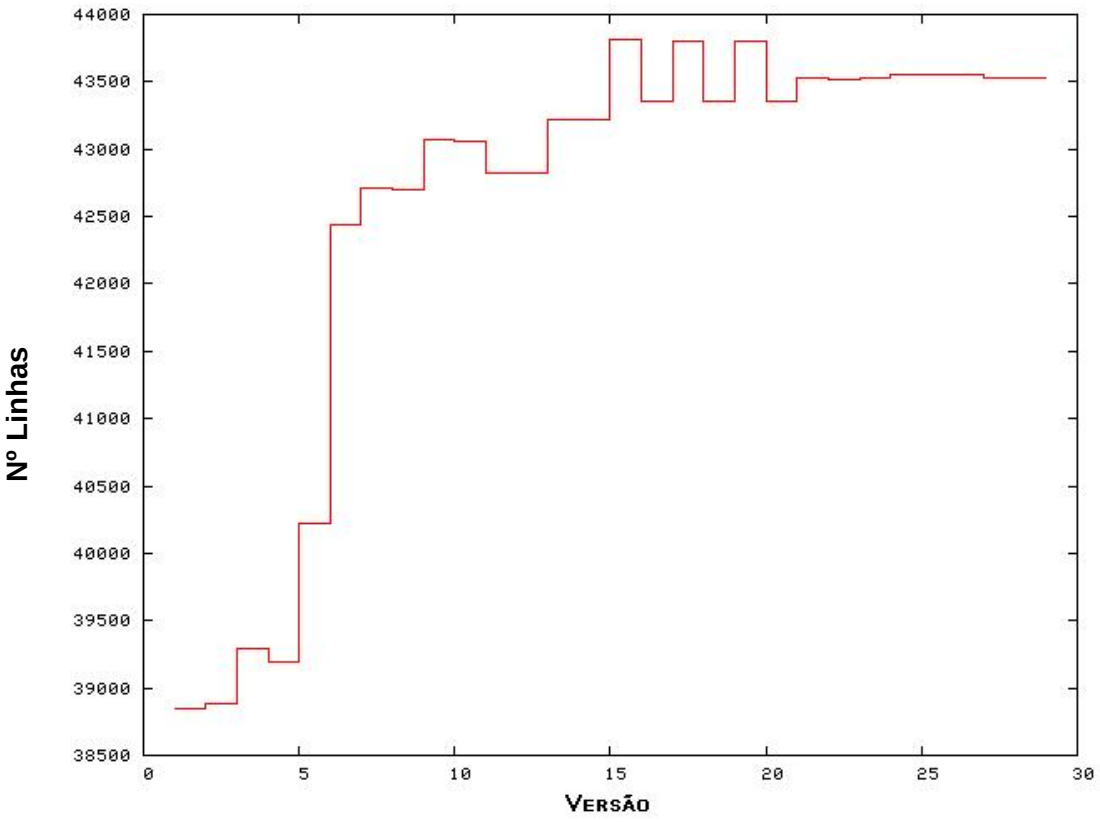


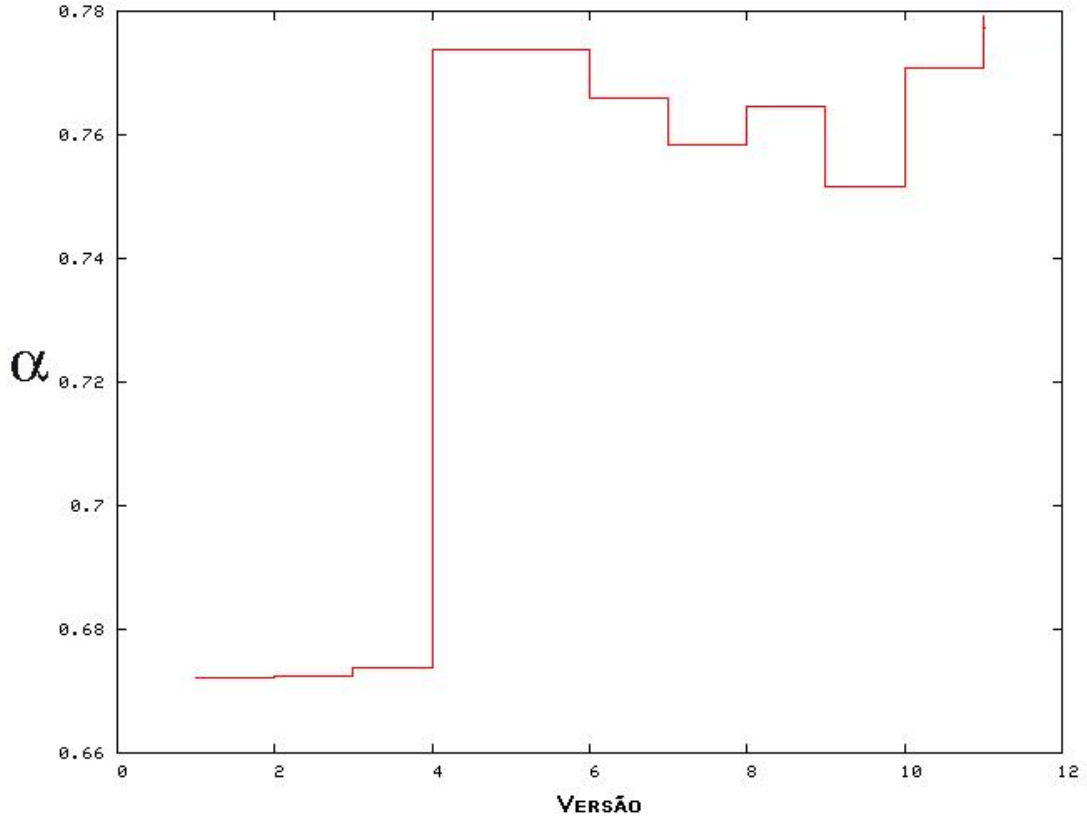


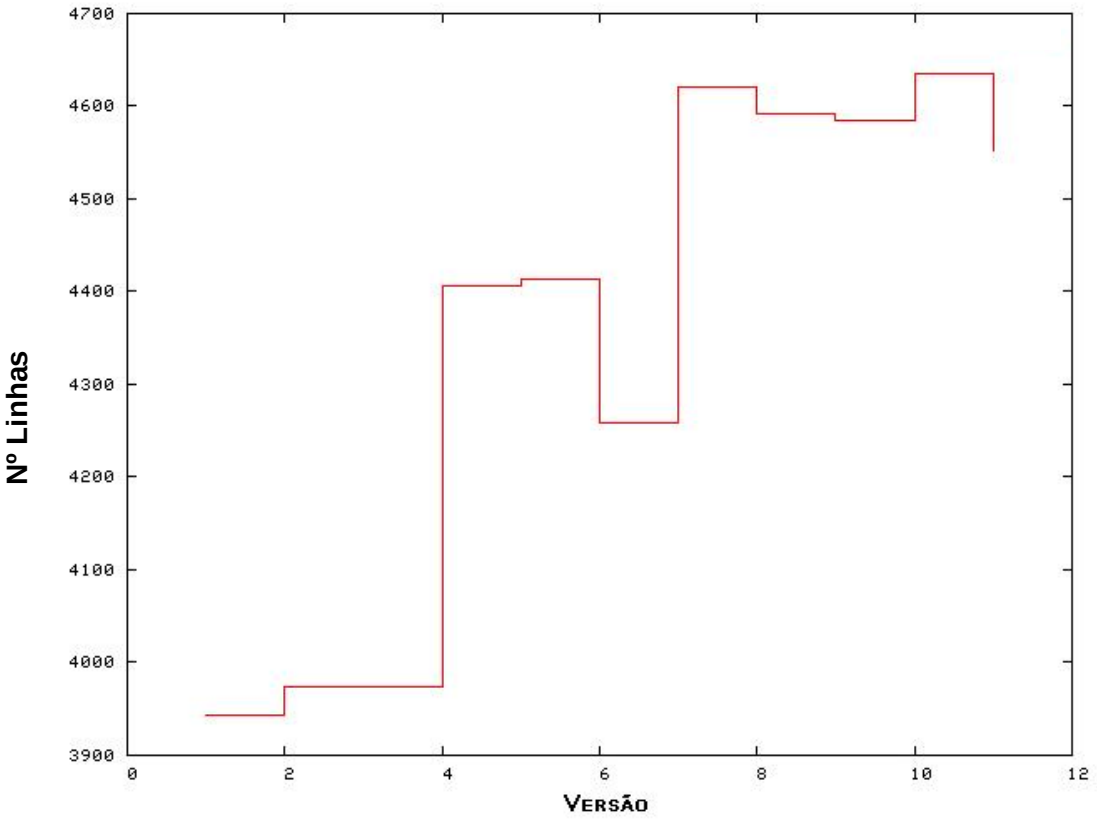
2º GRUPO

JDBCEJBQLCompiler.java

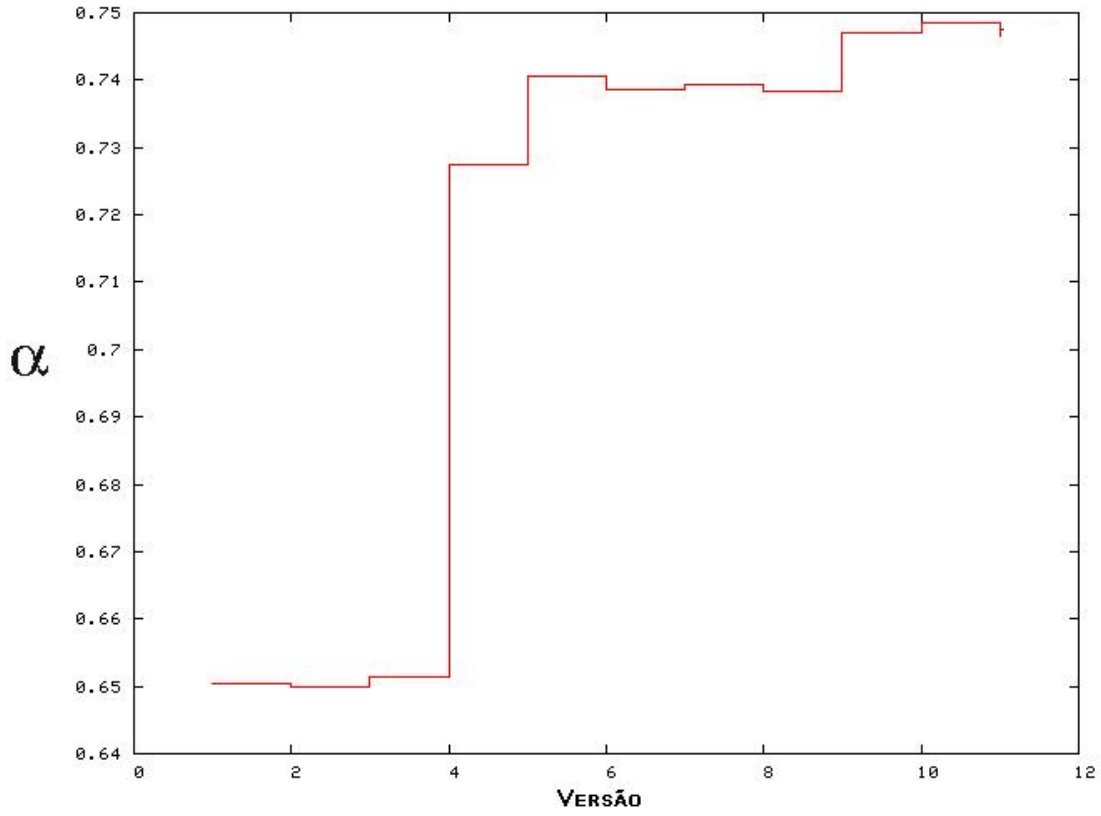


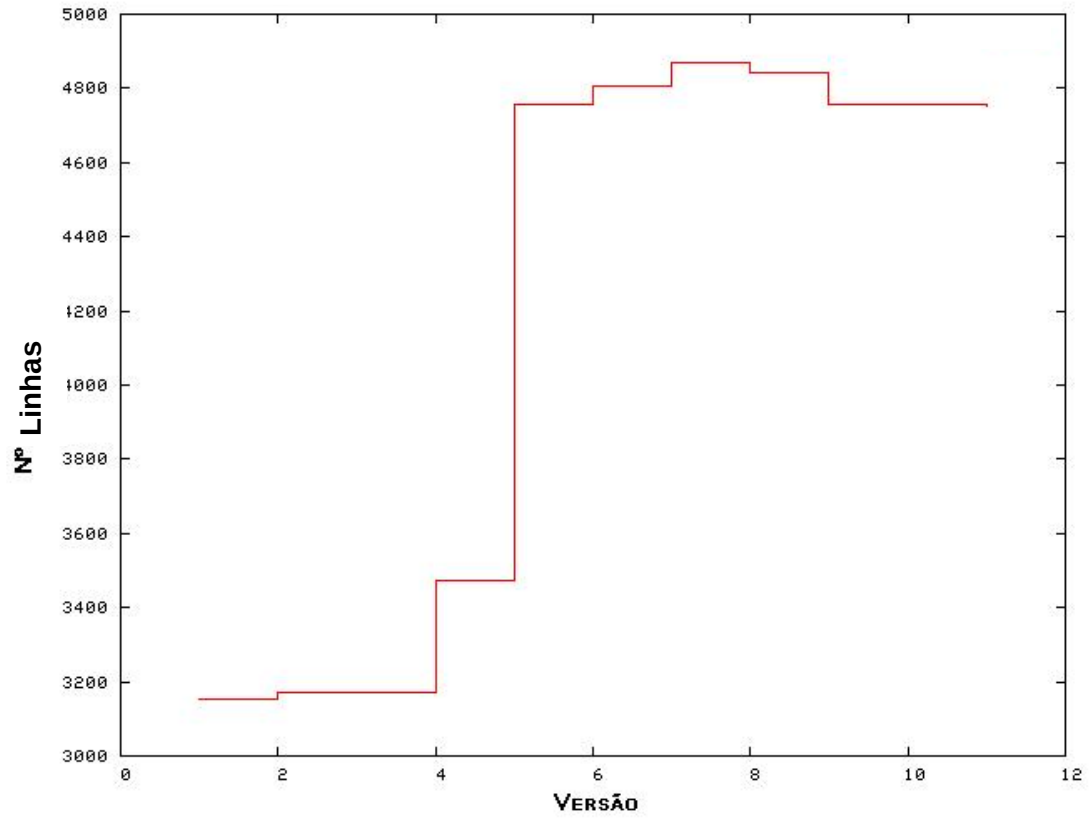



α
0.78
0.76
0.74
0.72
0.7
0.68
0.66
0
2
4
6
8
10
12
Versão

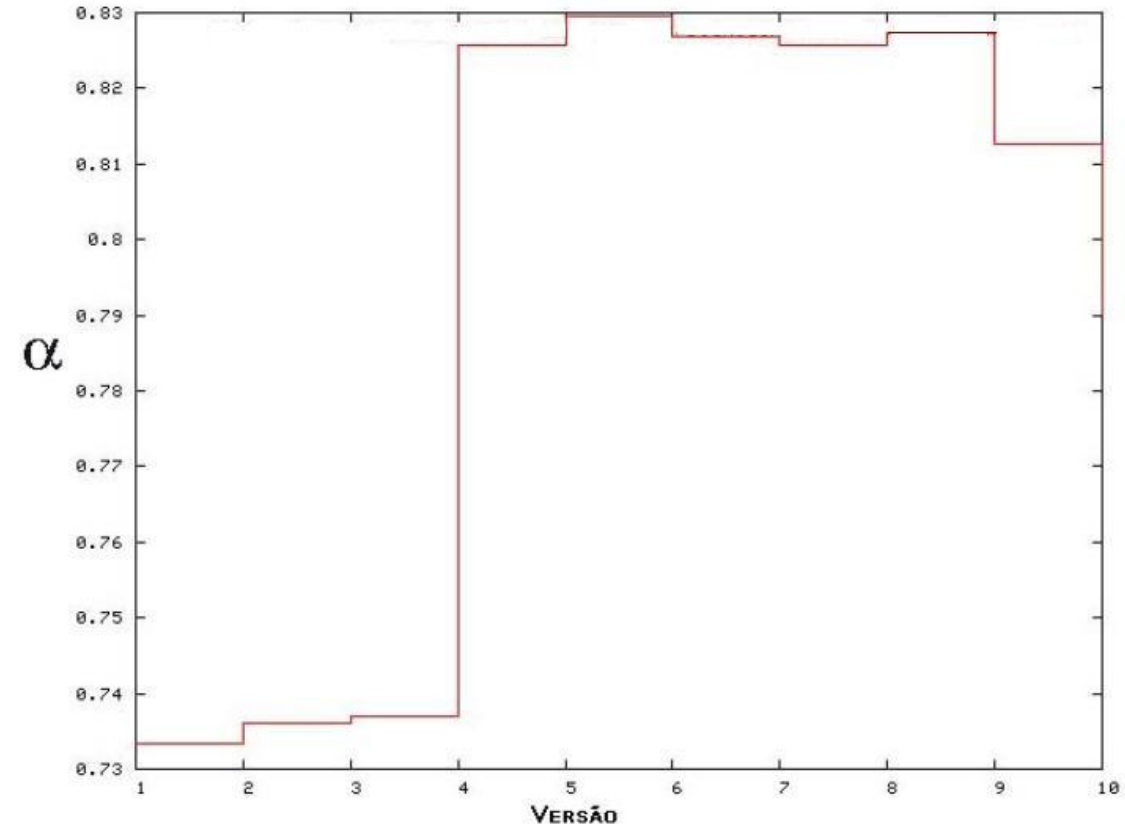

Nº Linhas
4700
4600
4500
4400
4300
4200
4100
4000
3900
0
2
4
6
8
10
12
Versão

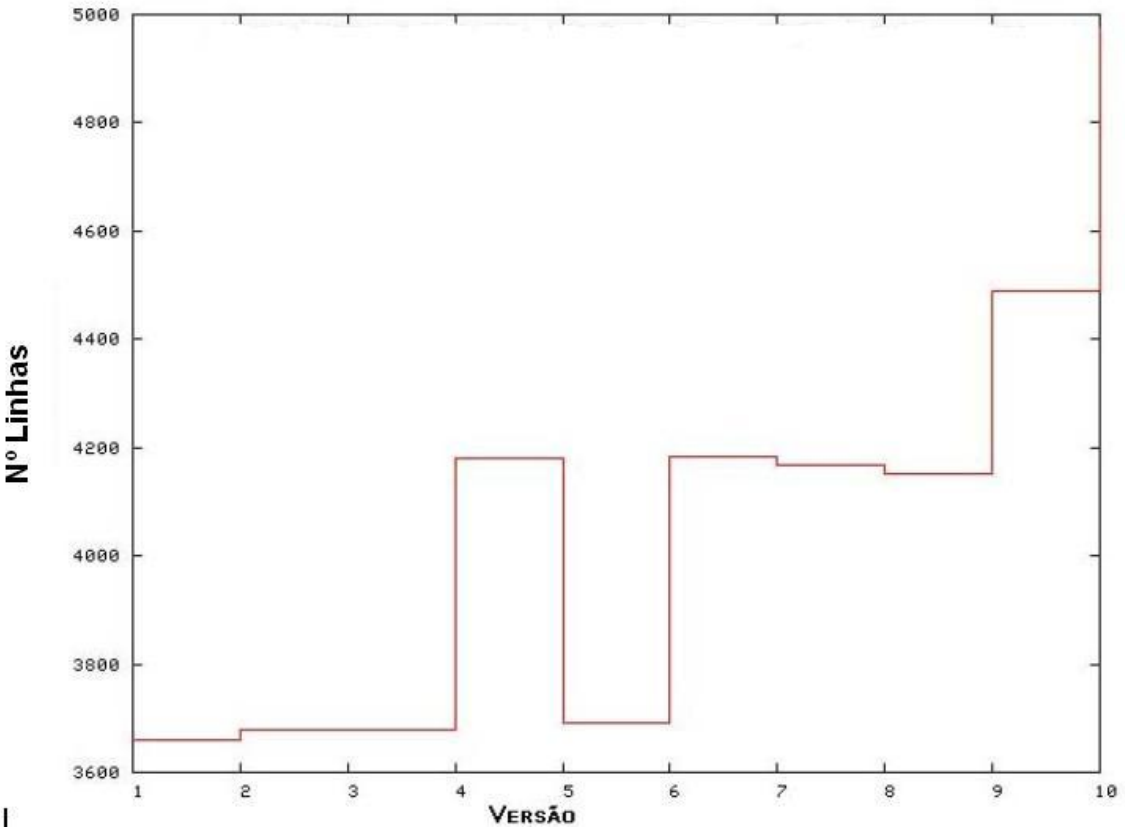
JDBCDeleteRelationsCommand.java



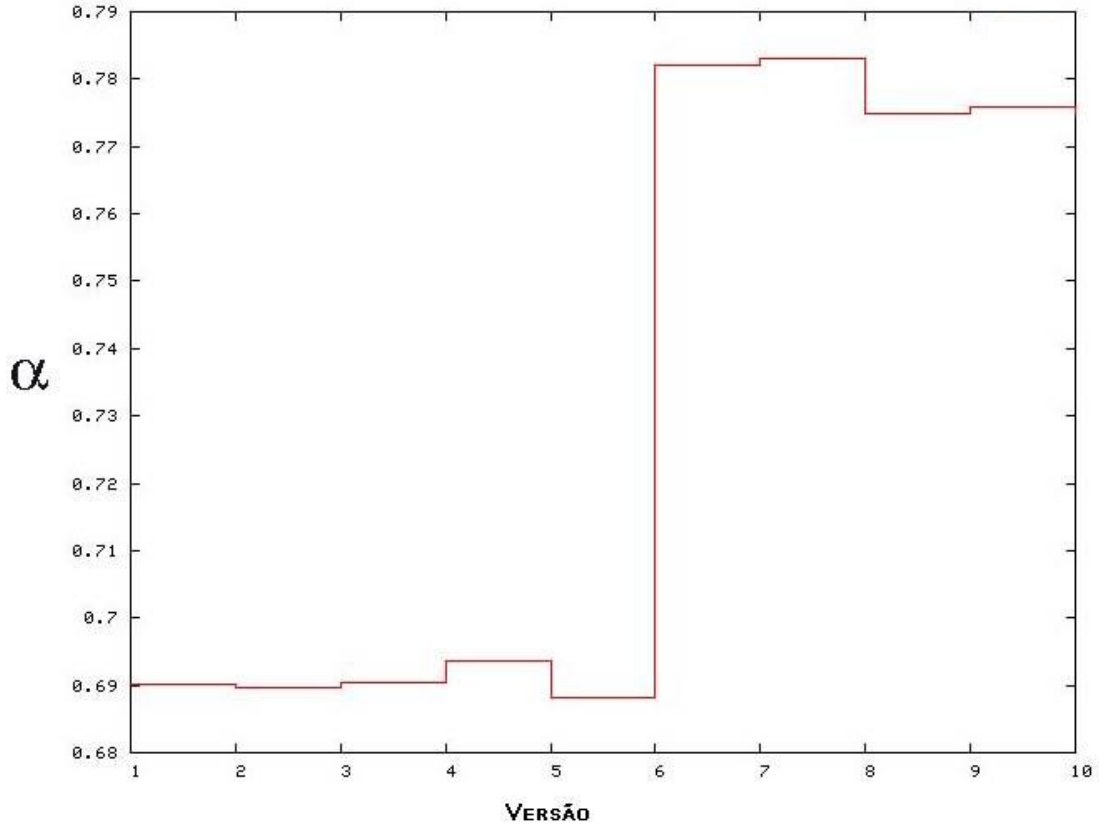


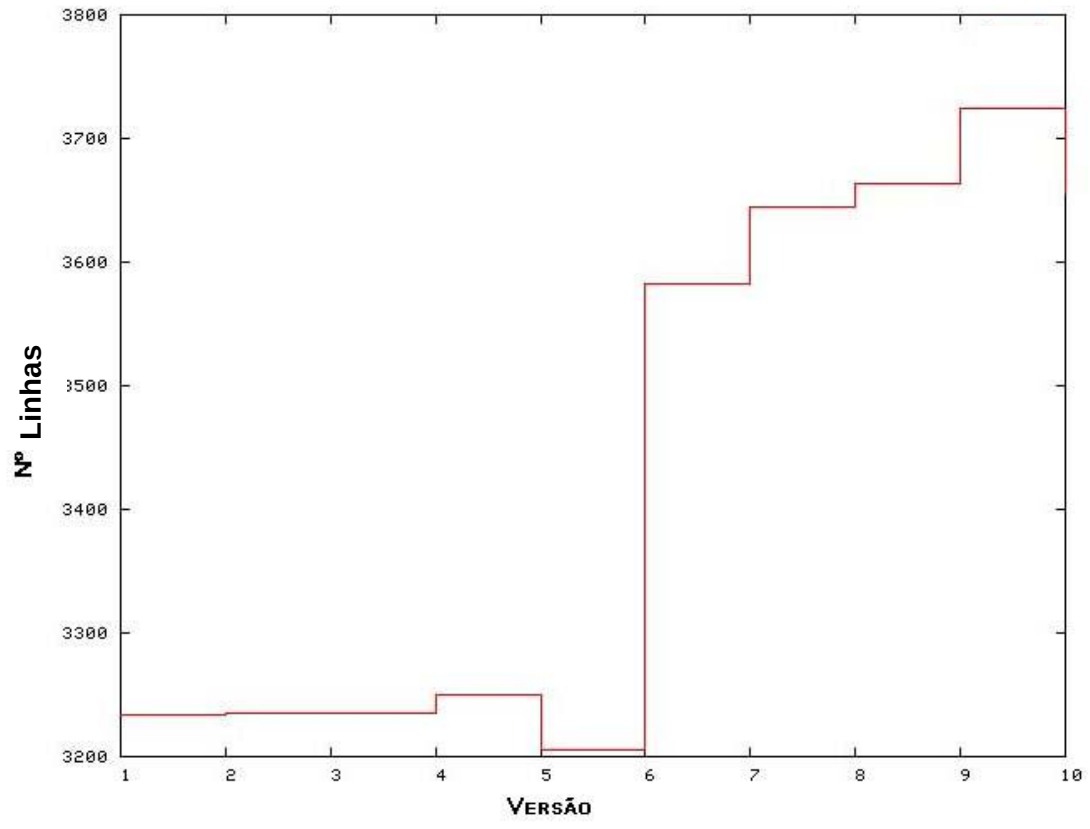
BeanCacheMonitor.java



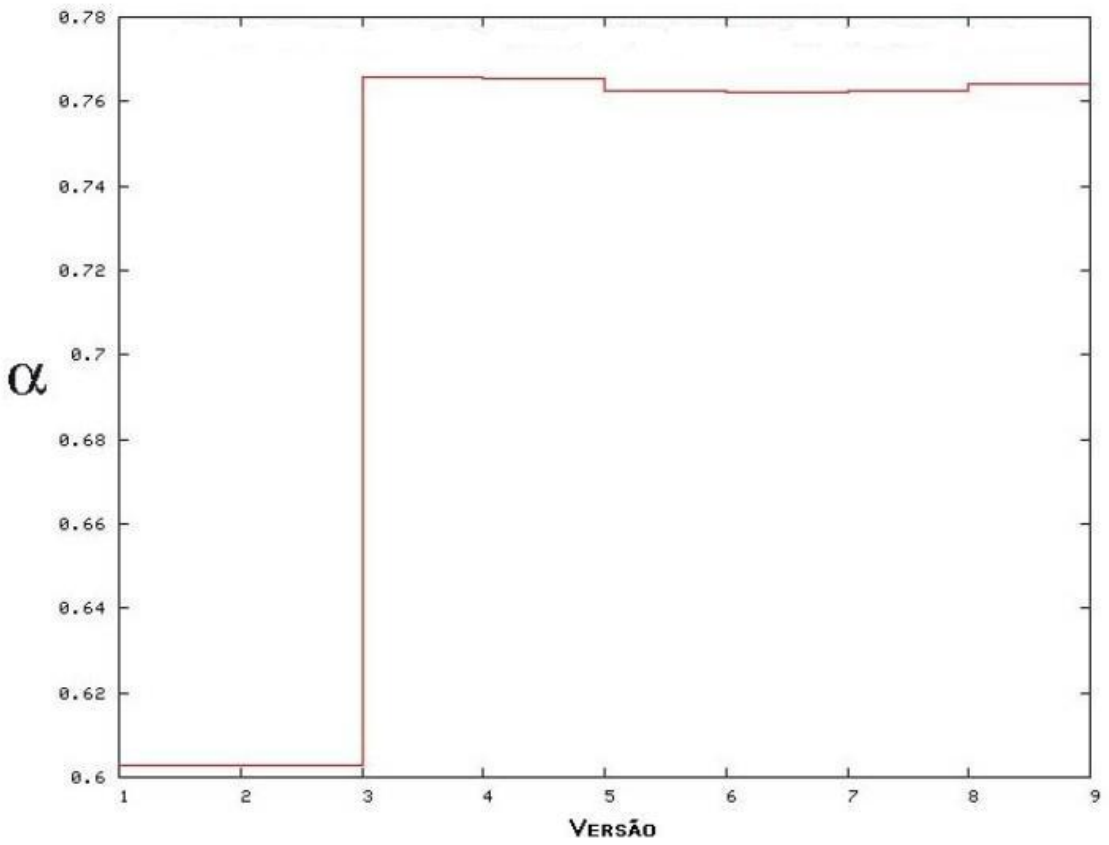


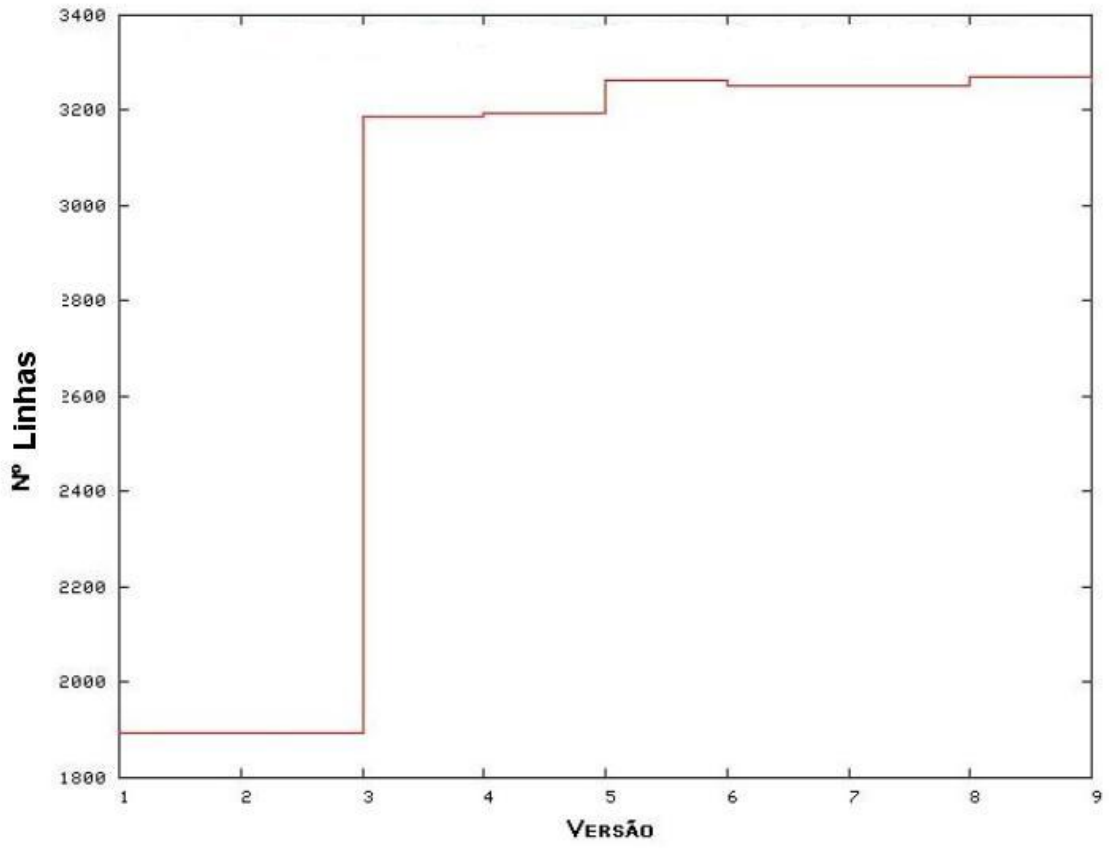
J2eeApplicationMetaData.java



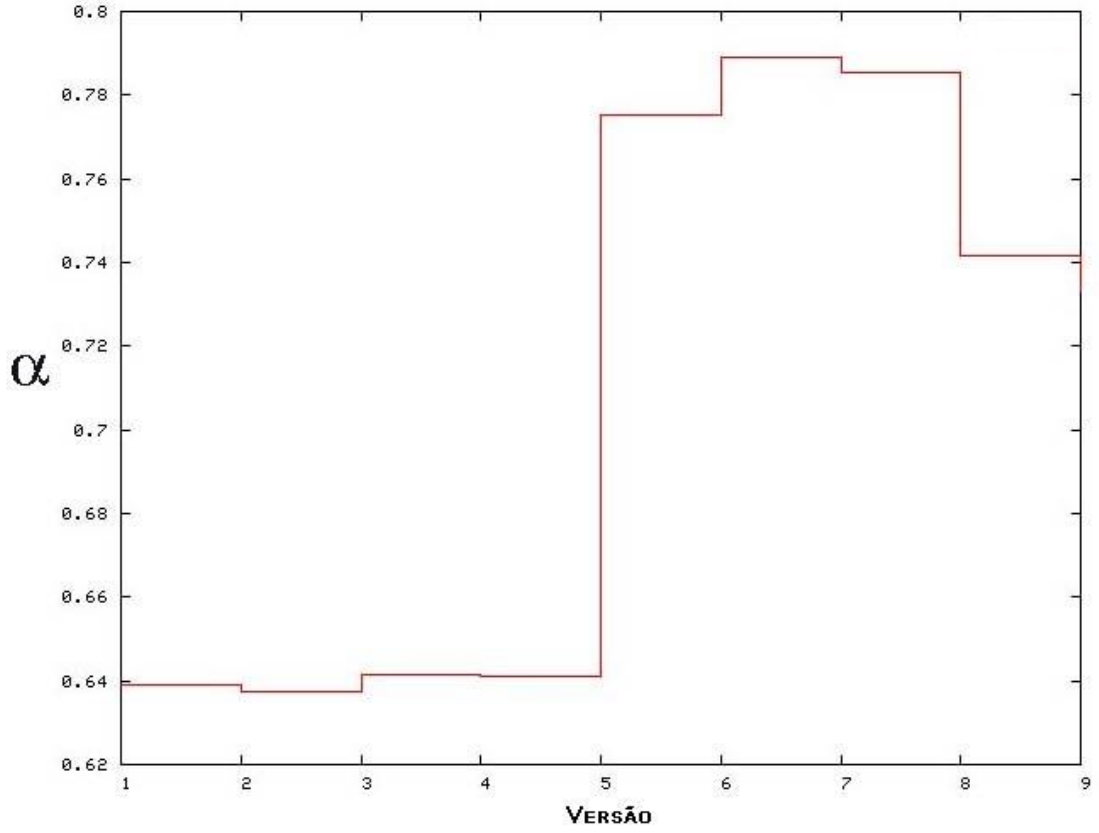


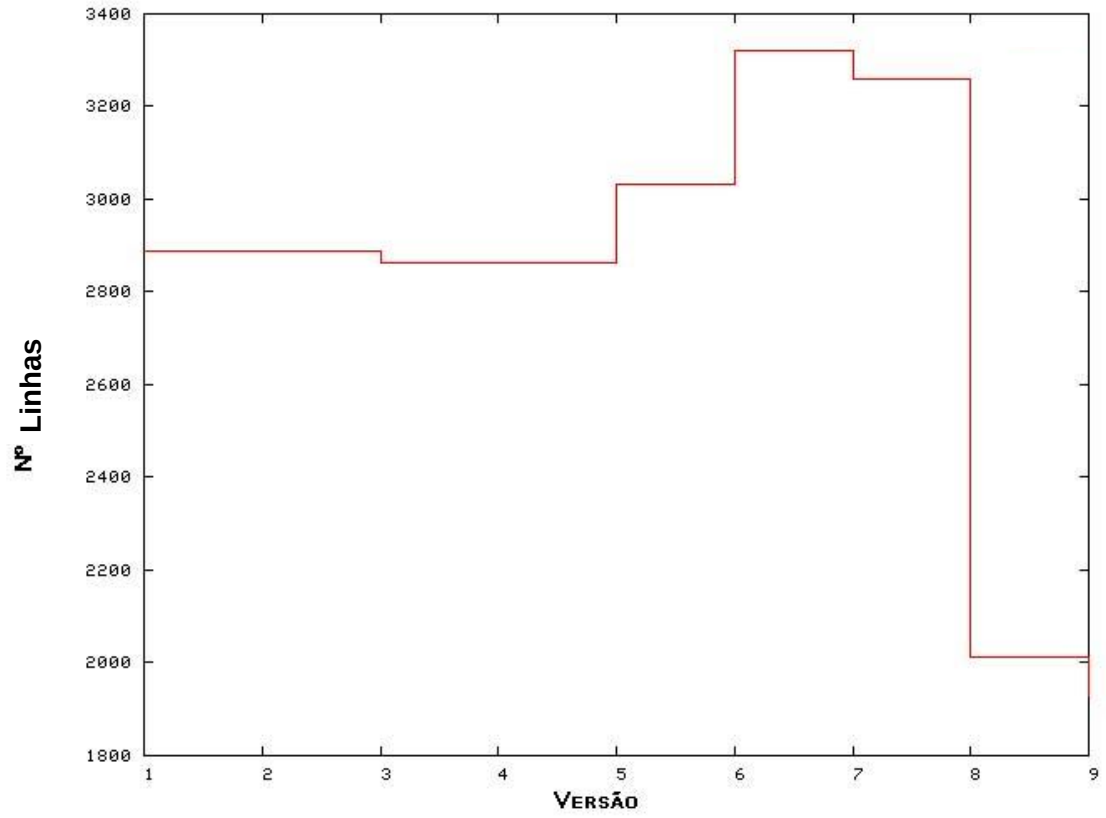
EnvEntryMetaData.java



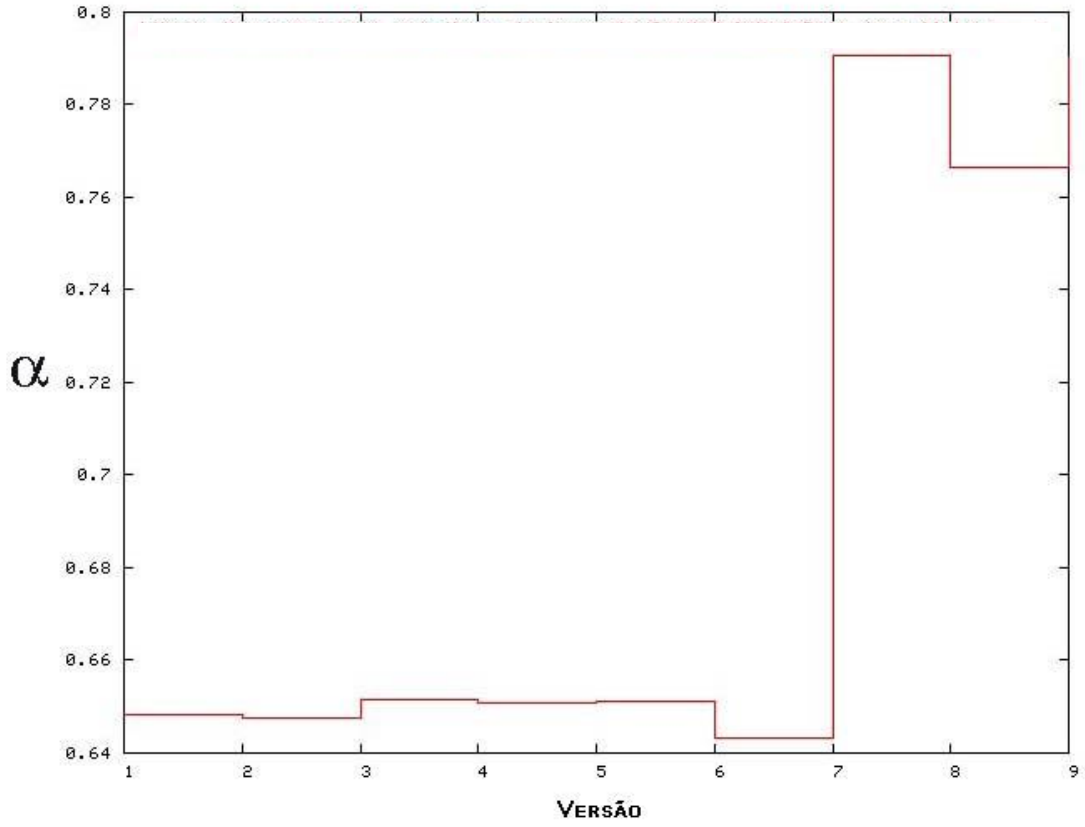


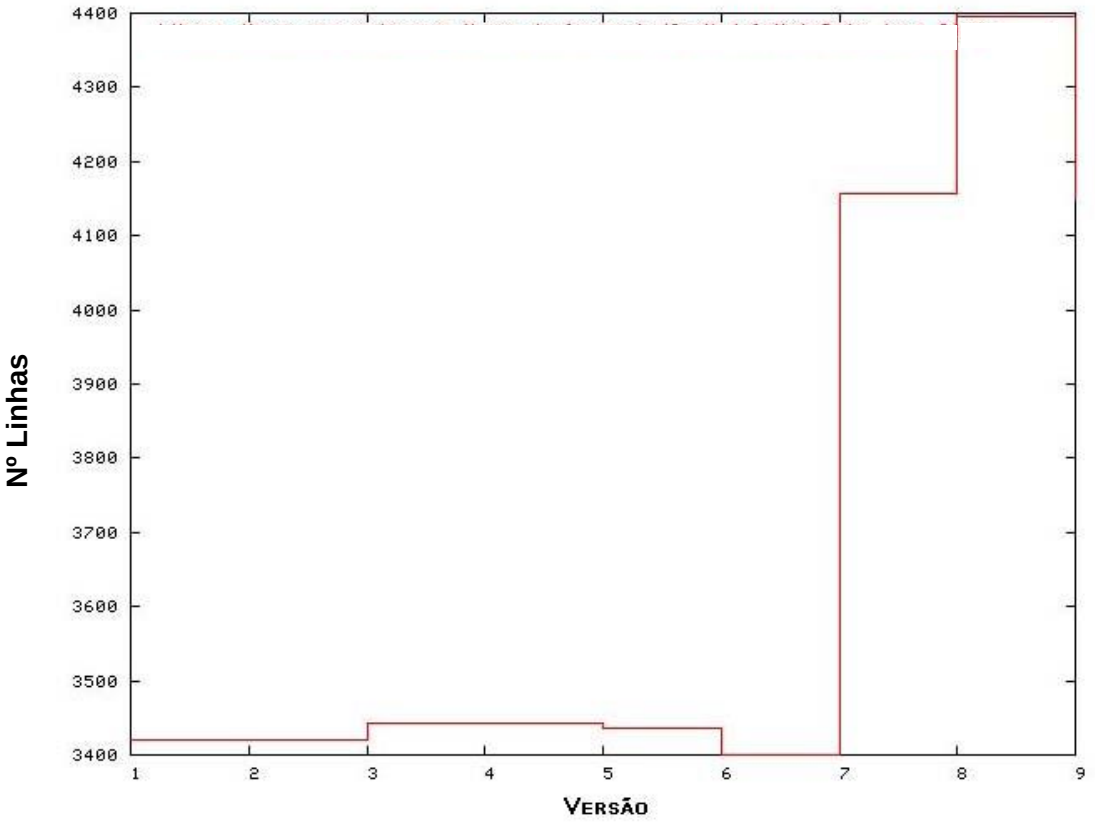
JDBCFindEntityCommand.java



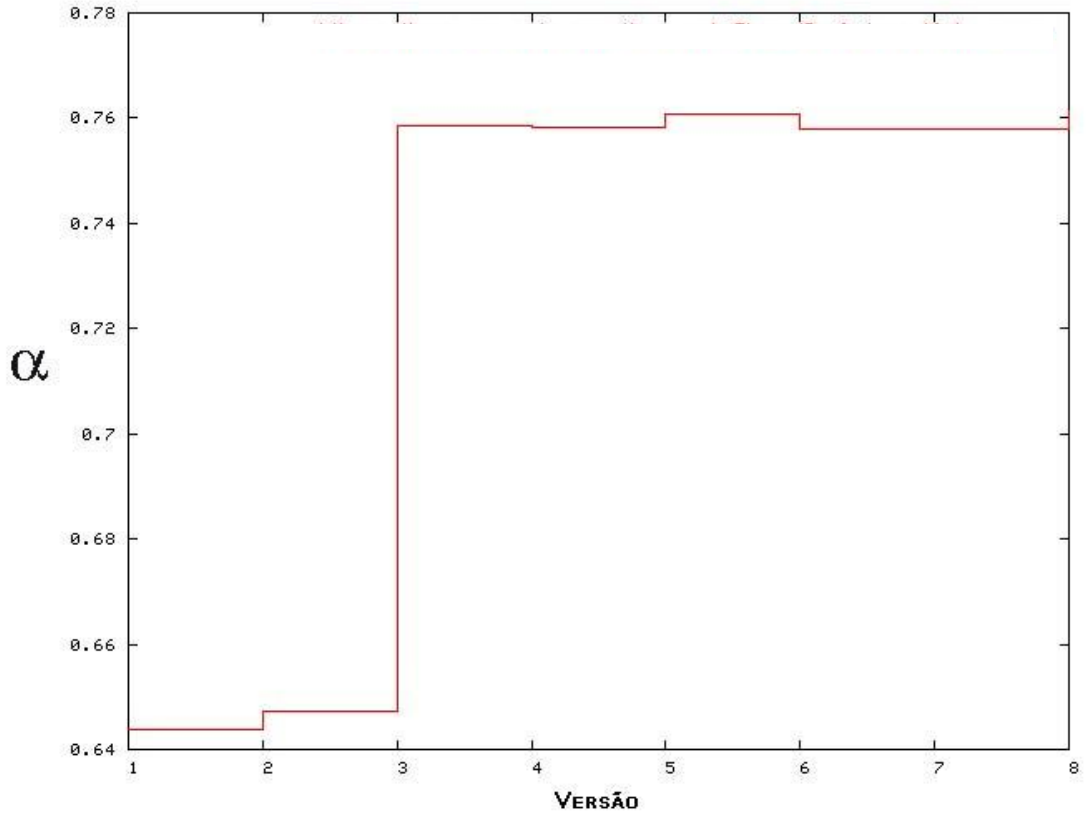


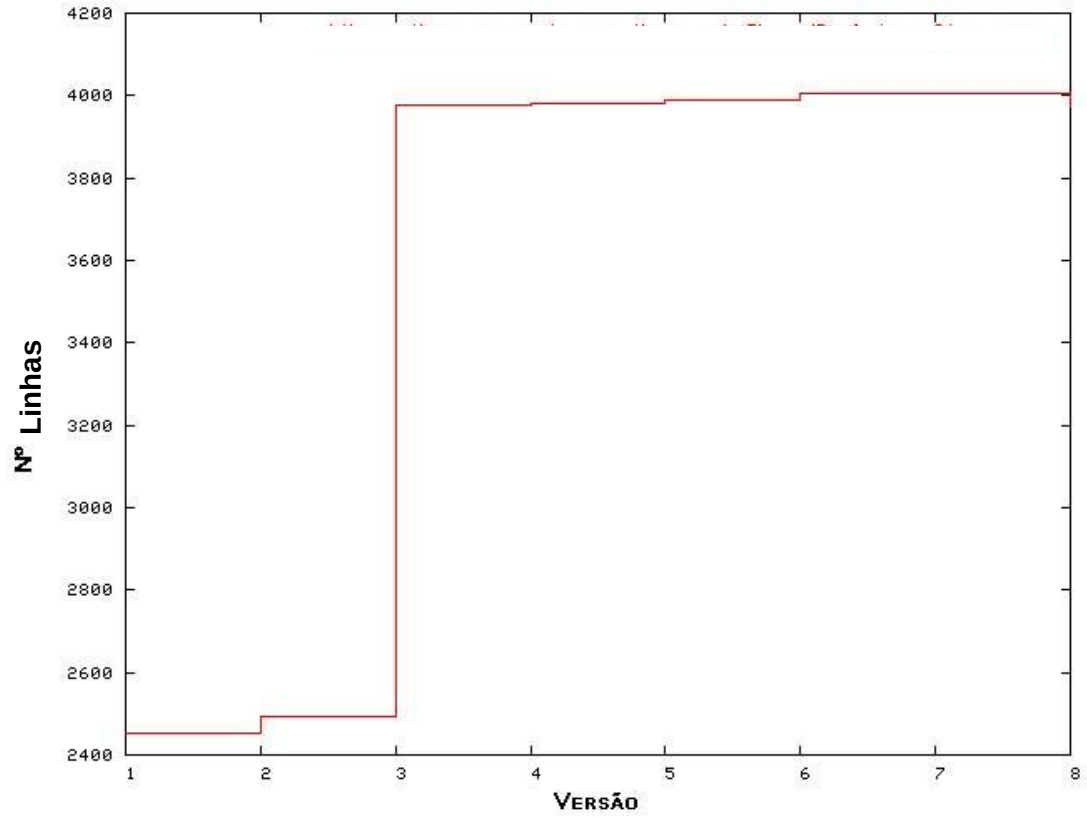
J2eeModuleMetaData.java



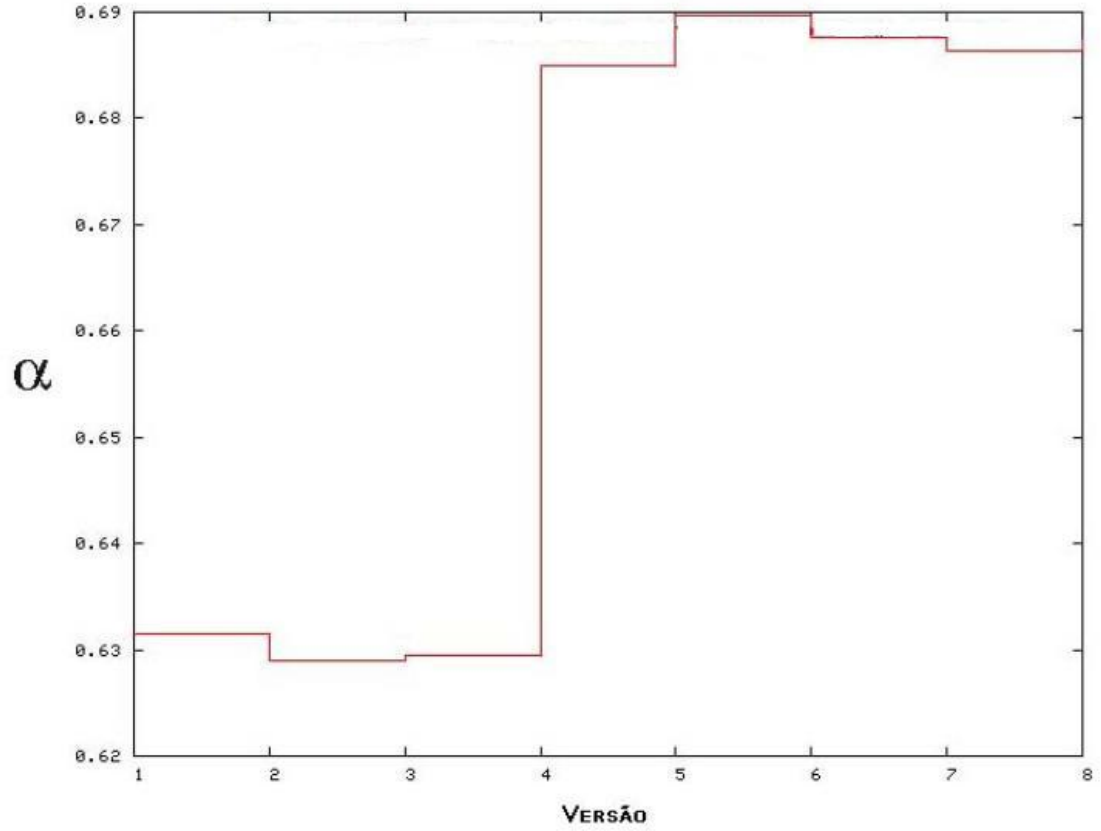


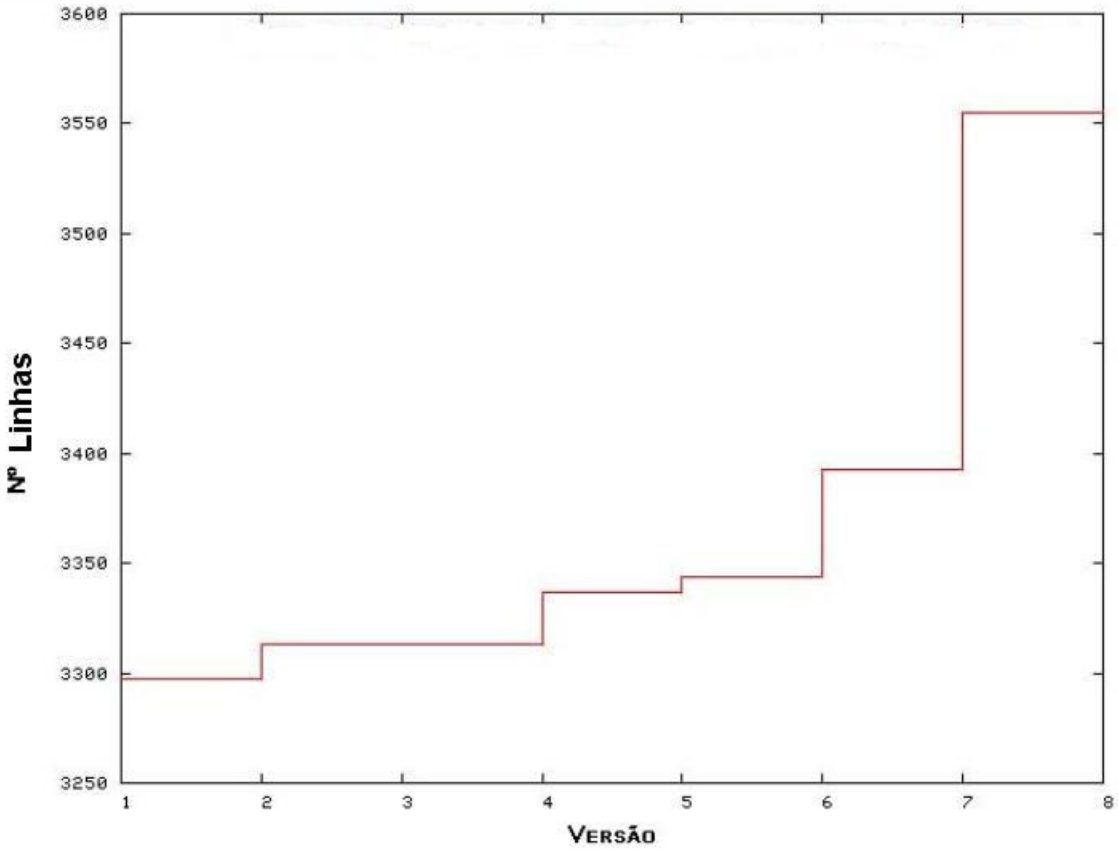
ThreadPool.java



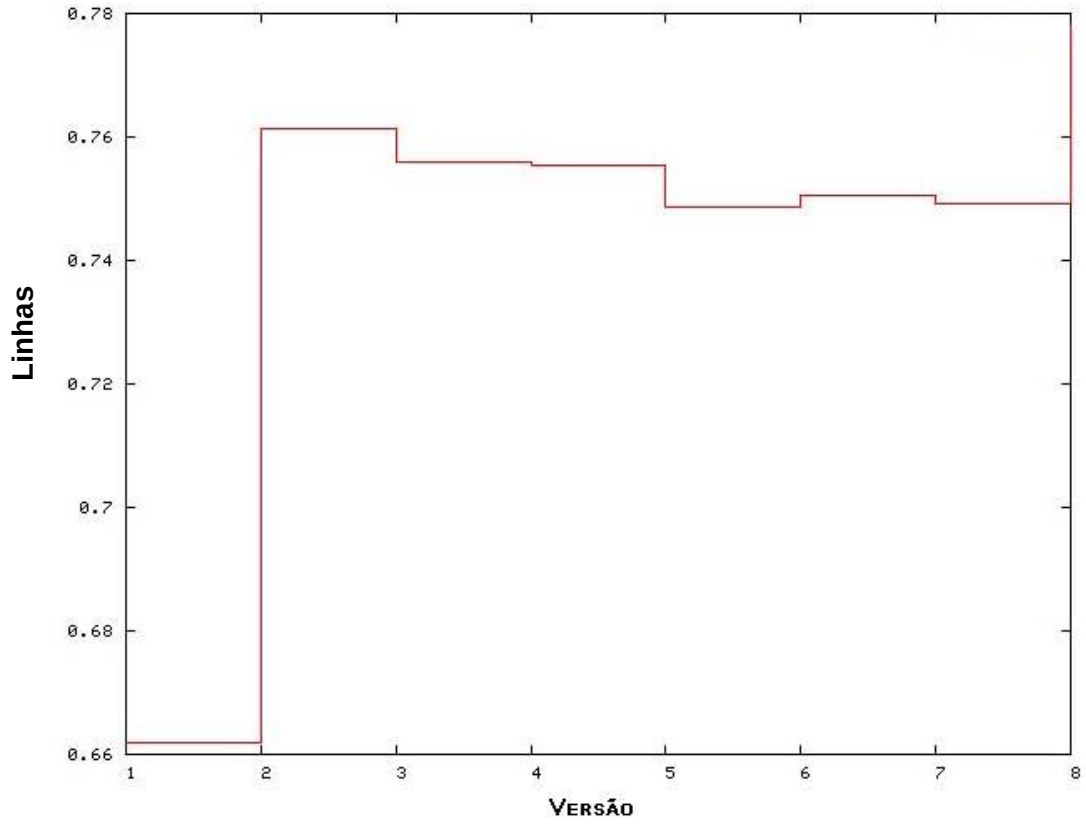


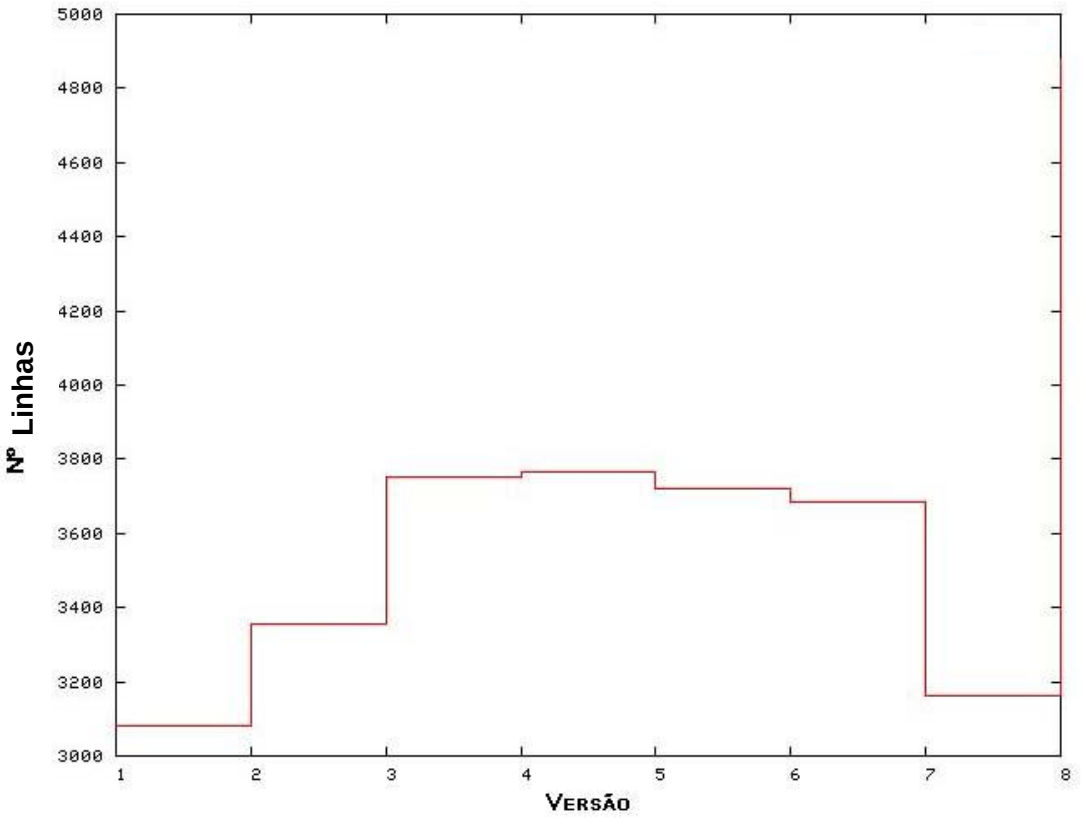
QueryMetaData.java



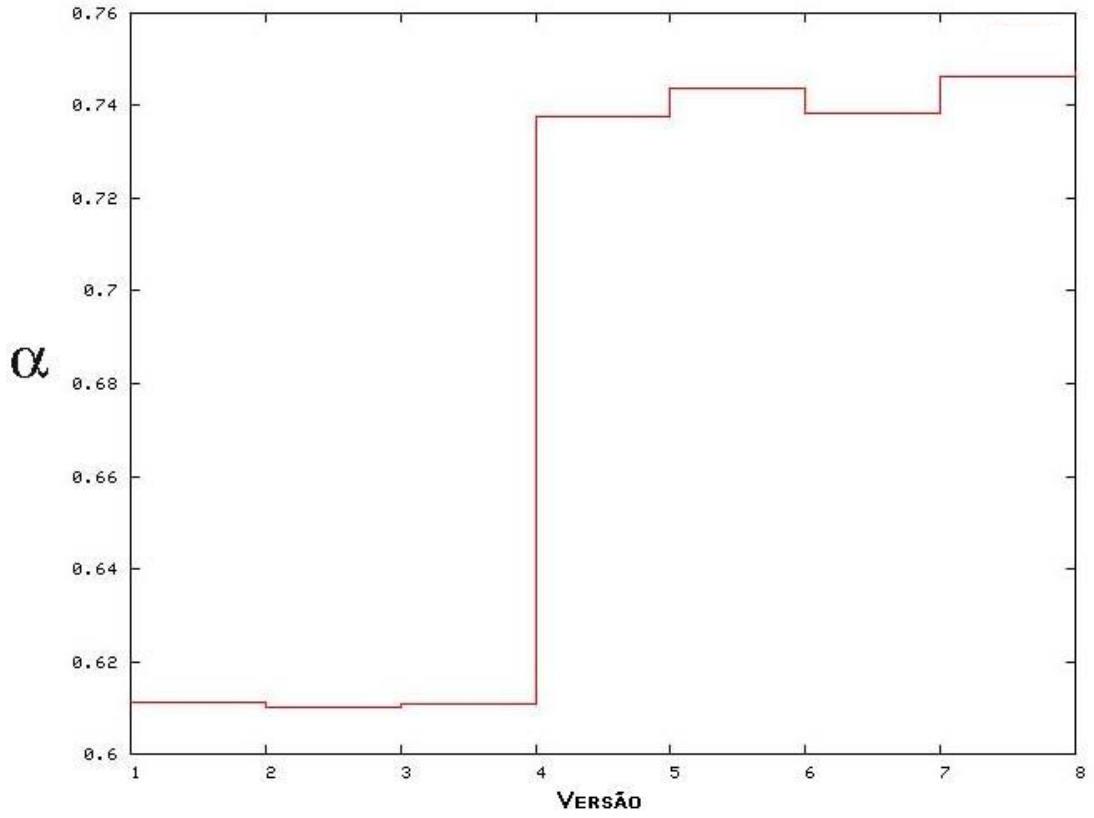


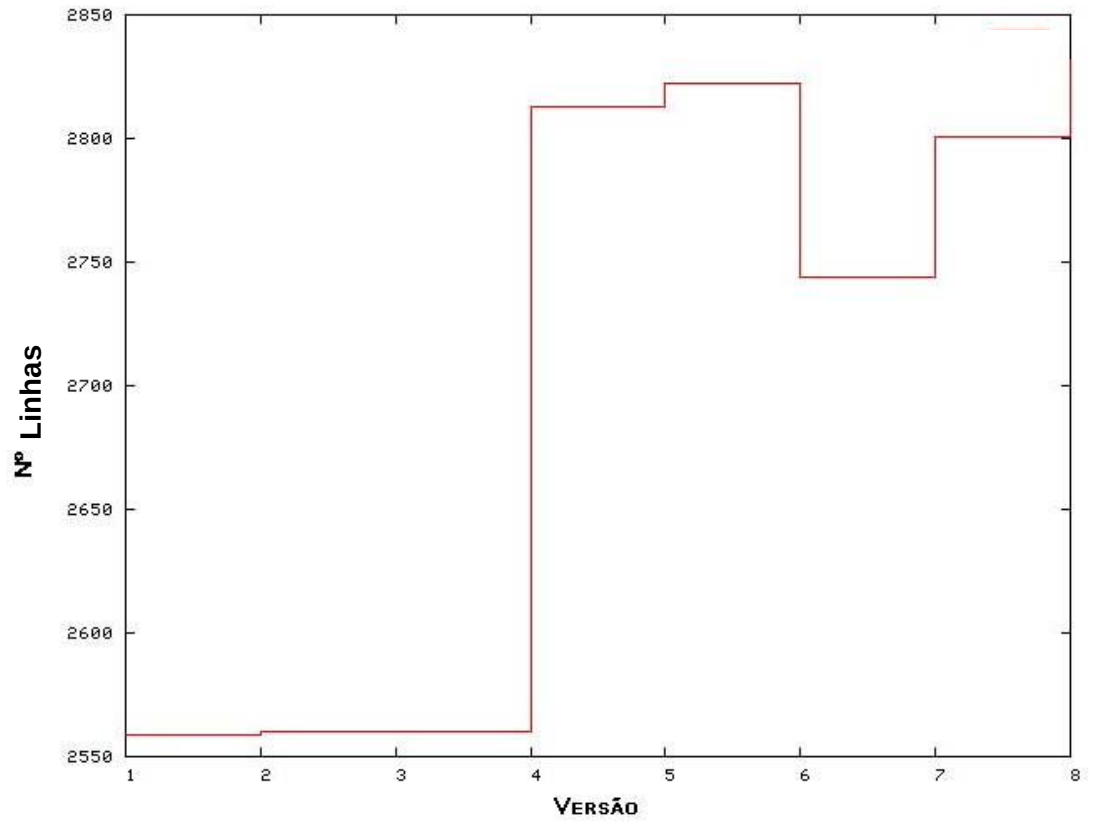
ConnectorFactoryService.java



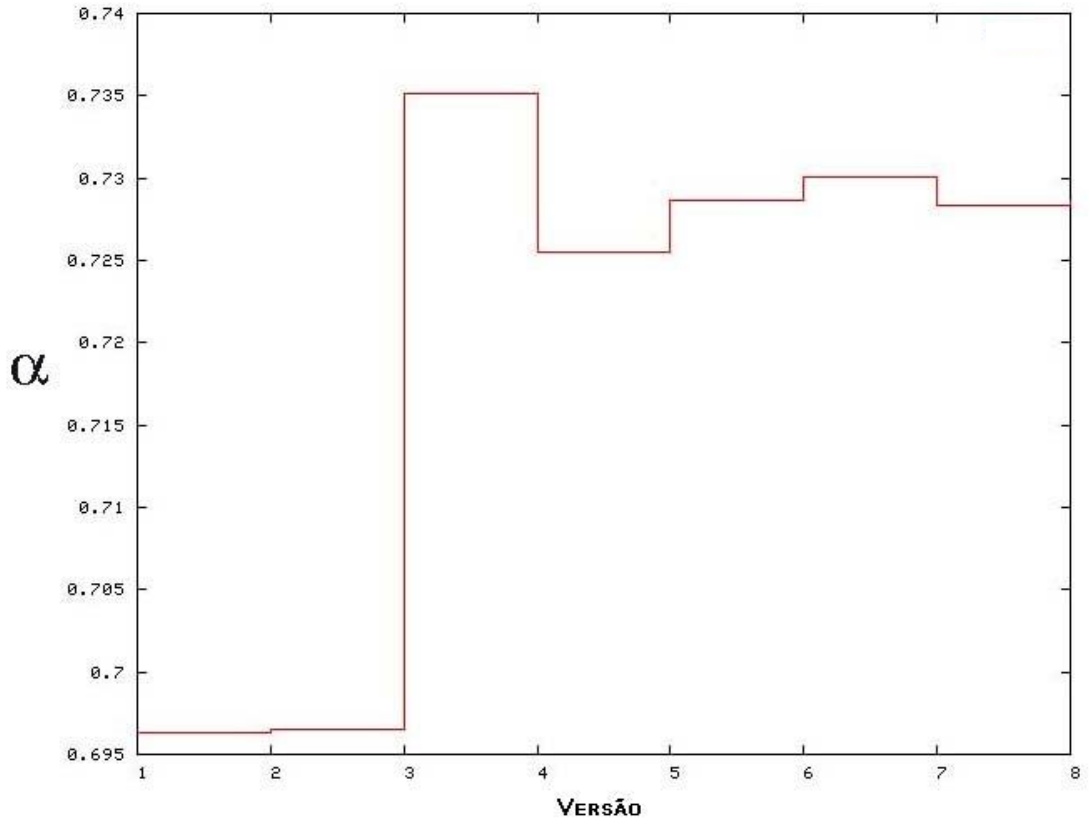


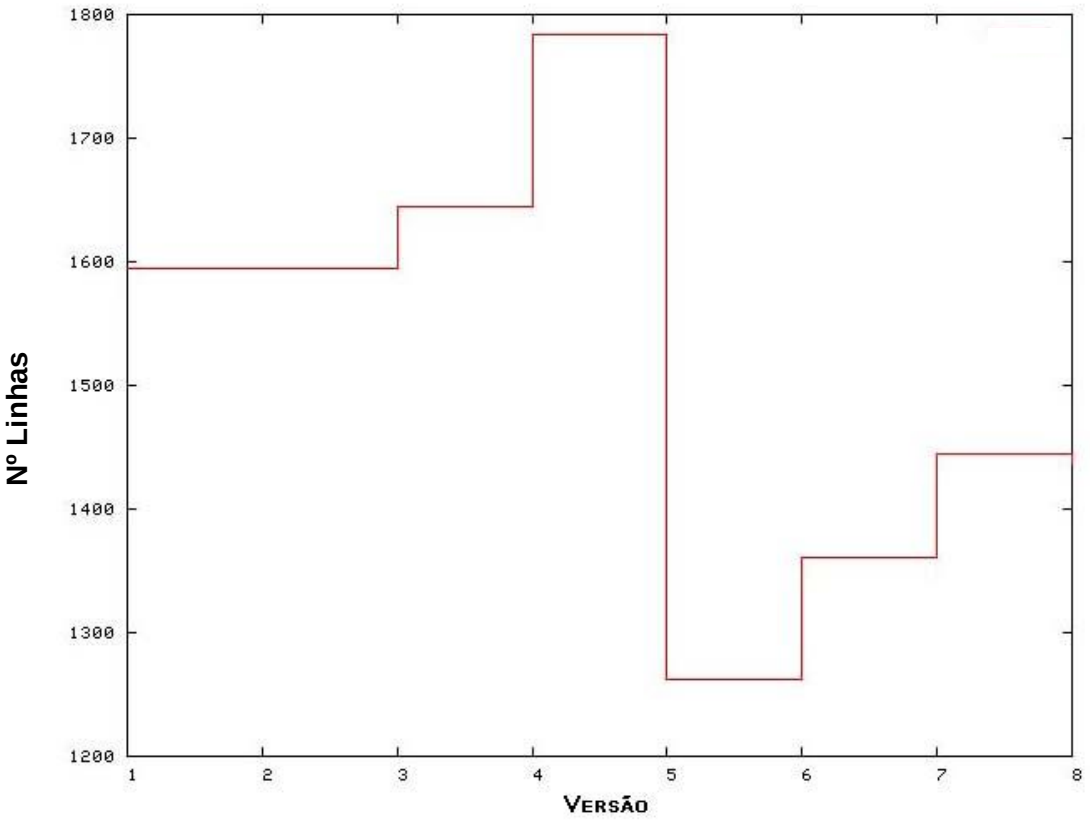
MappingMetaData.java



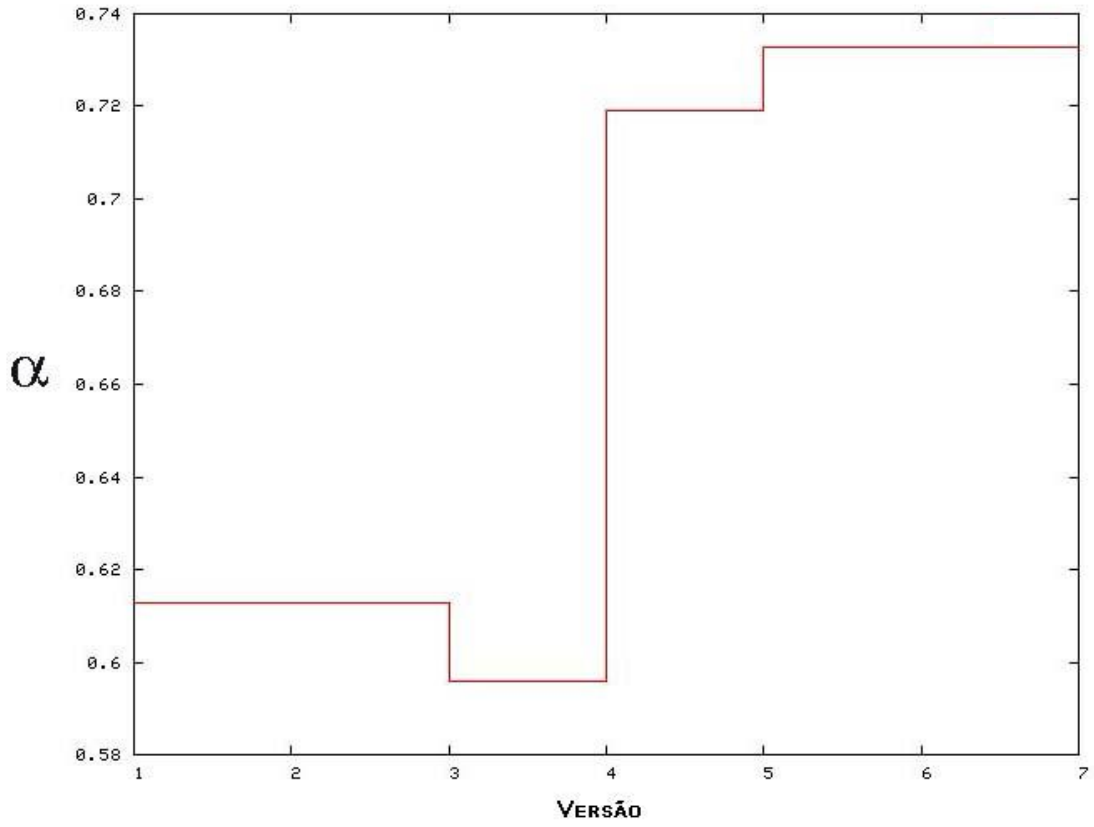


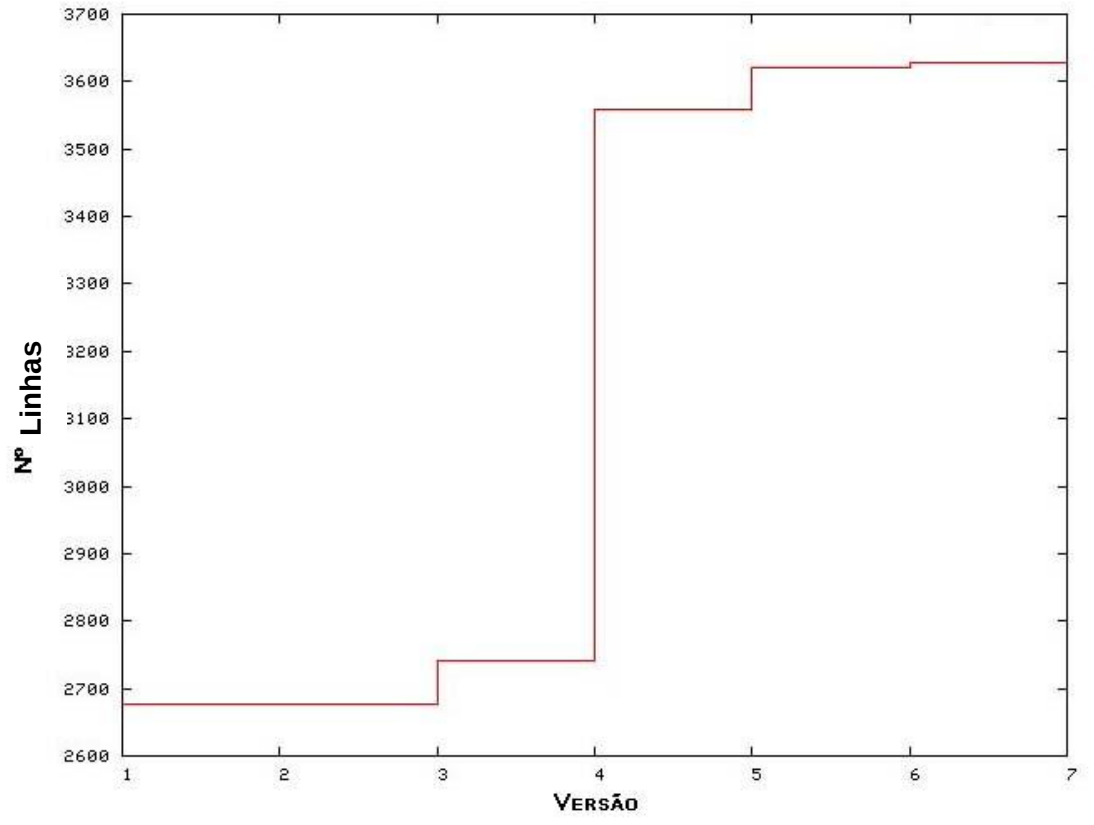
EntityBridge.java



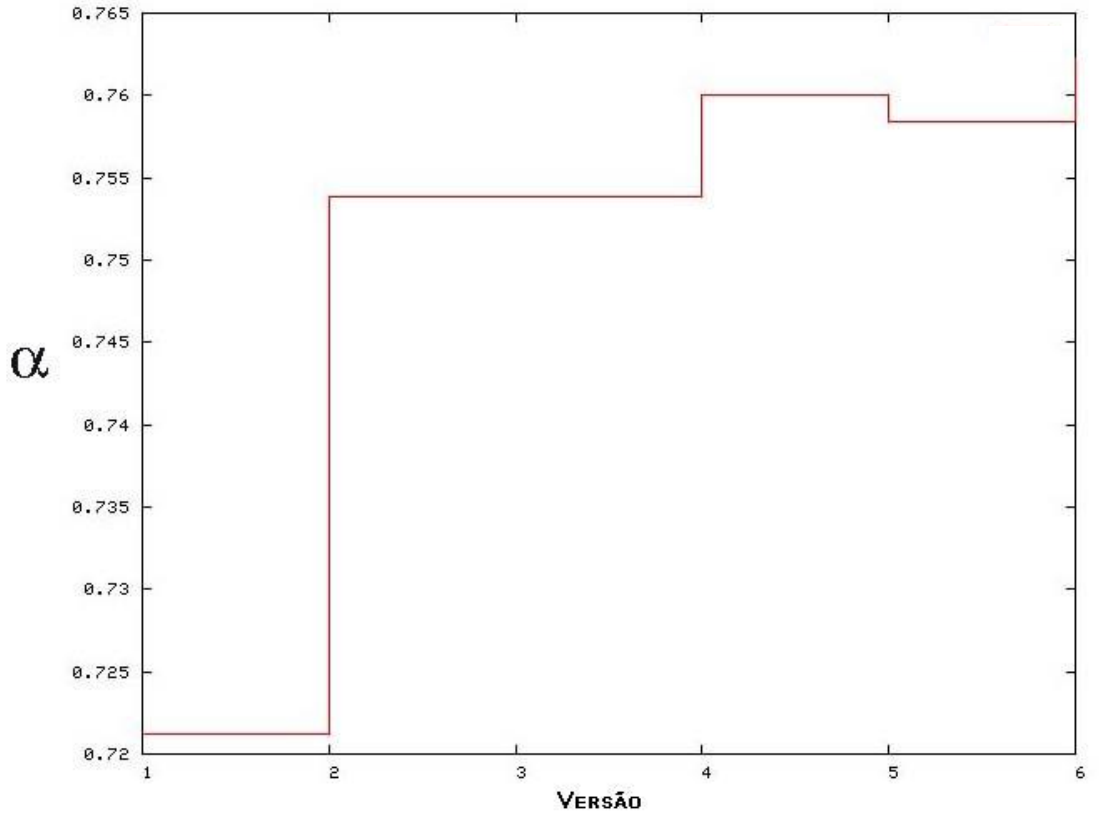


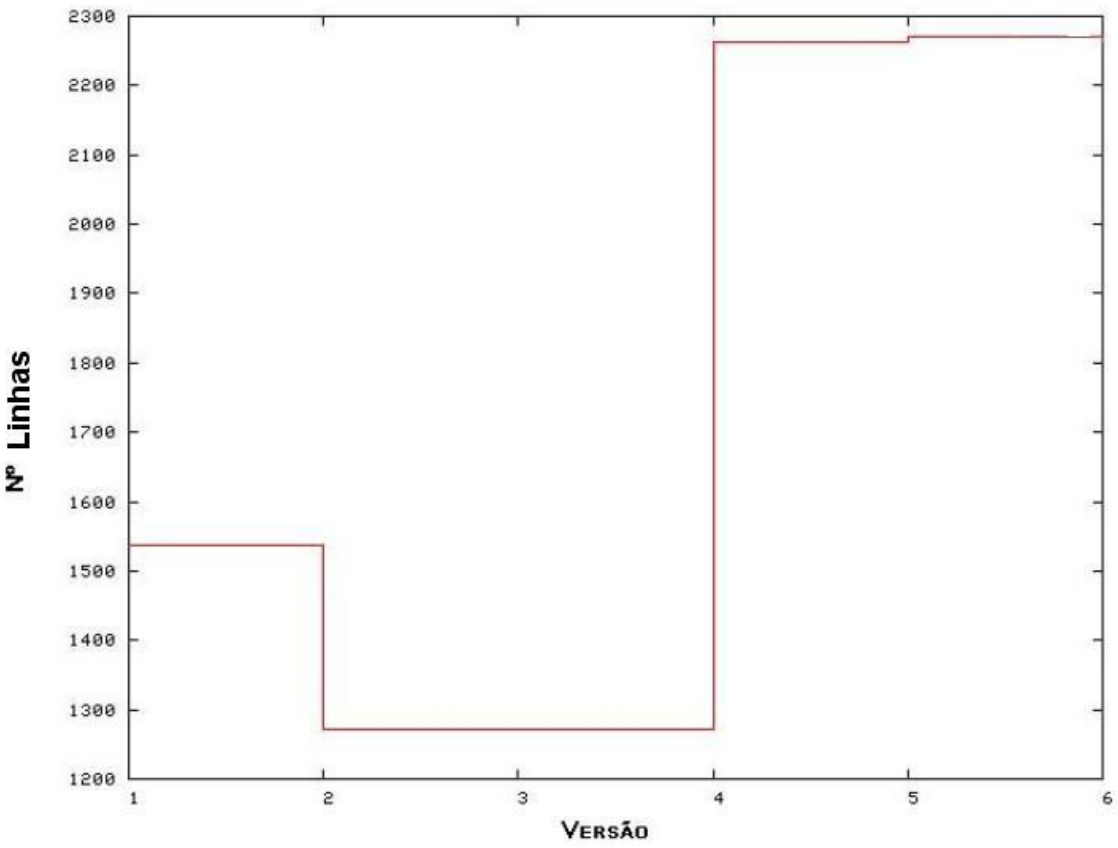
JDBCAutomaticQueryMetaData.java



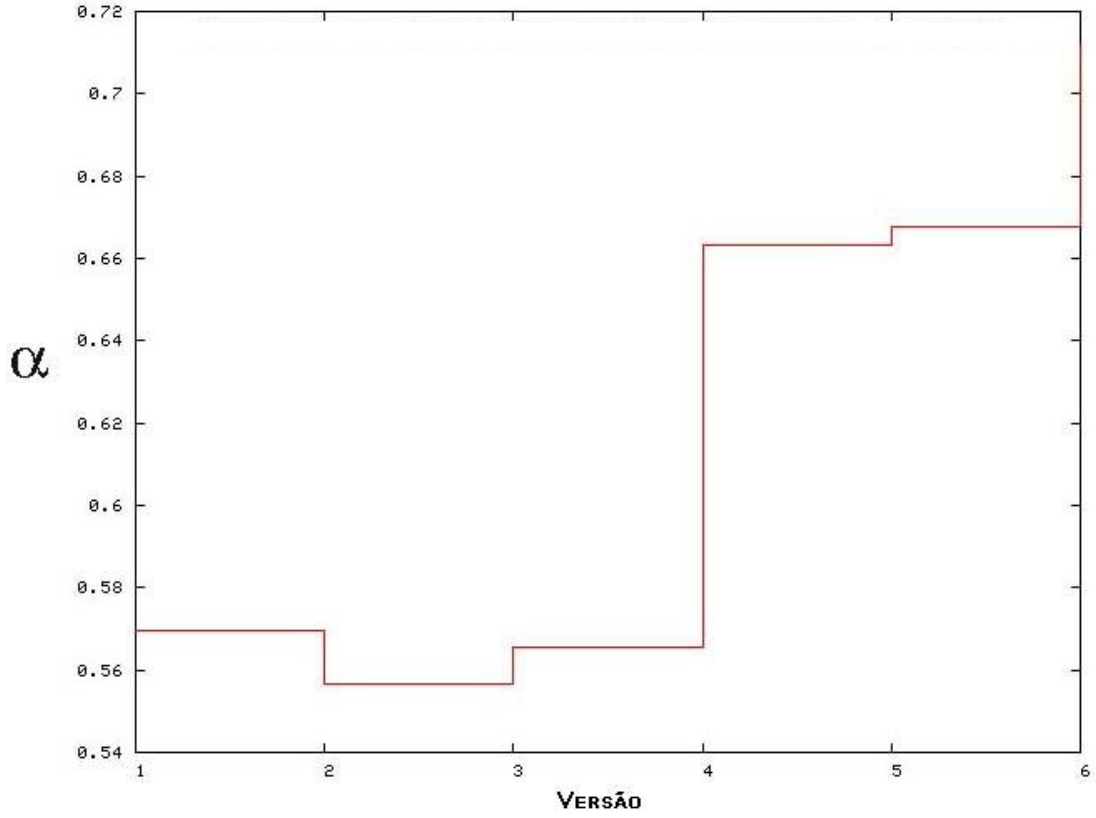


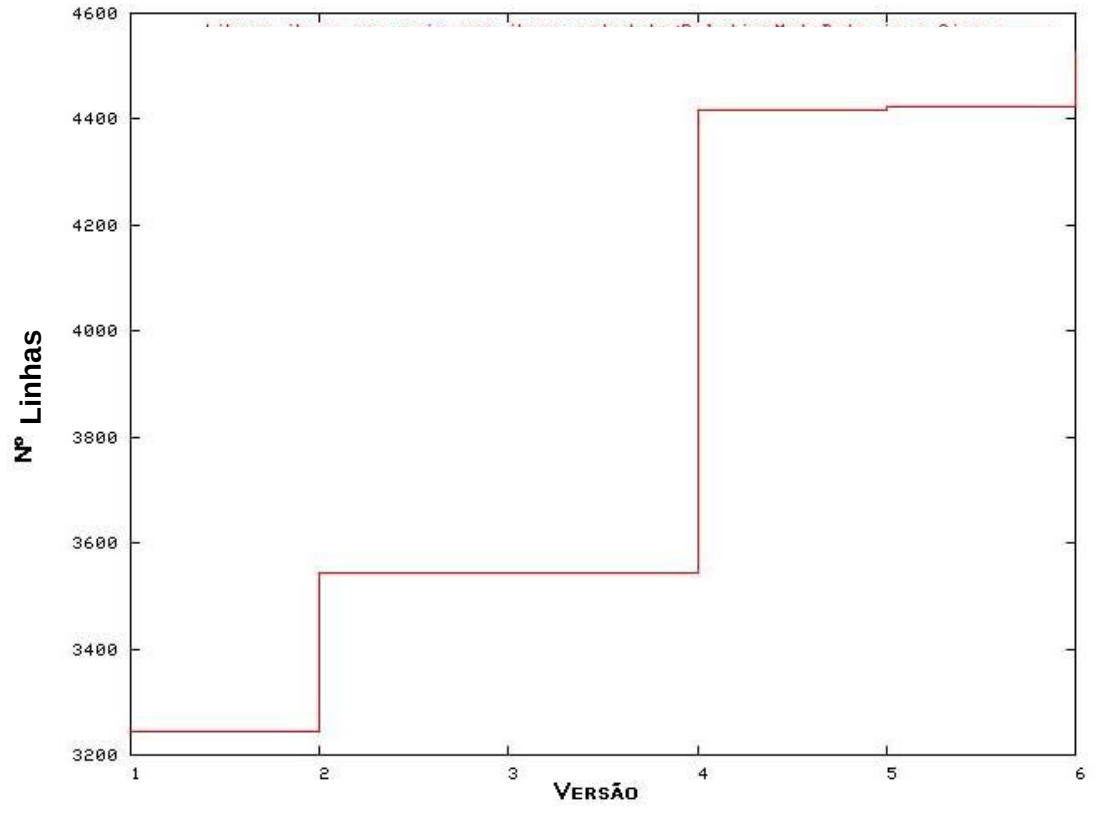
ClientUserTransactionObjectFactory.java



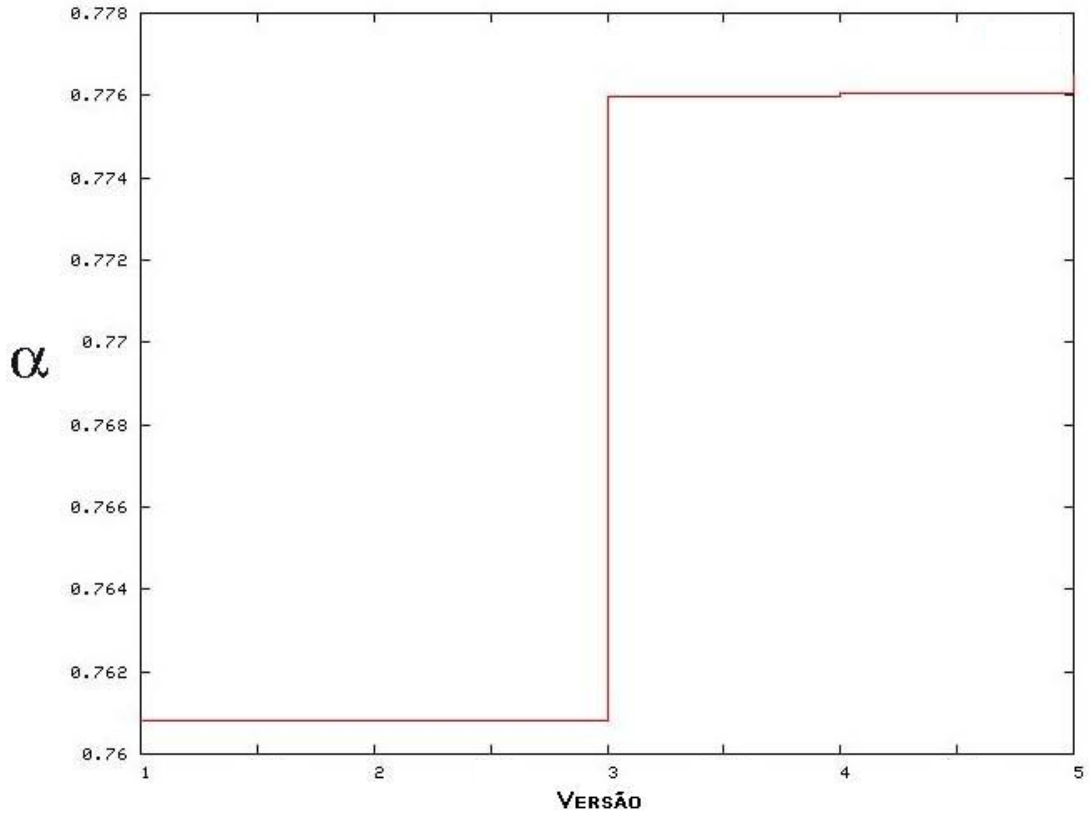


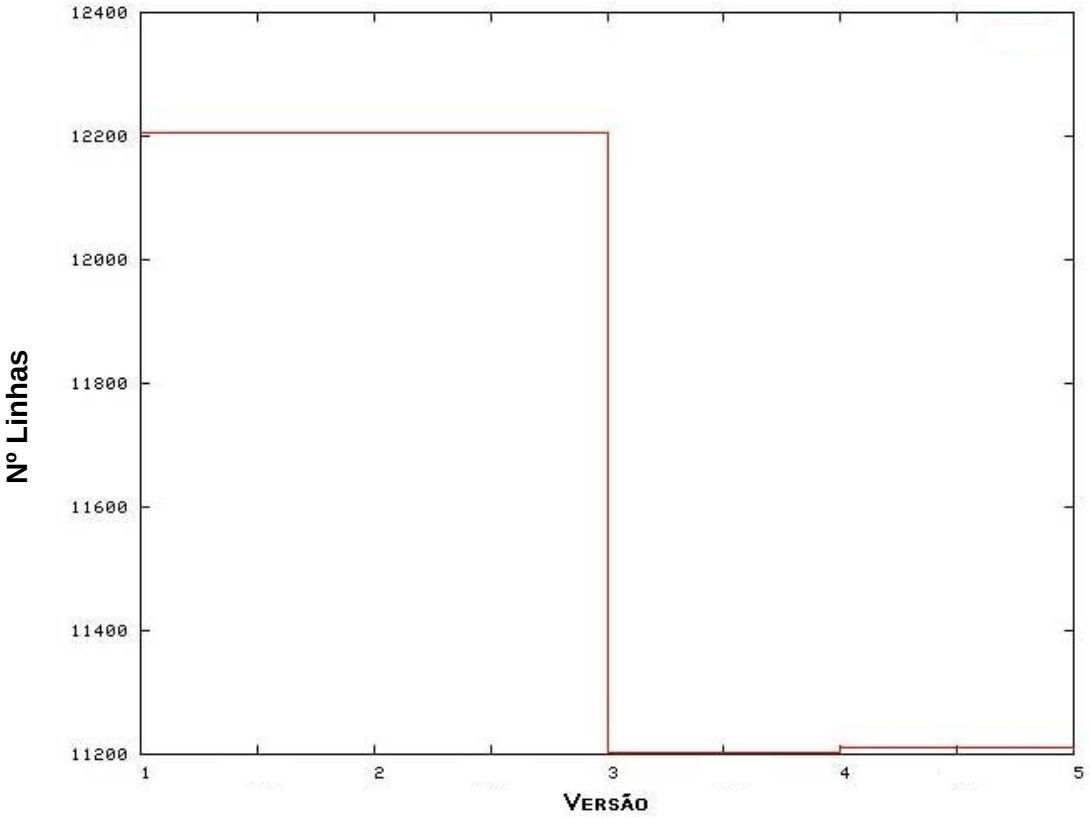
RelationMetaData.java



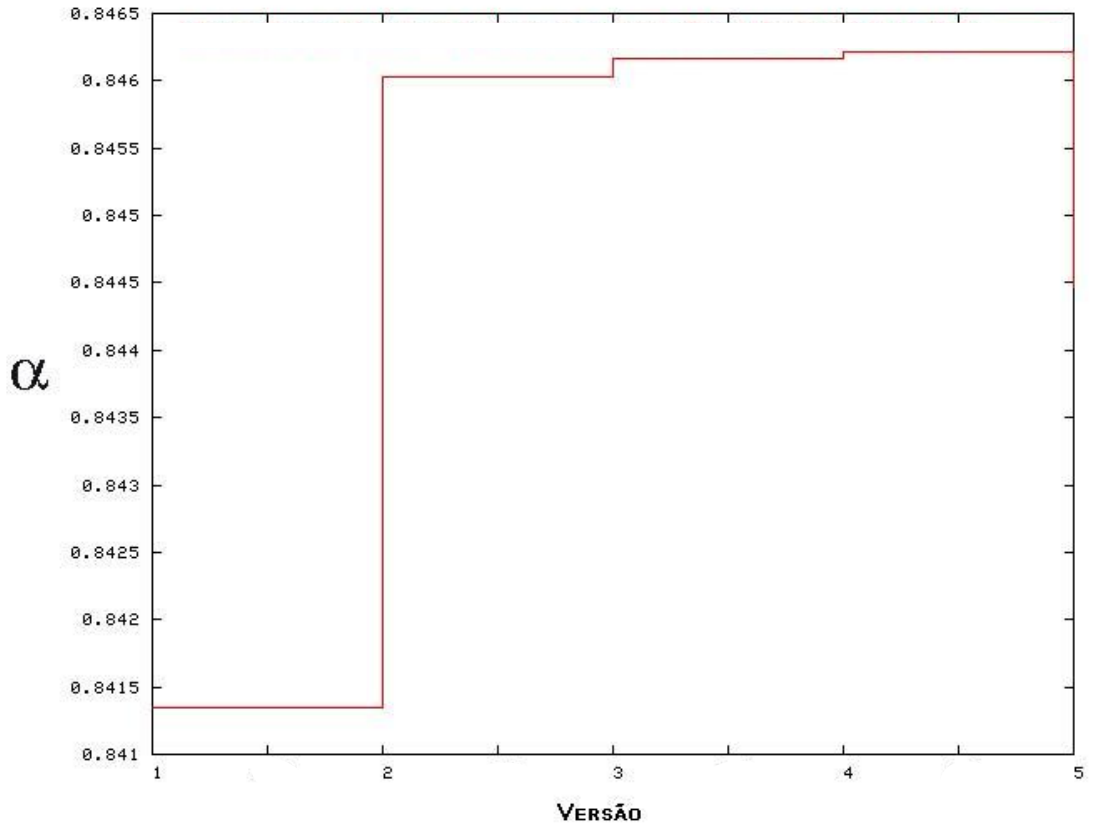


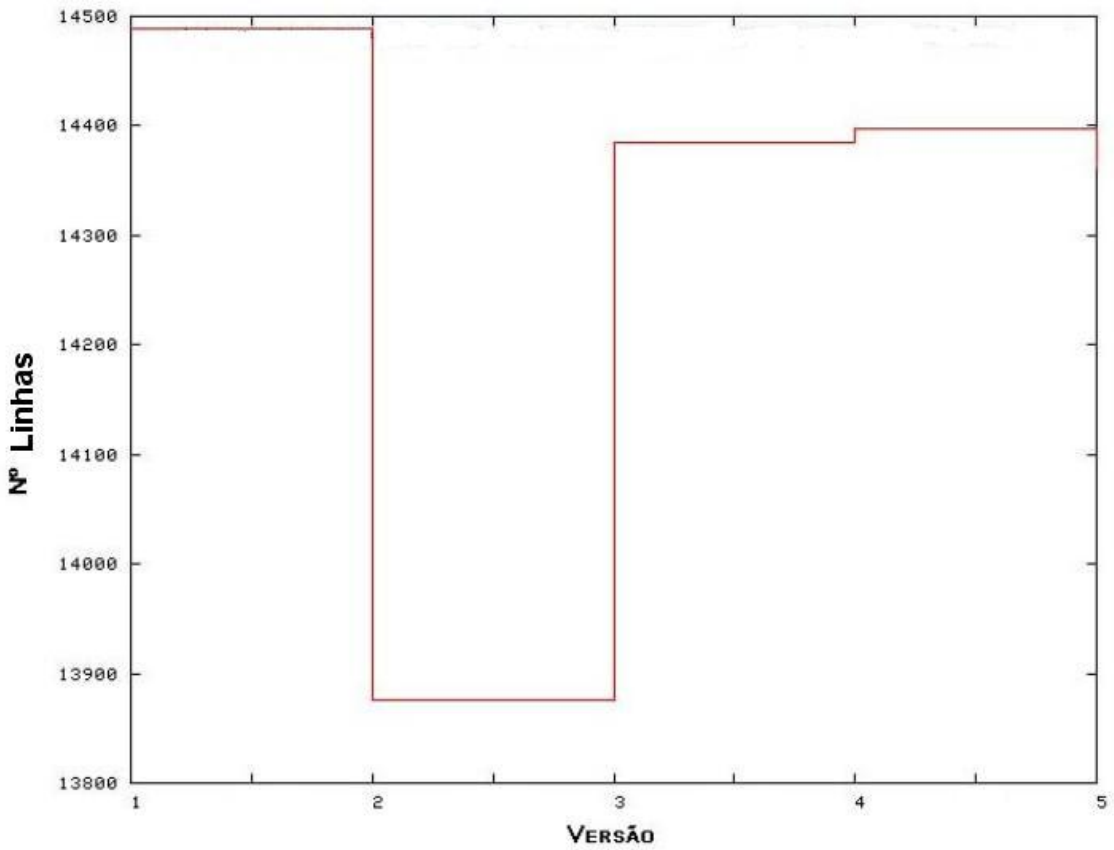
ClientUserTransaction. Java



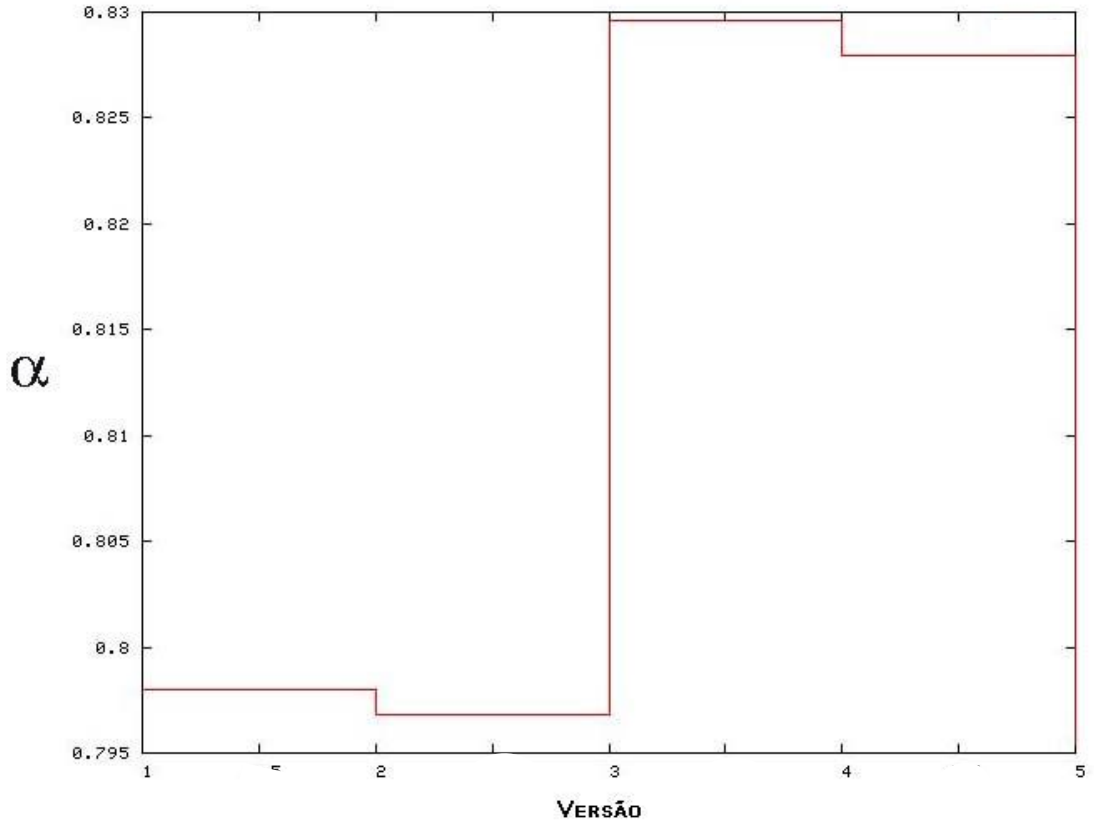


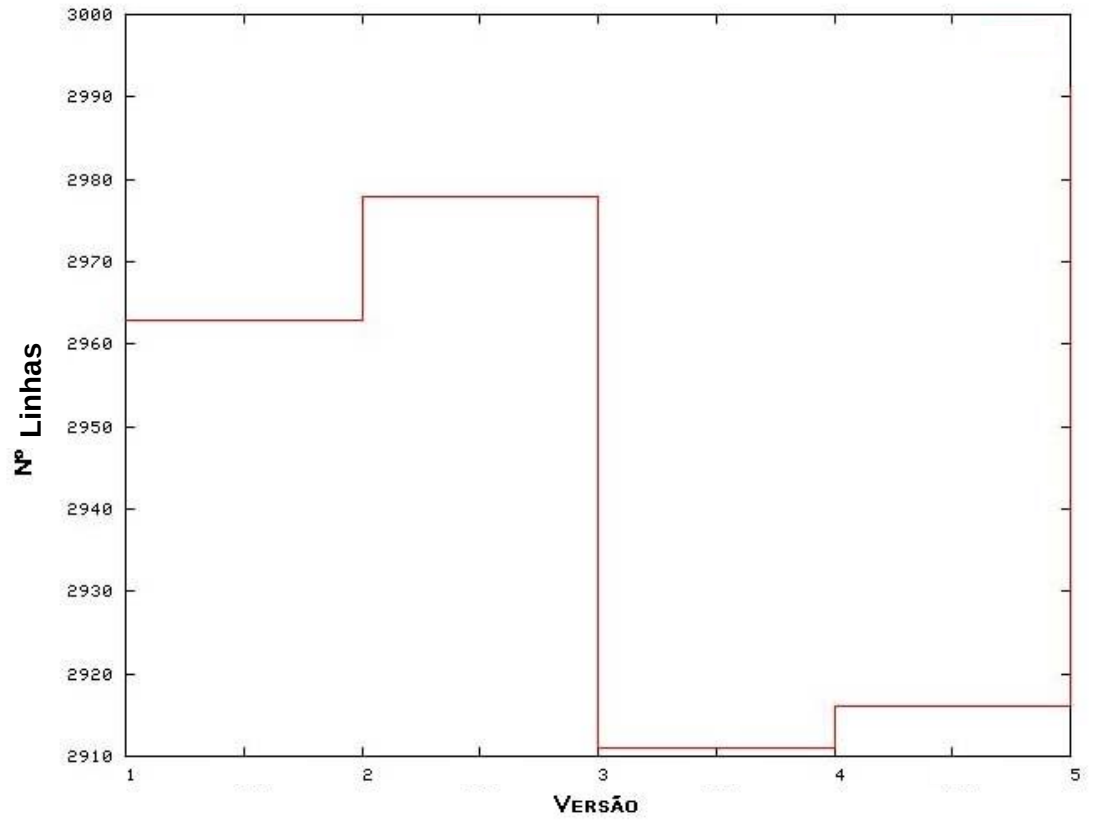
RemoteDeployer.java



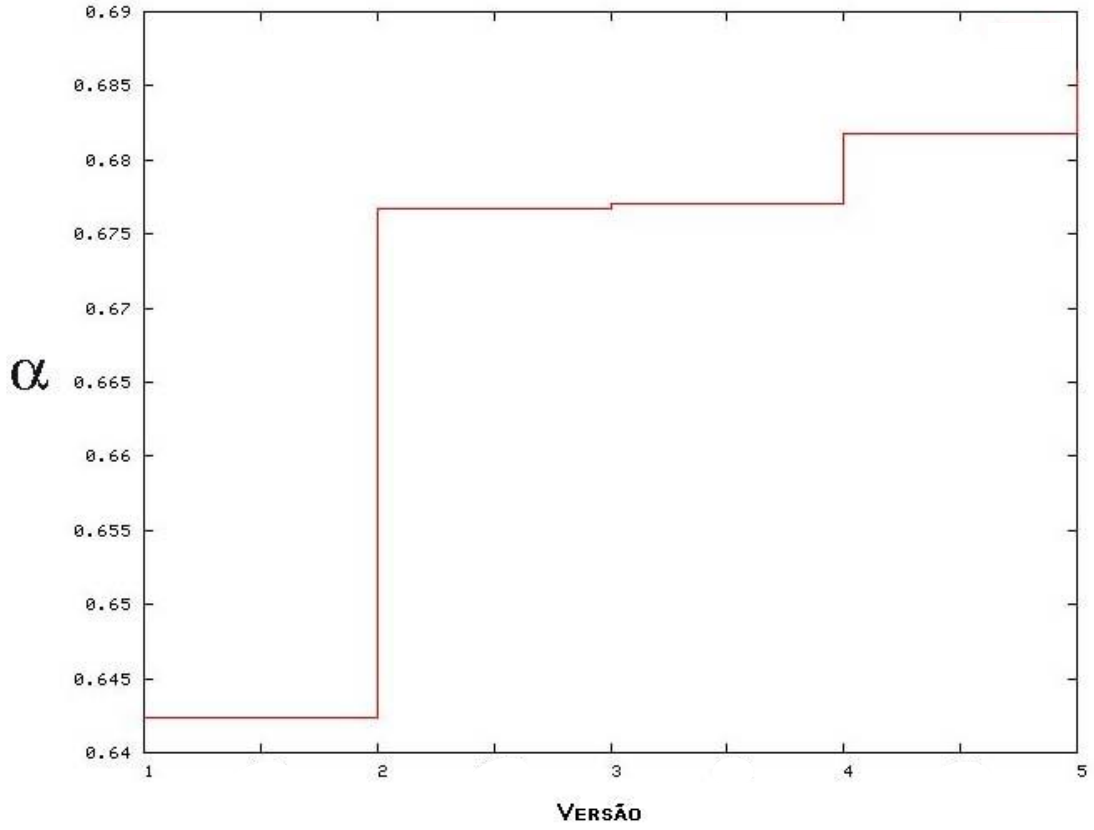


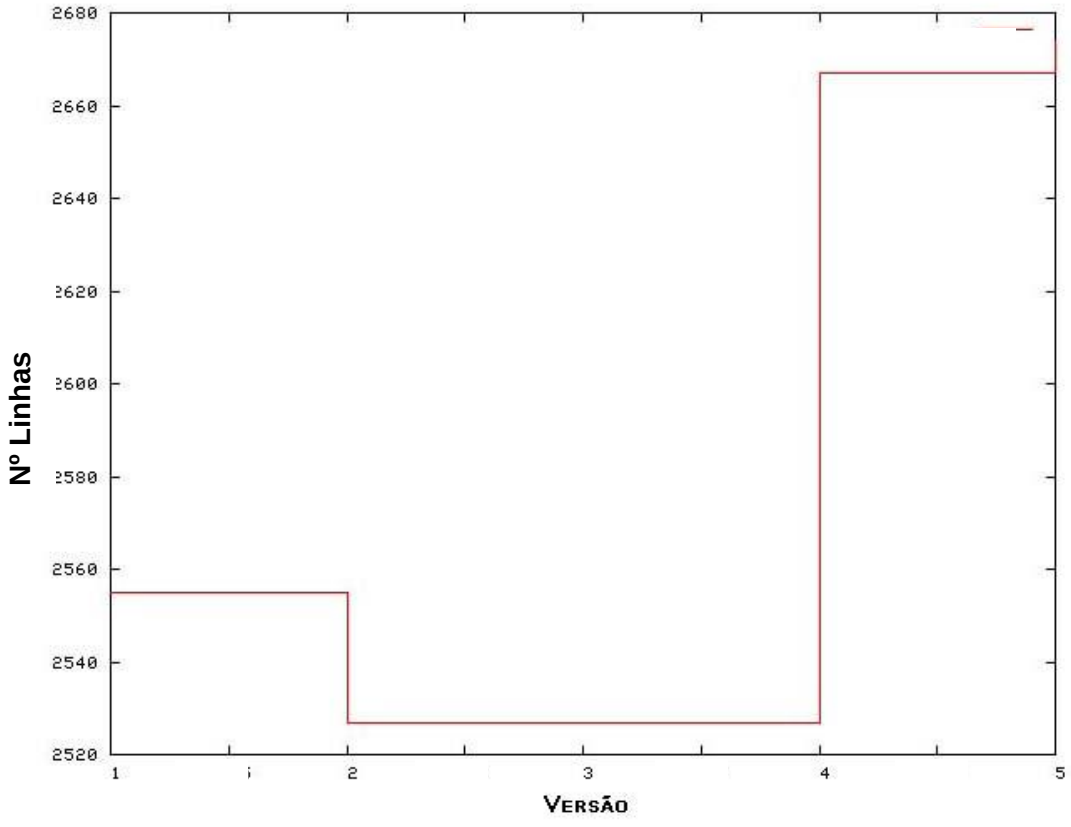
PollingClientNotificationListener.java



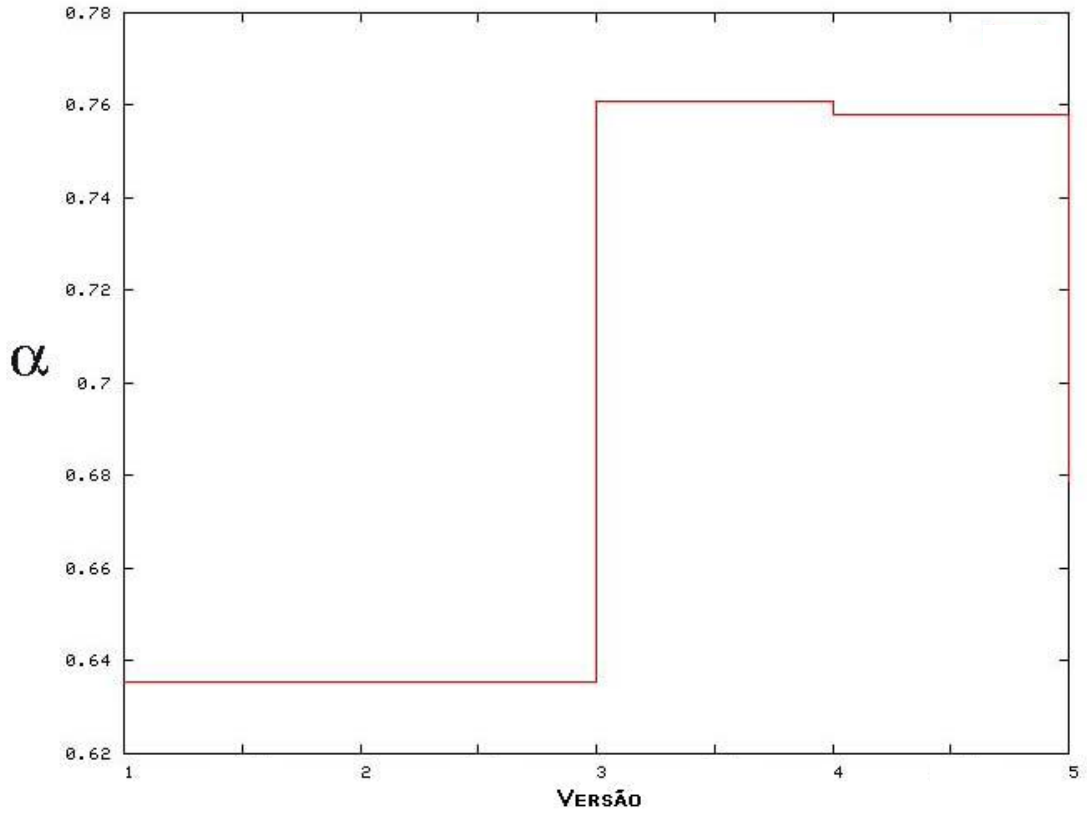


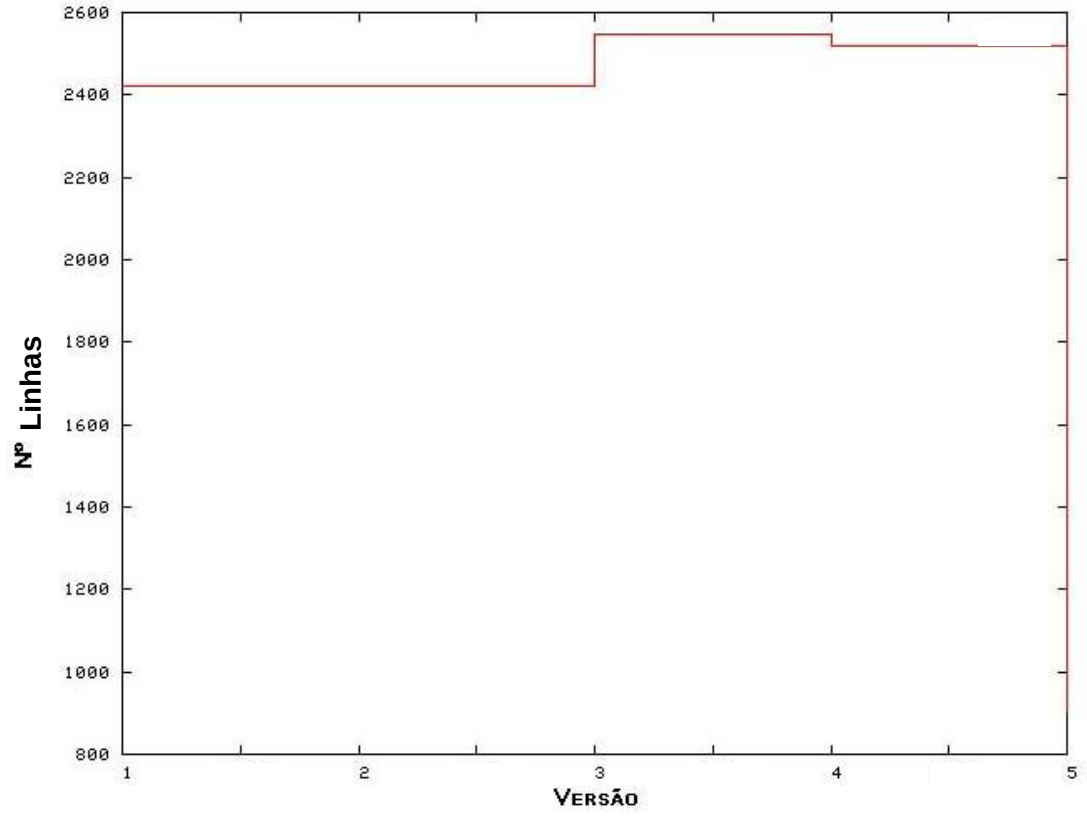
JDBCValueClassMetaData.java



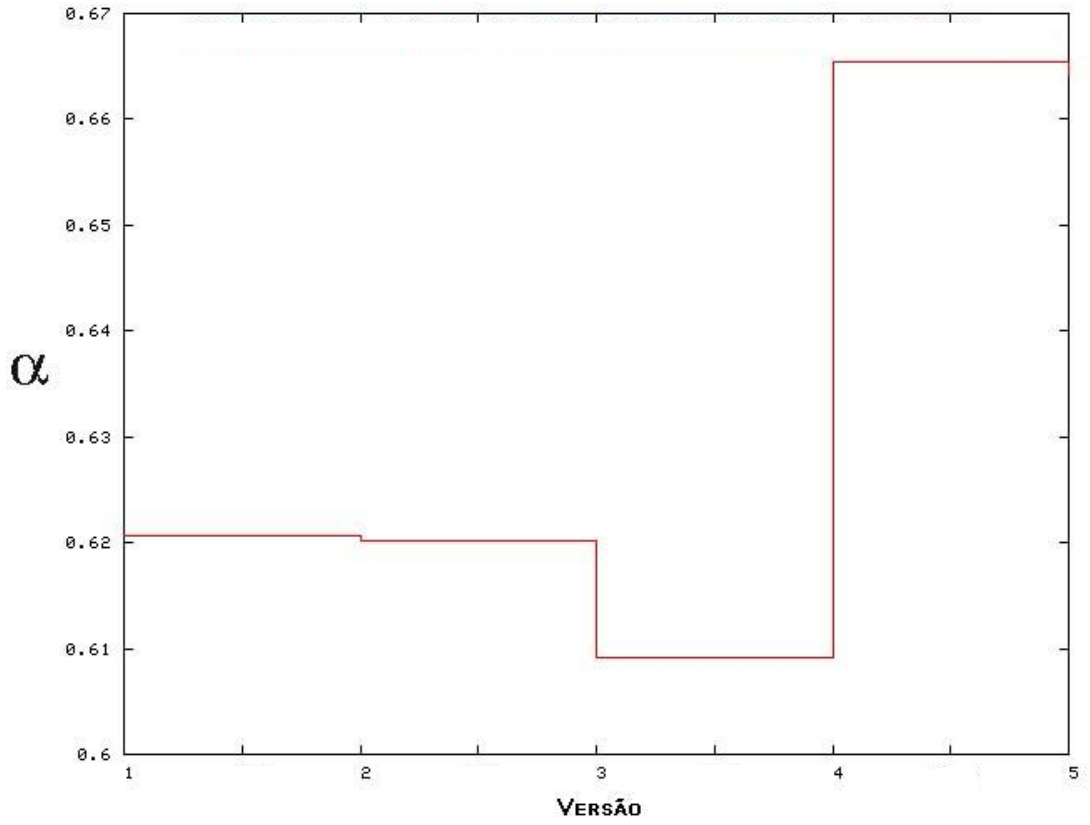


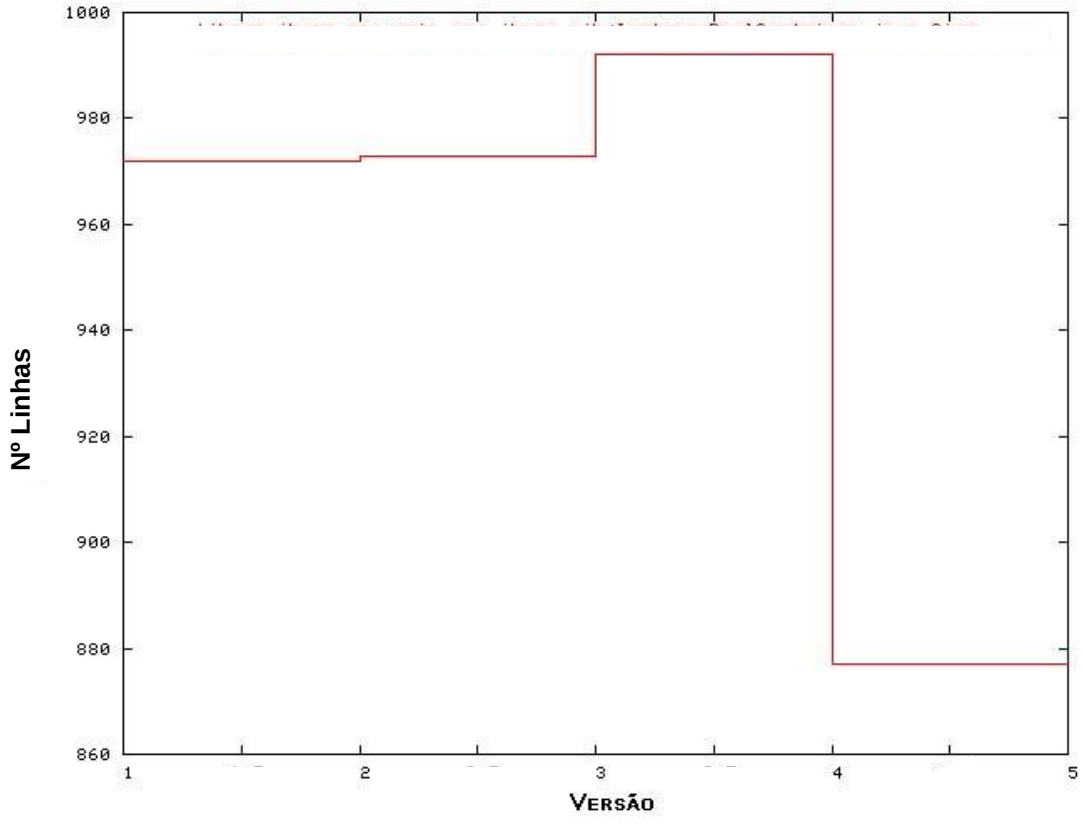
JDBCQueryCommand.java



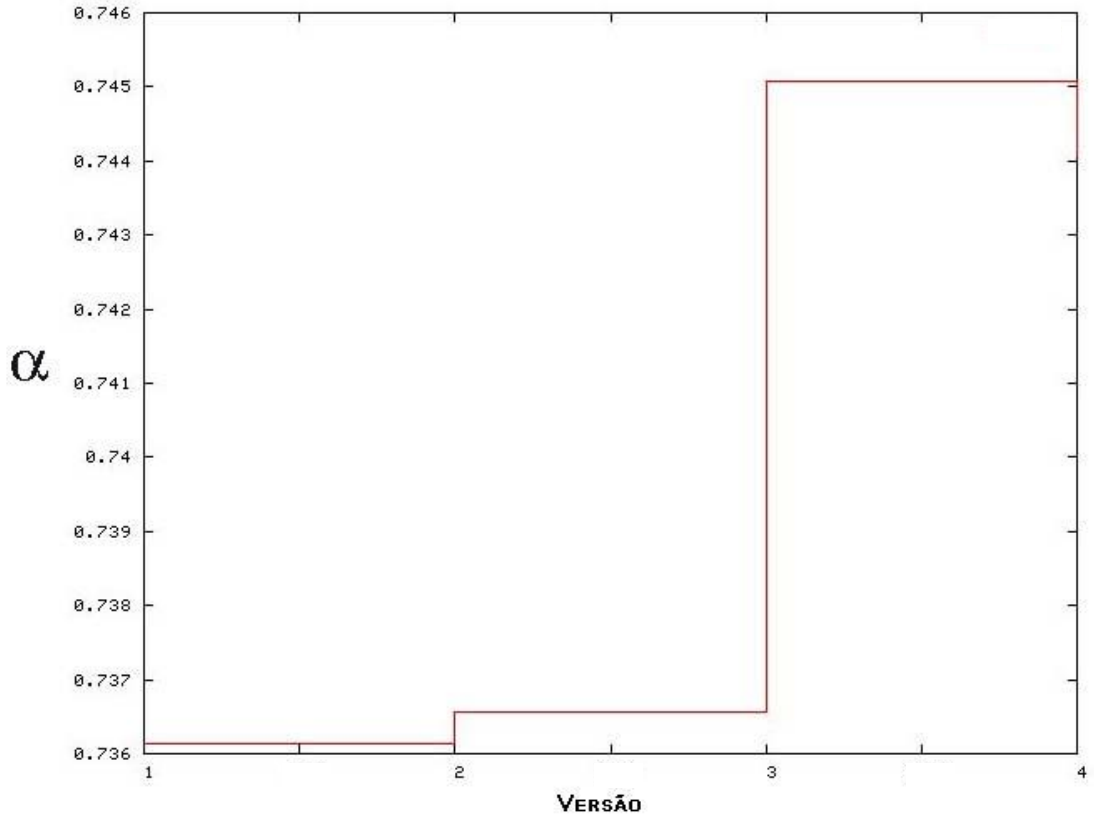


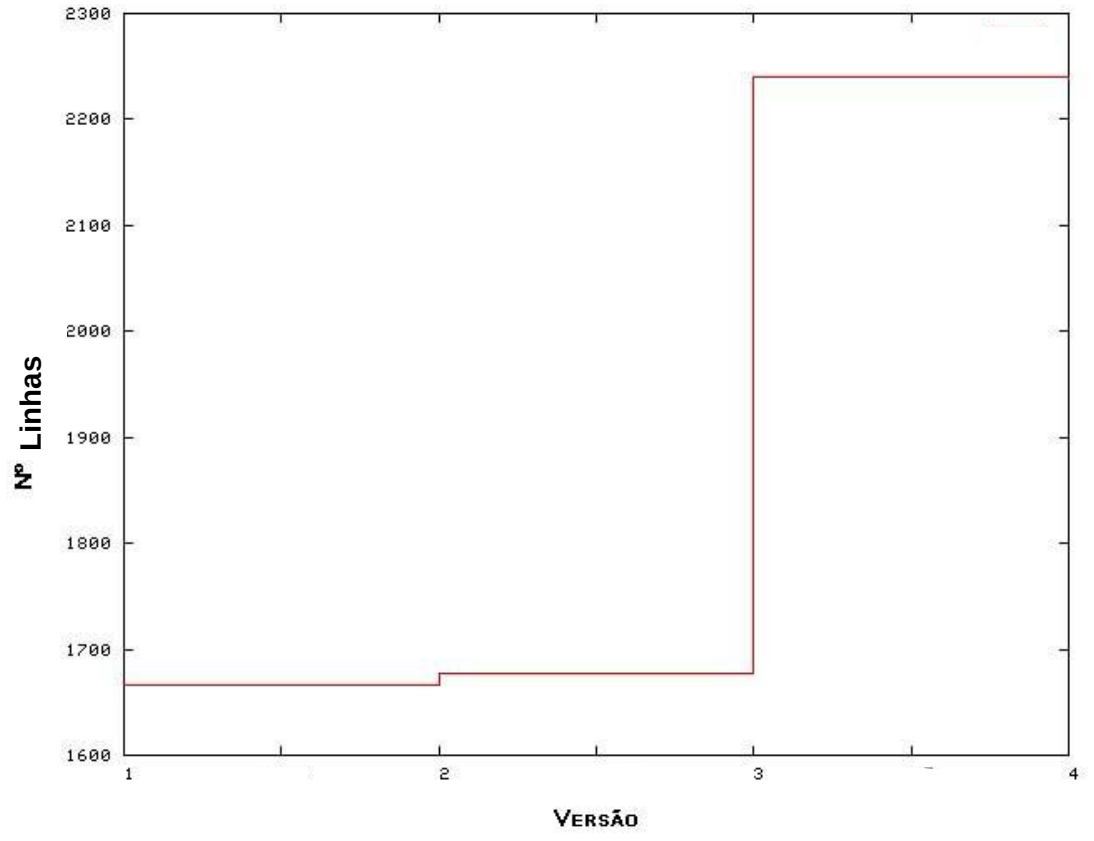
InstancePoolContainer.java



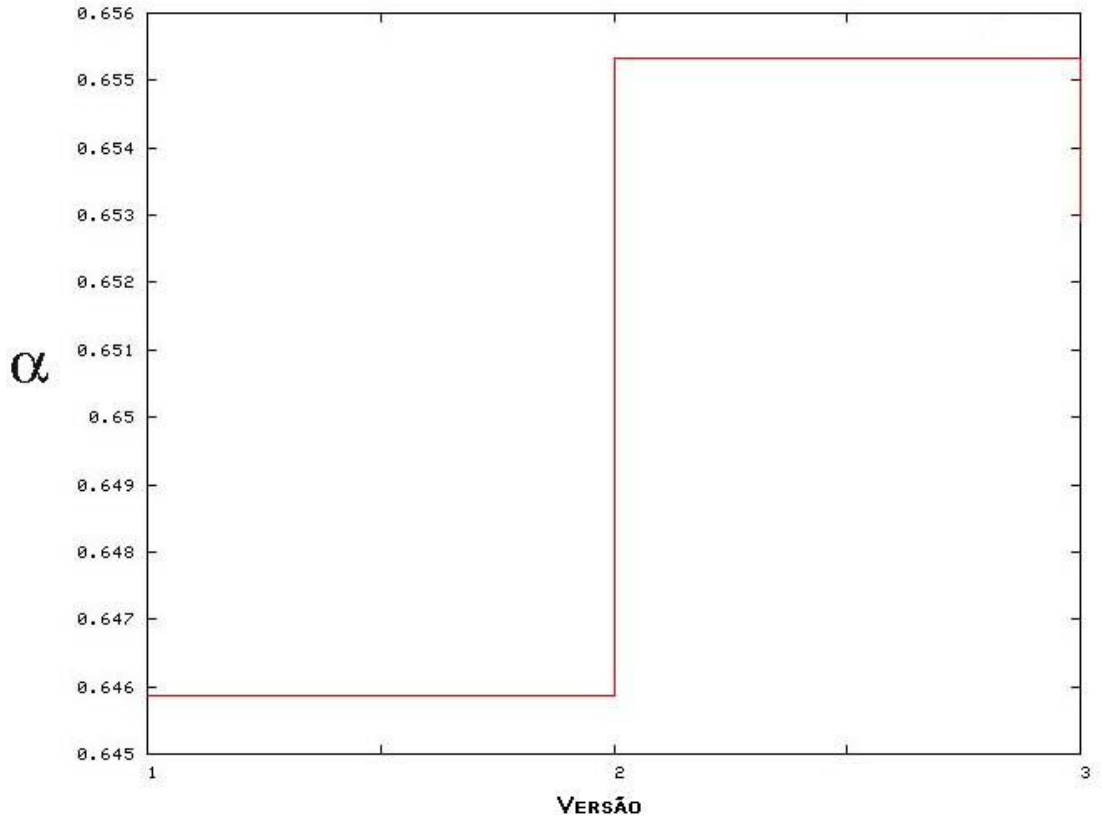


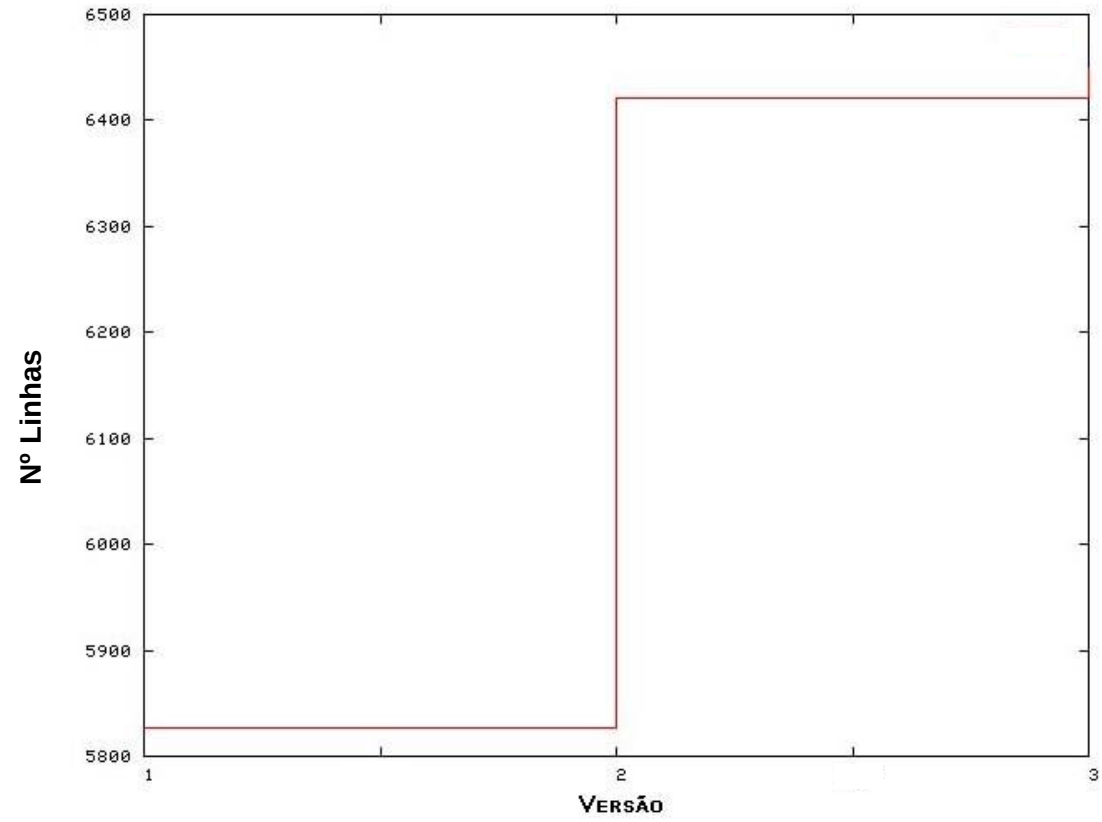
JDBCFindByPrimaryKeyQuery.java



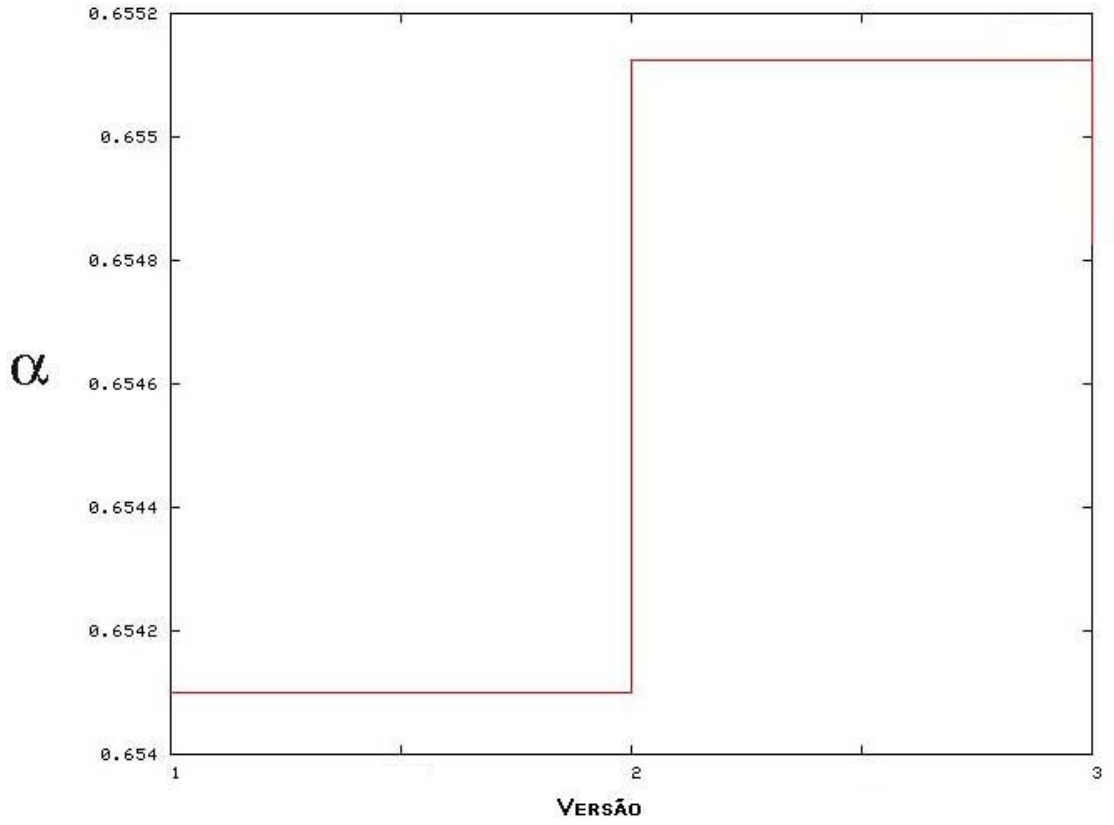


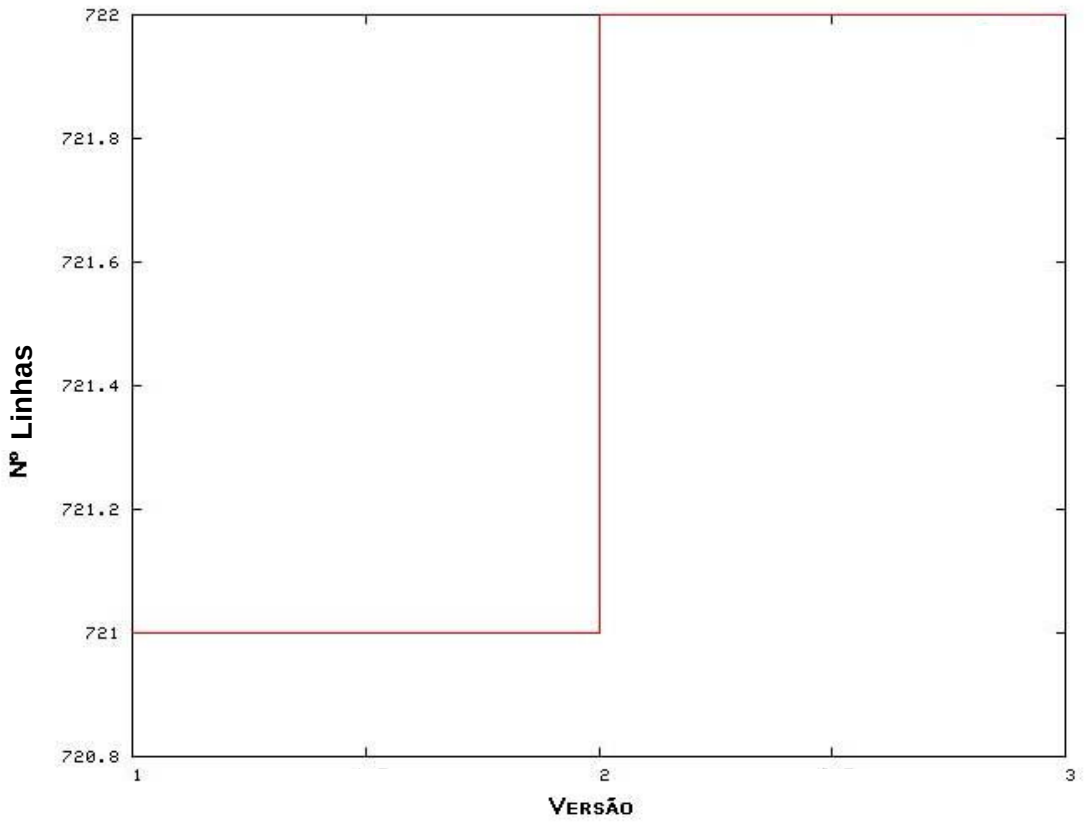
AbstractClient.java



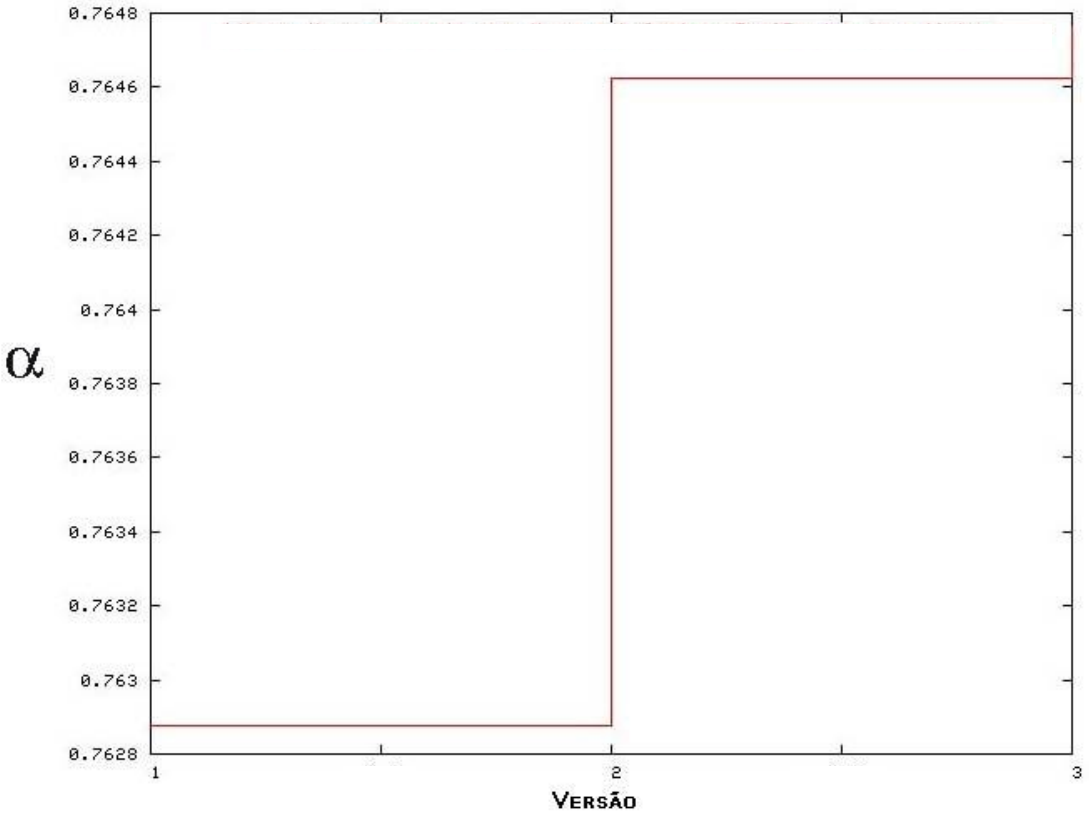


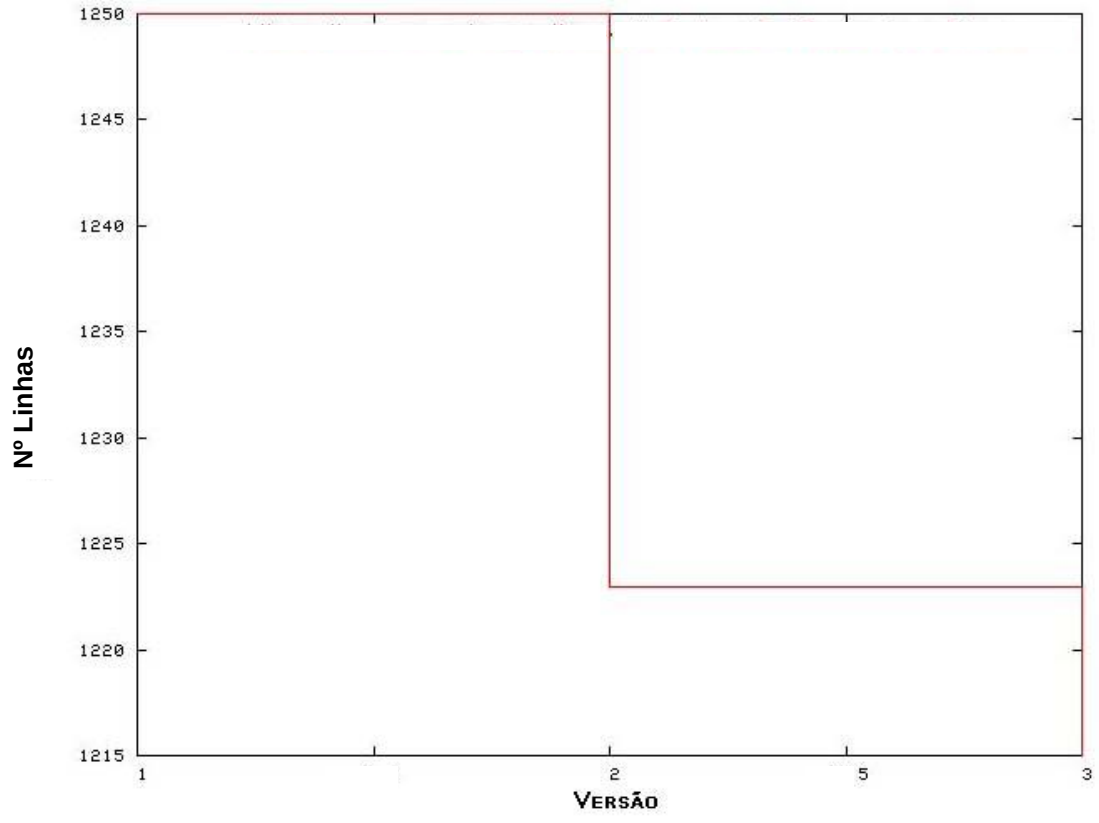
JPMStopCommand.java





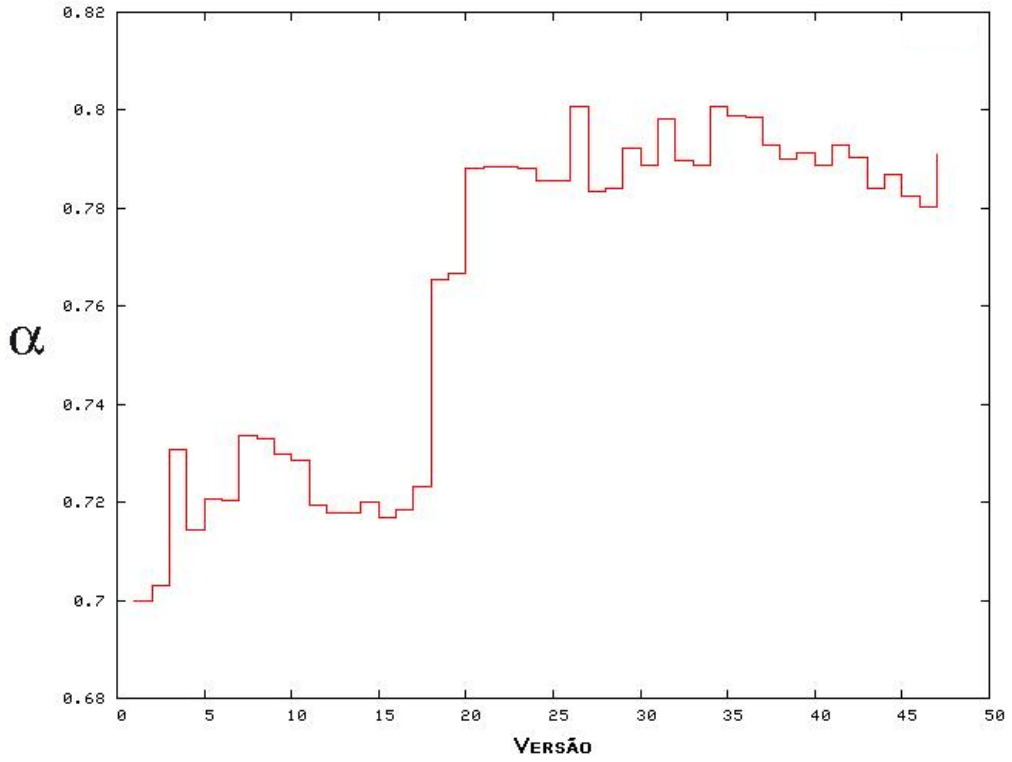
InstancePoolFeeder.java

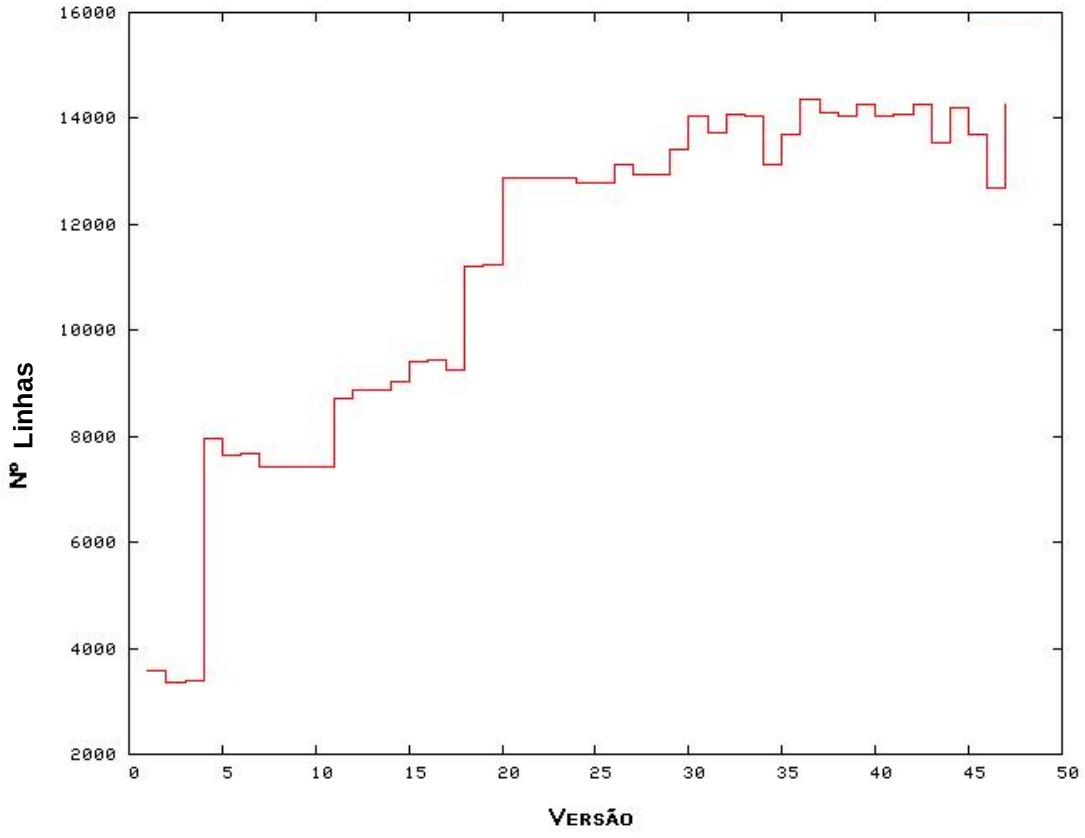




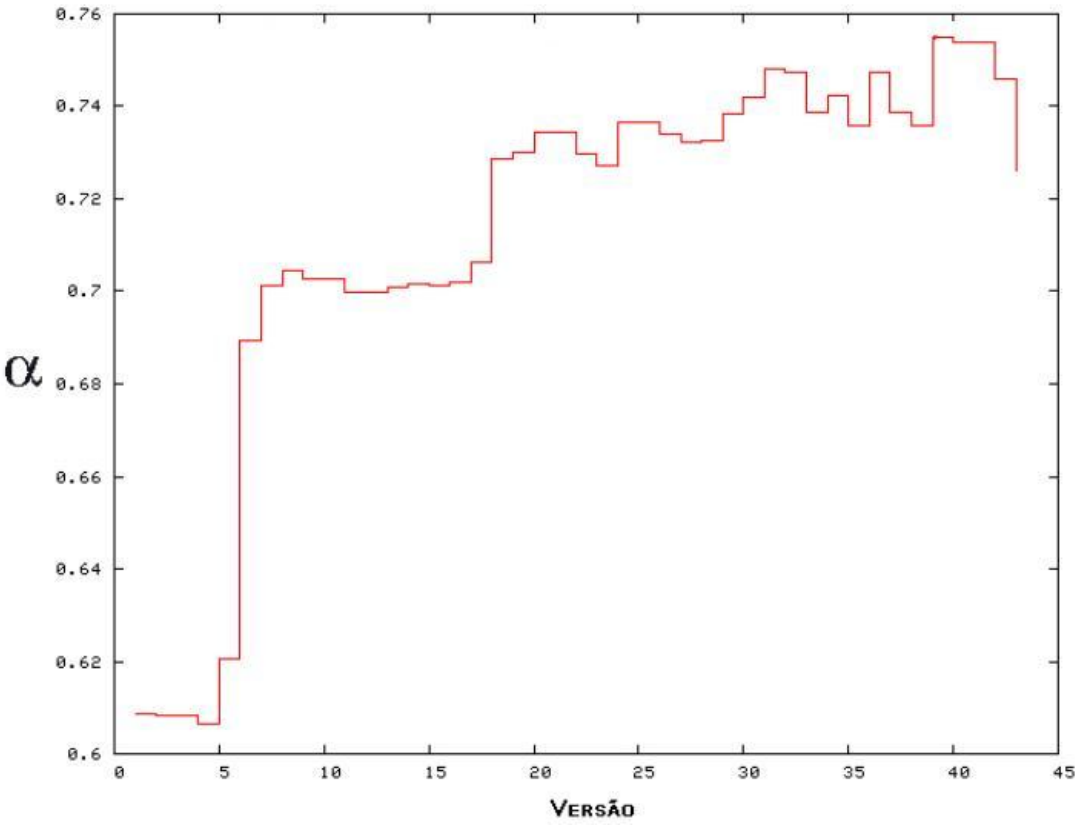
3º GRUPO

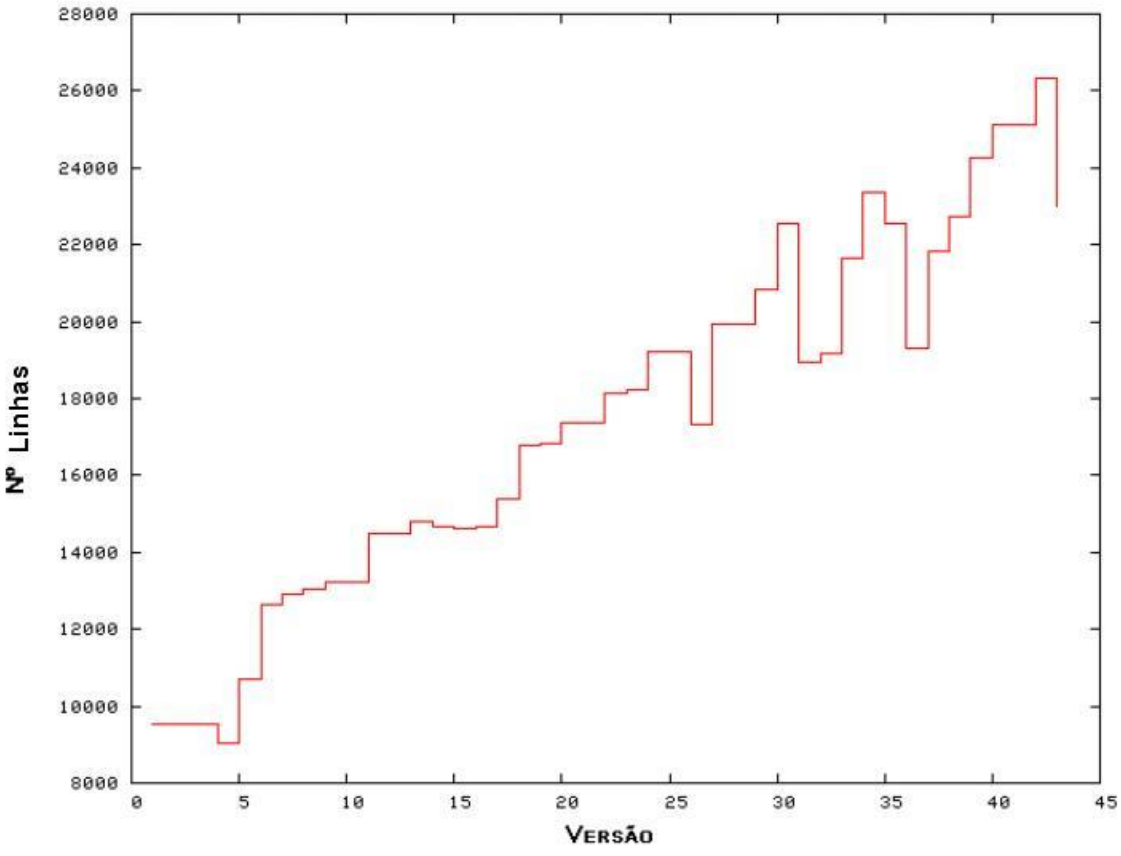
StatefulSessionInstanceInterceptor.java



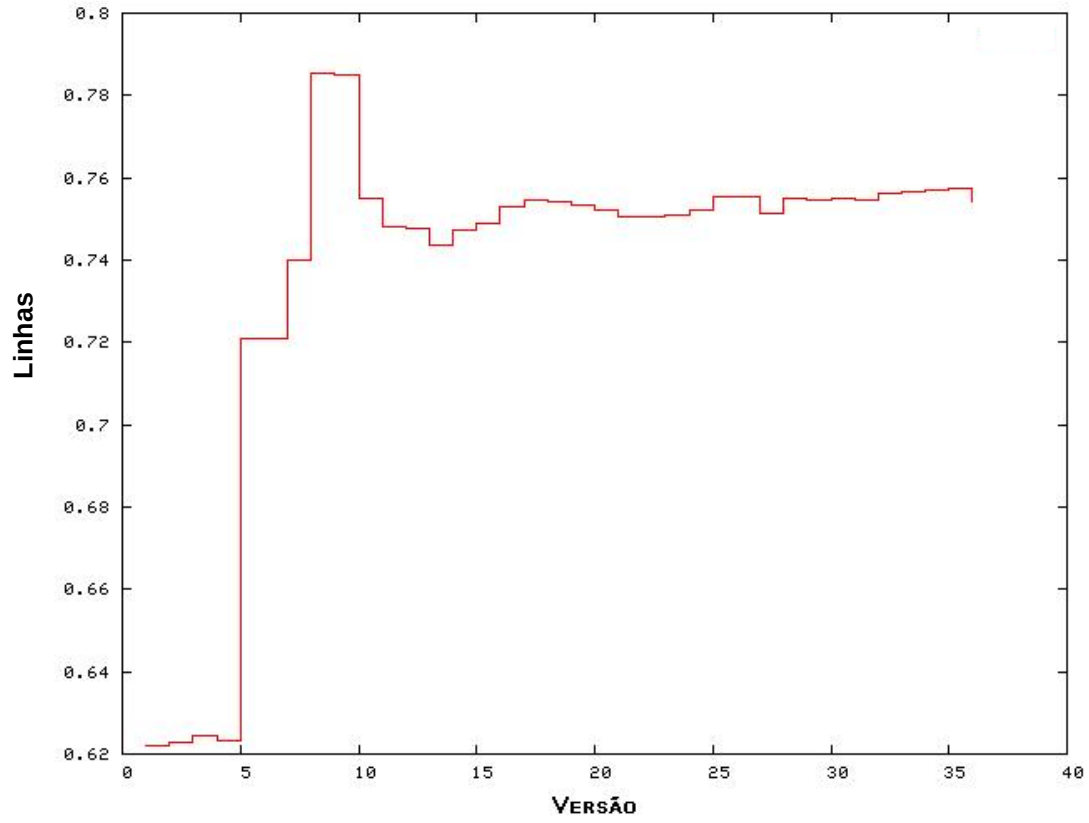


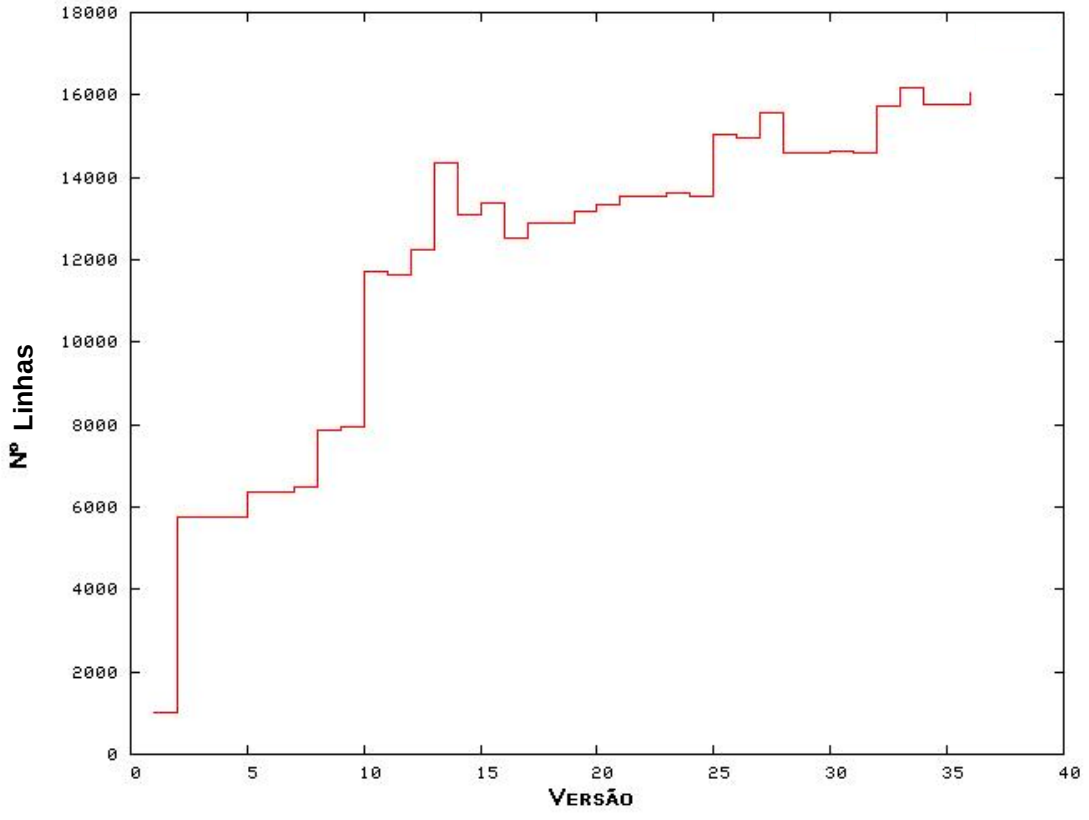
ApplicationMetaData.java



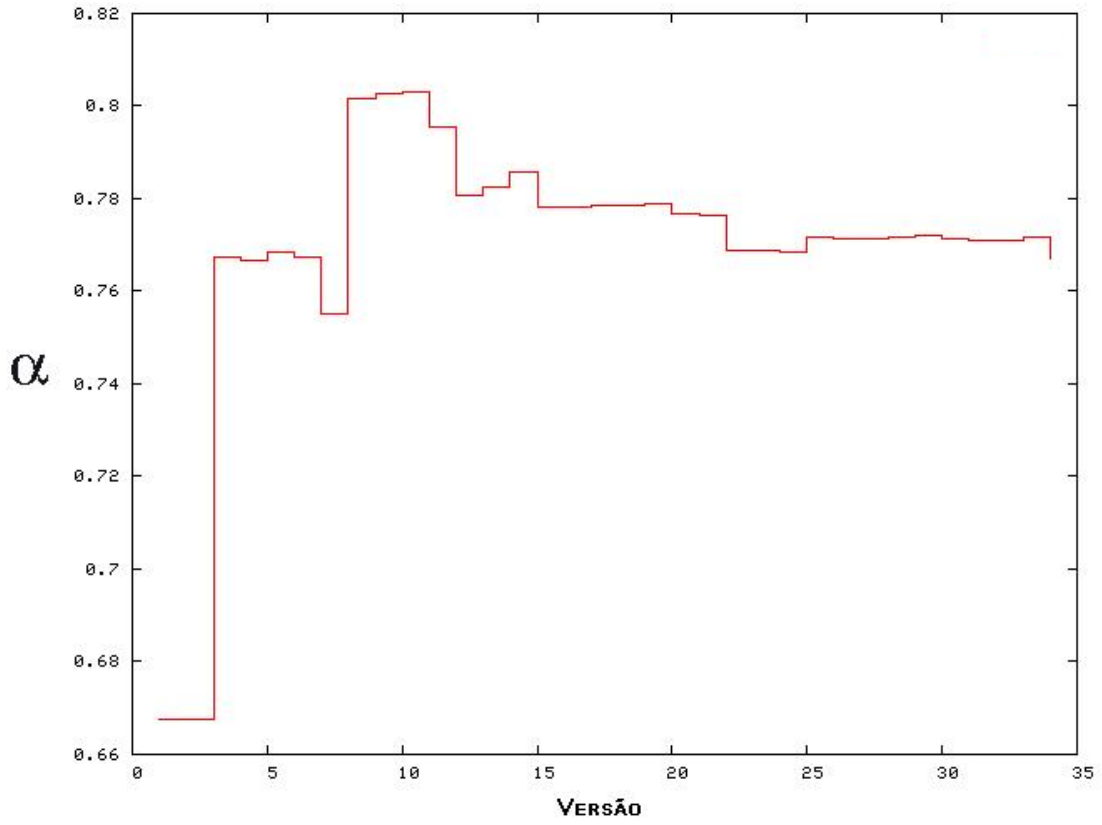


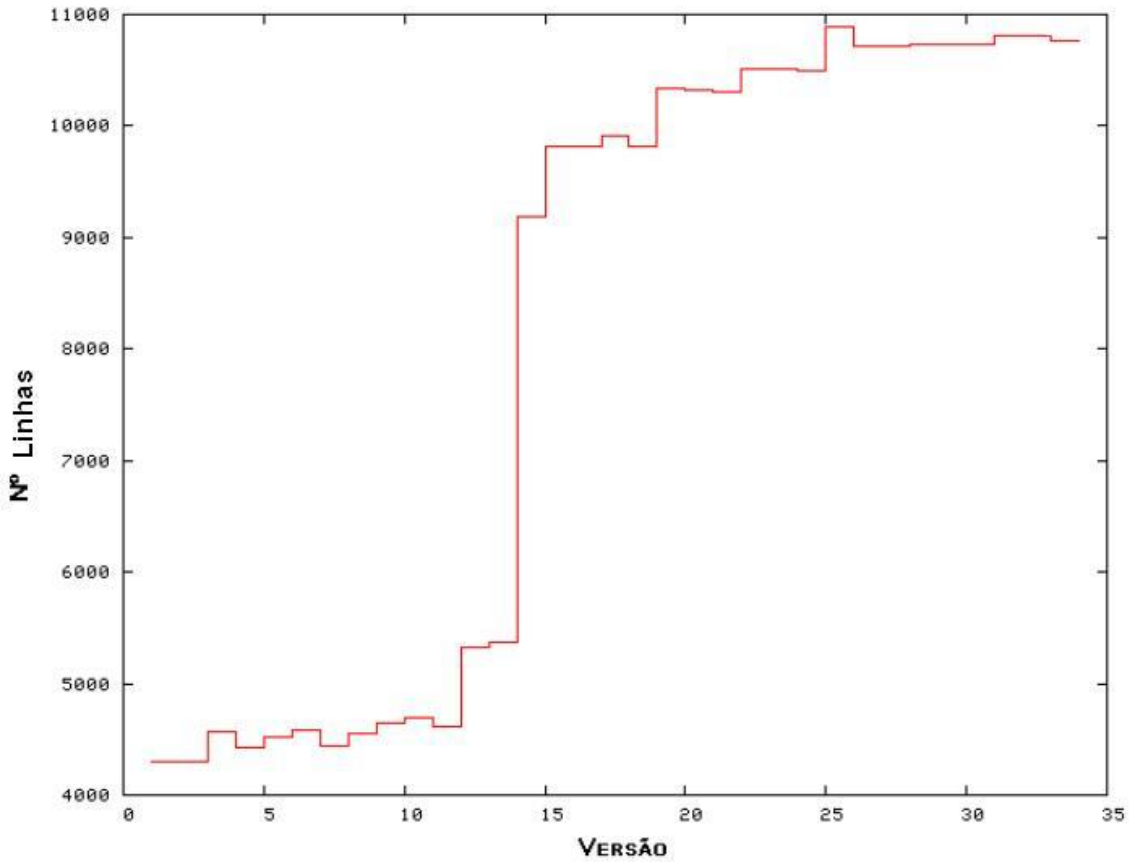
JDBCStartCommand.java



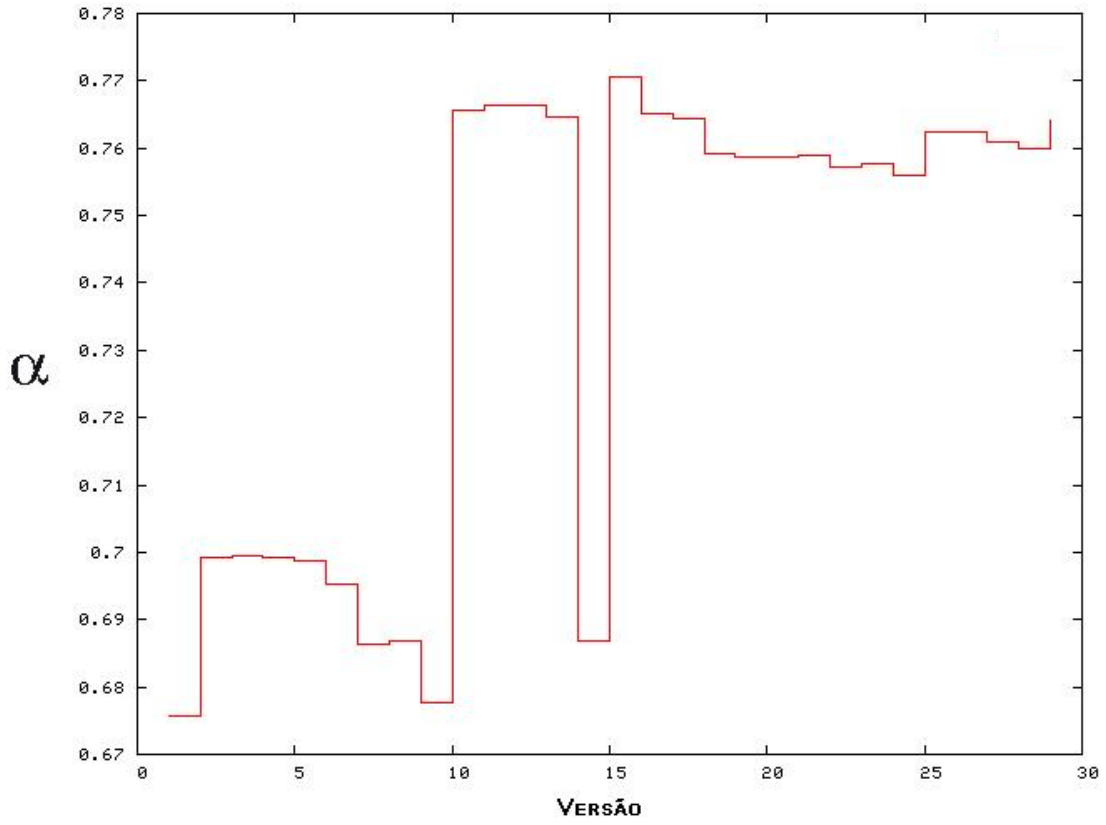


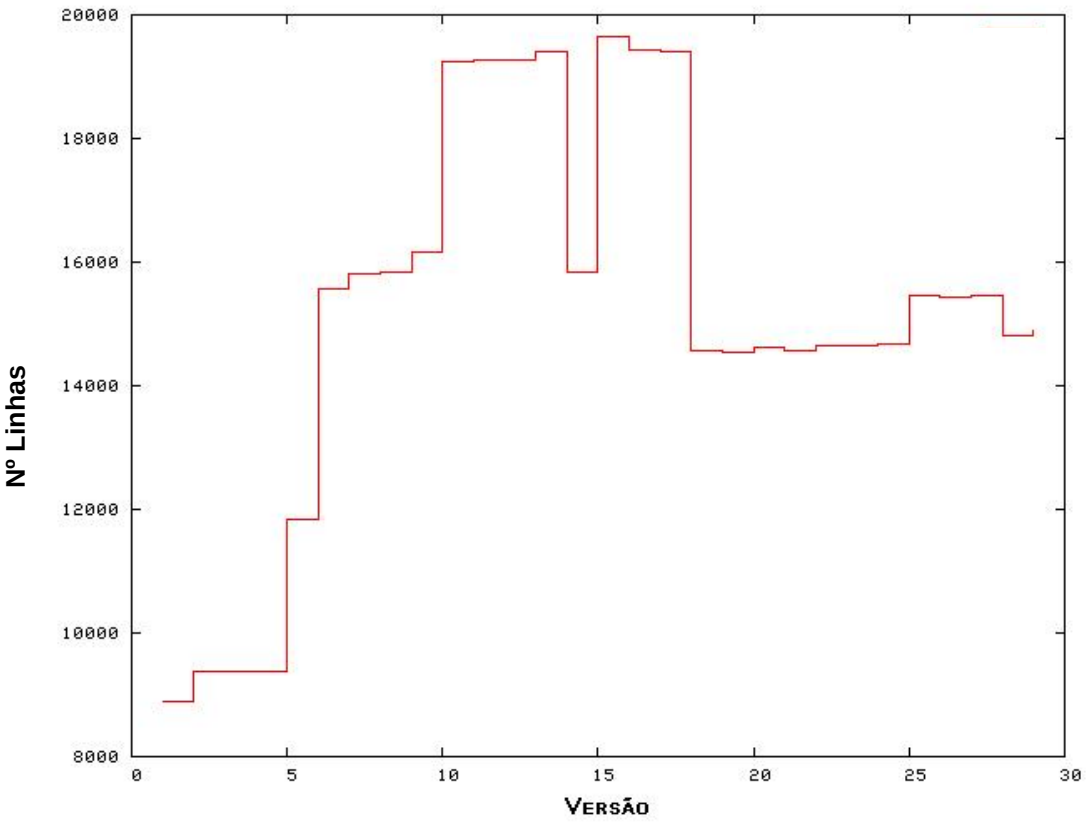
JDBCLoadEntityCommand.java



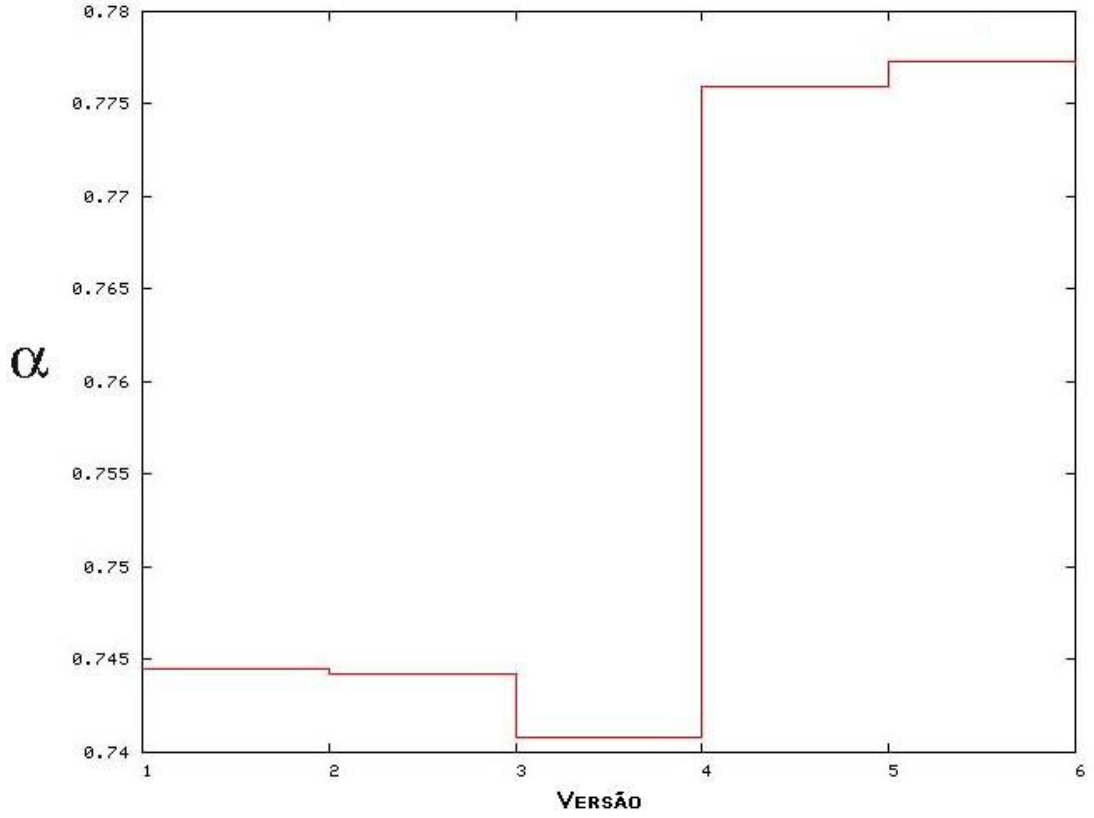


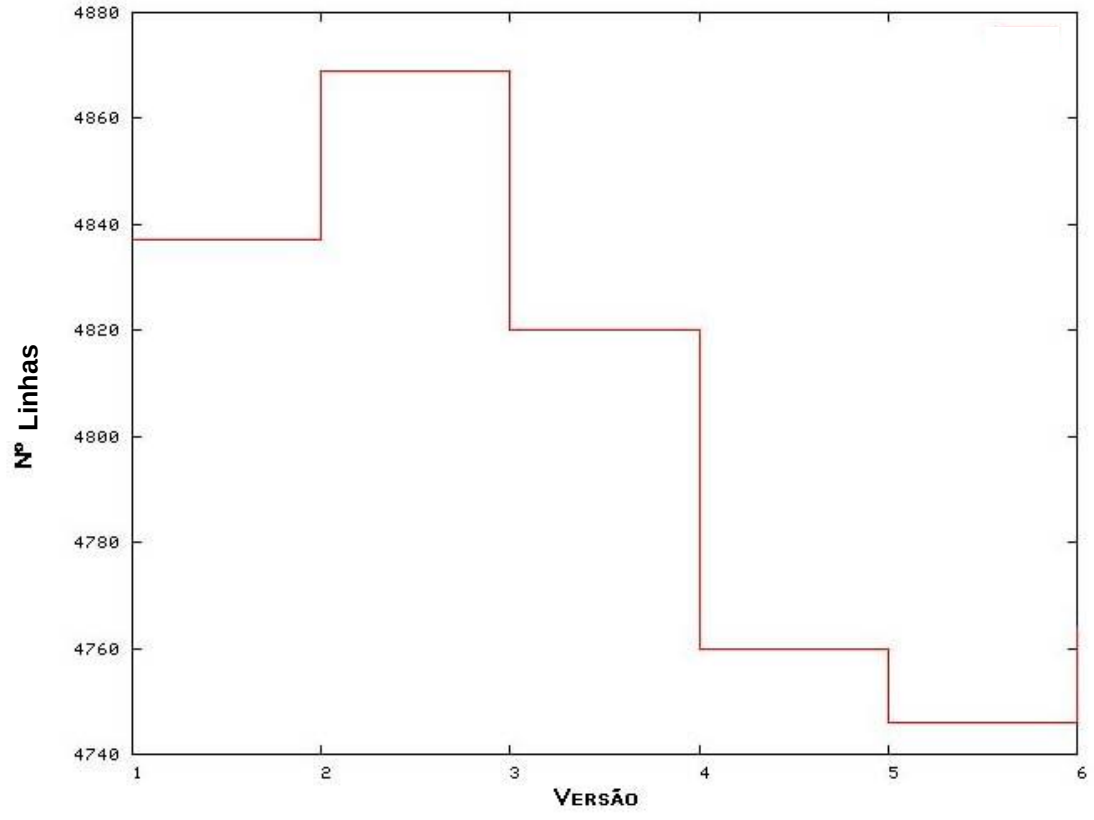
LRUEnterpriseContextCachePolicy.java



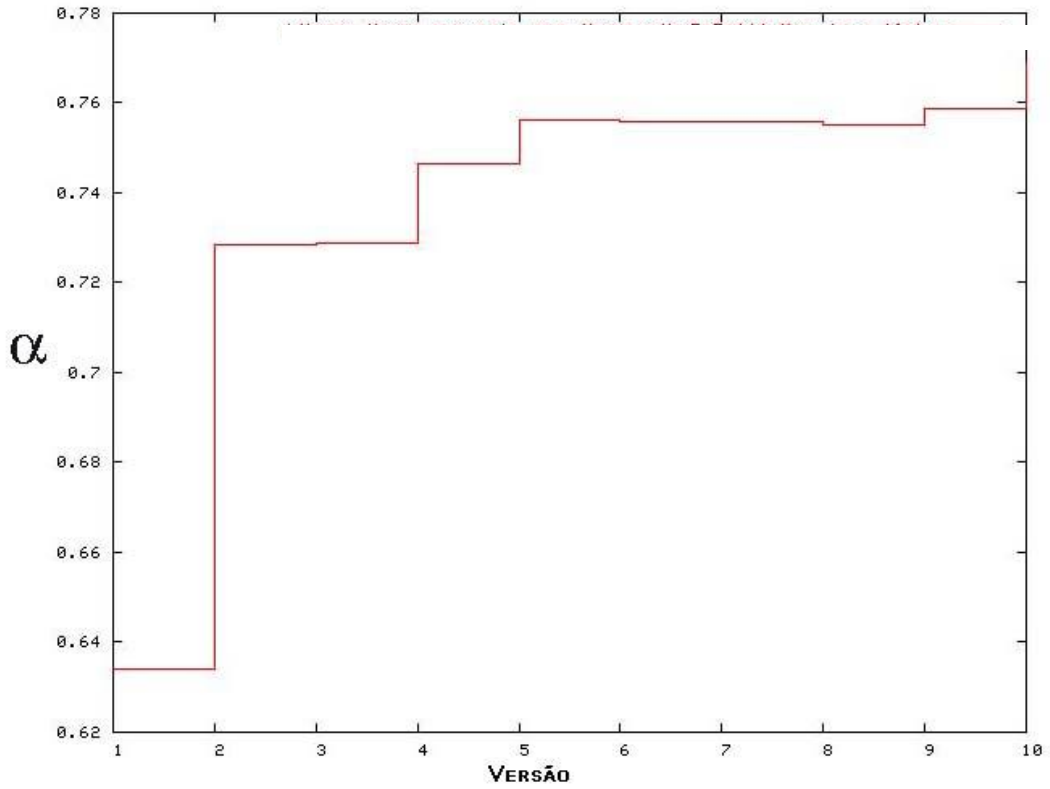


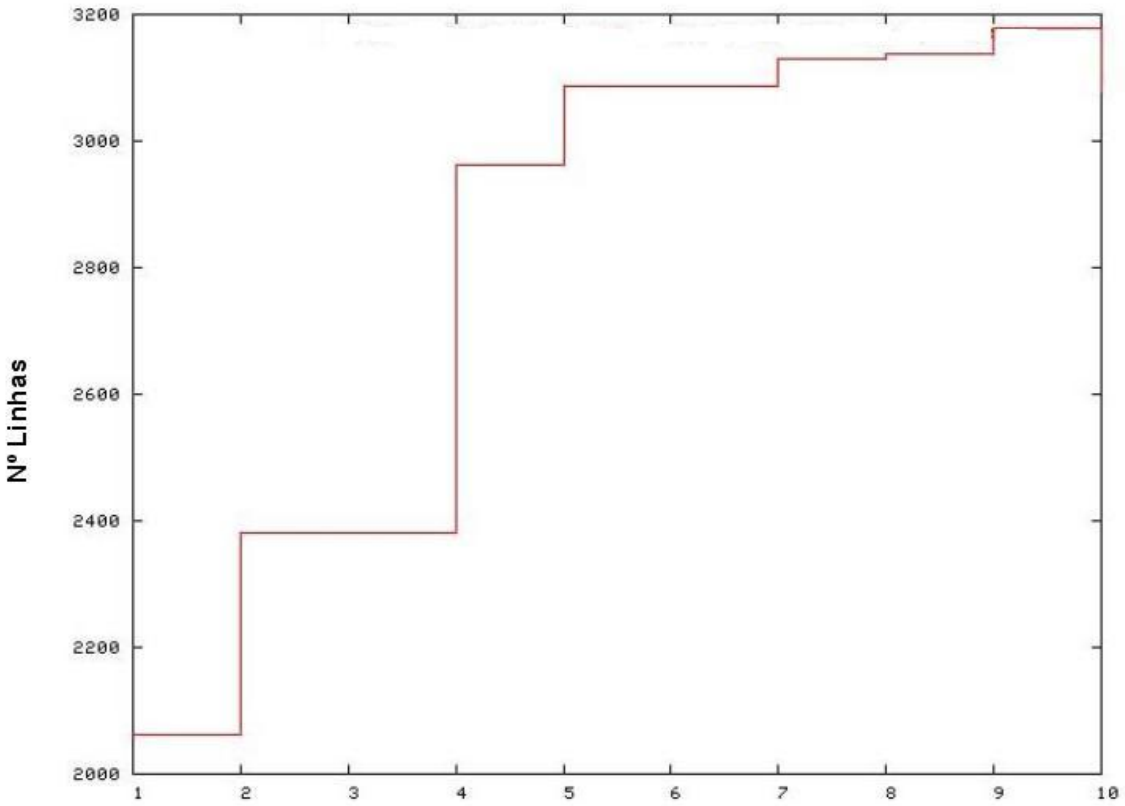
JMXInvokerInterceptor.java



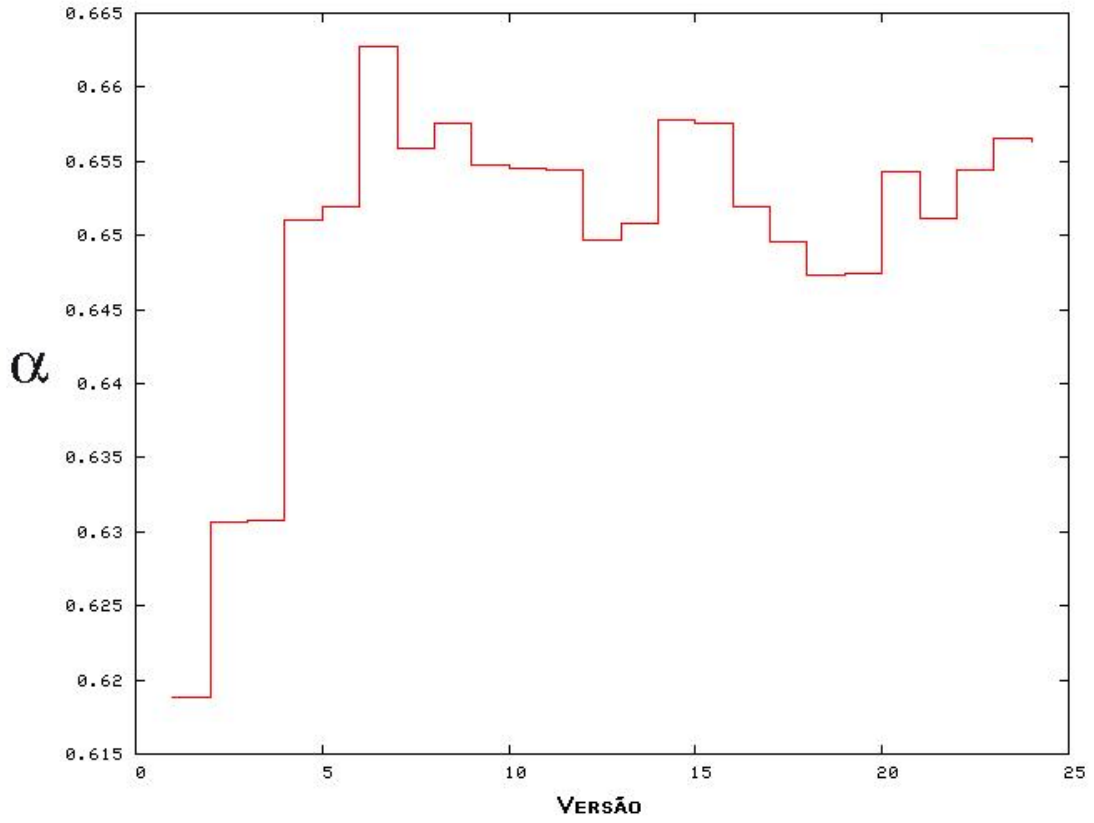


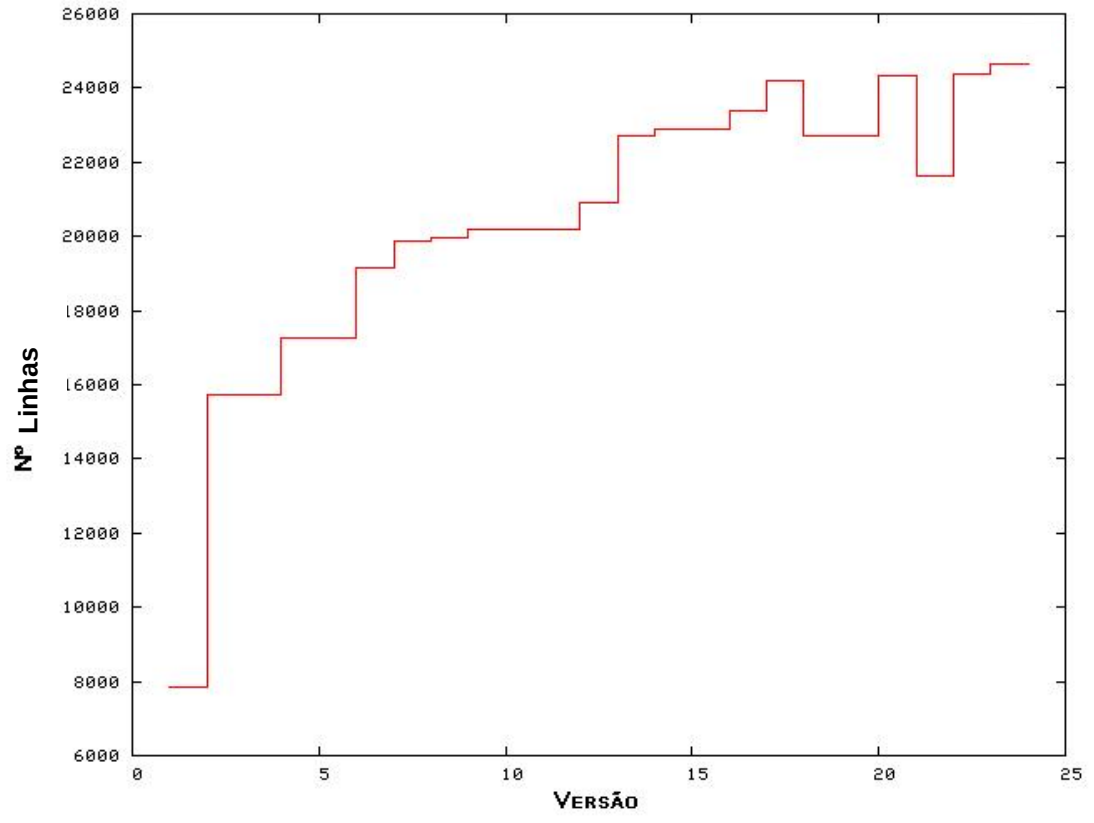
TxEntityMap.java



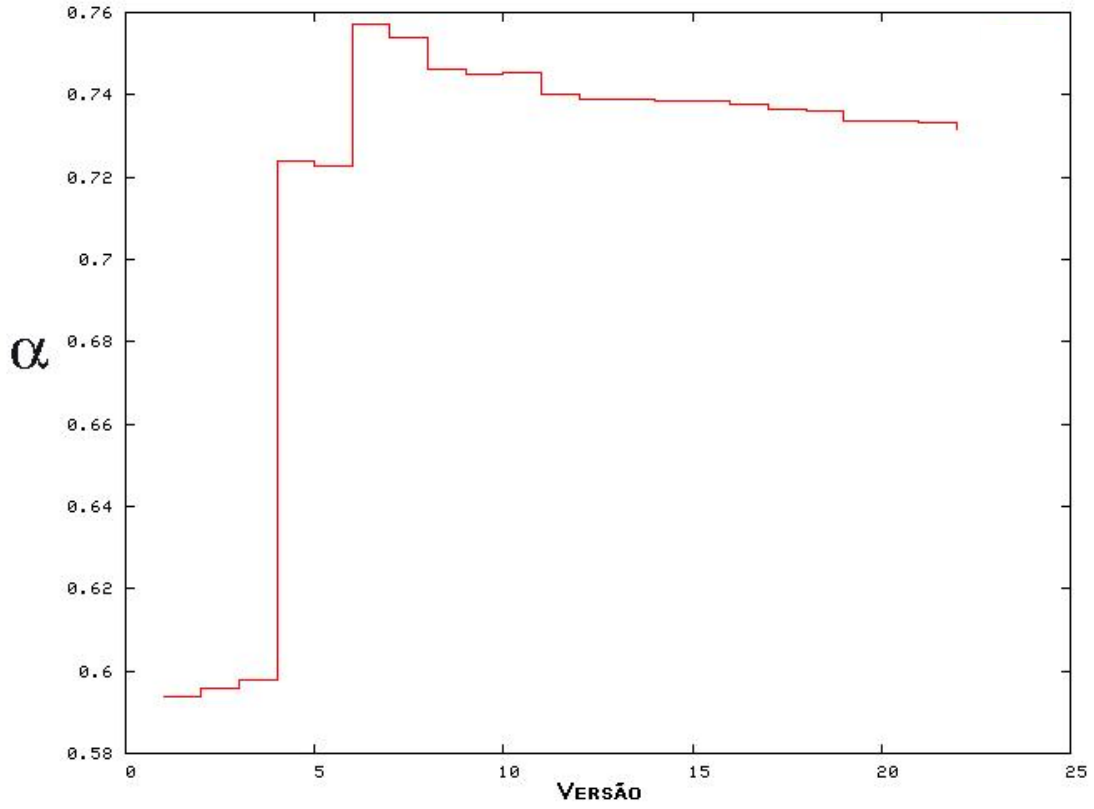


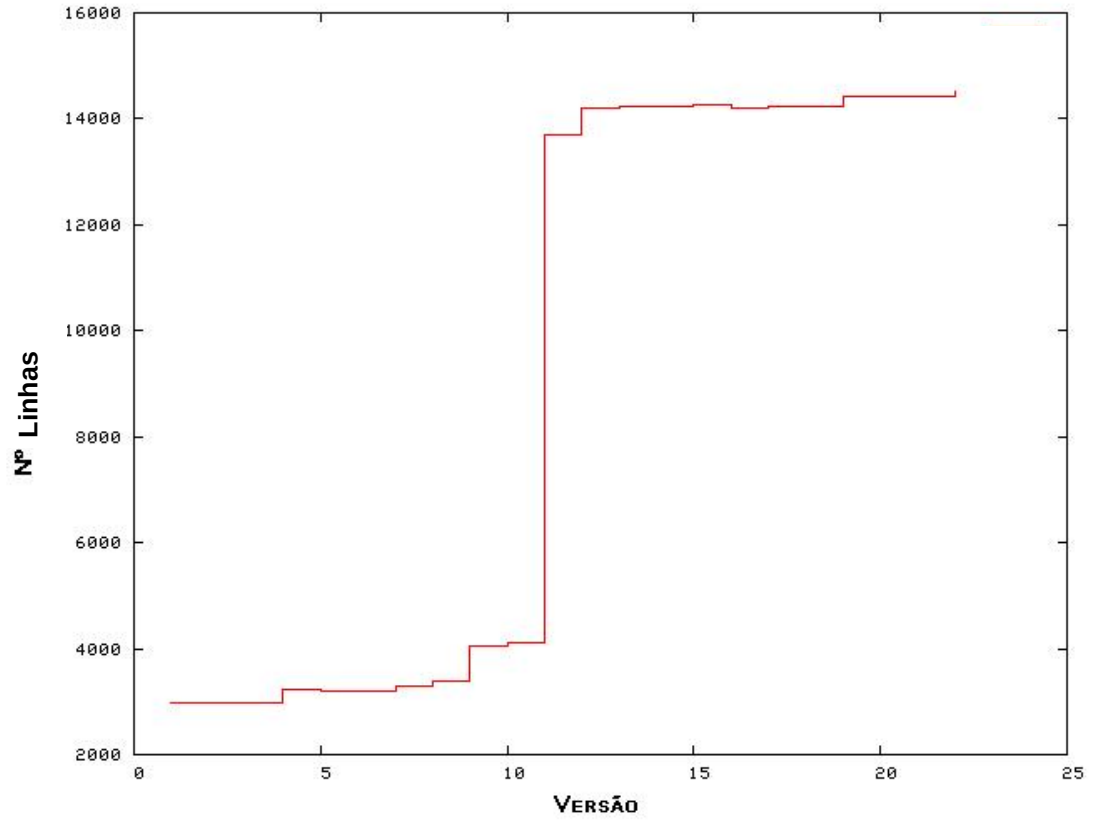
JDBCCMPFieldMetaData.java



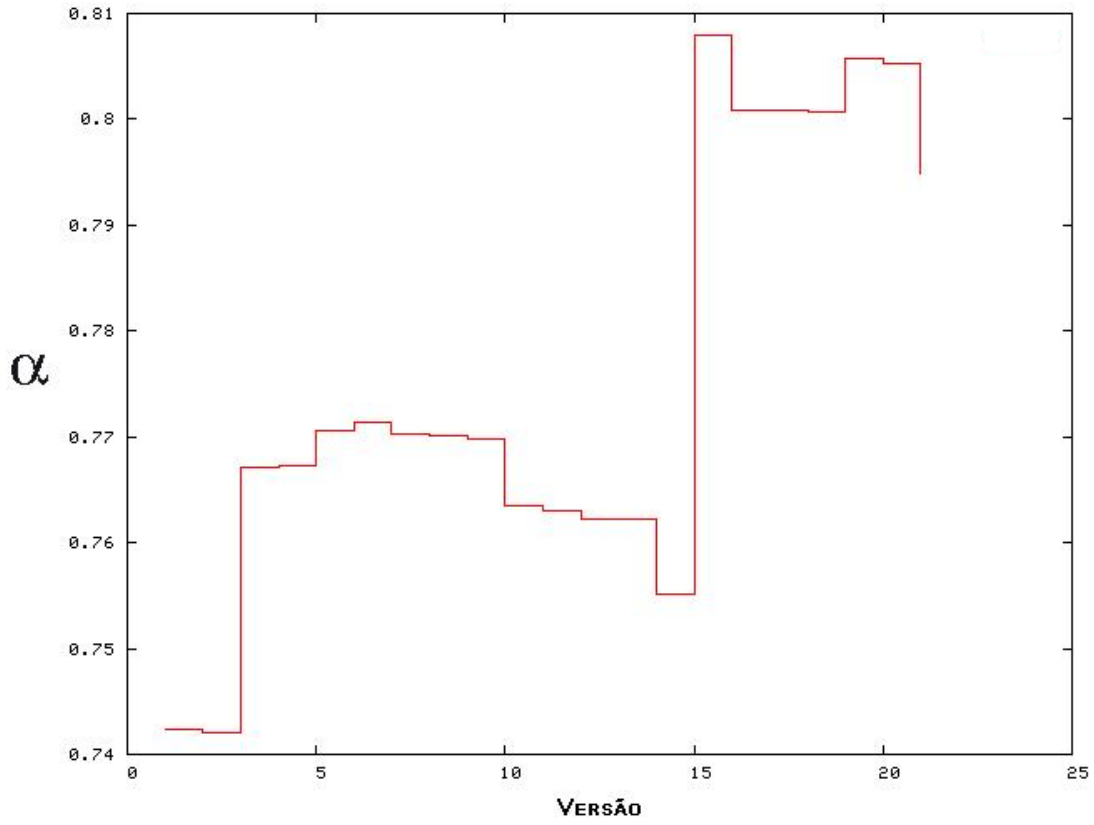


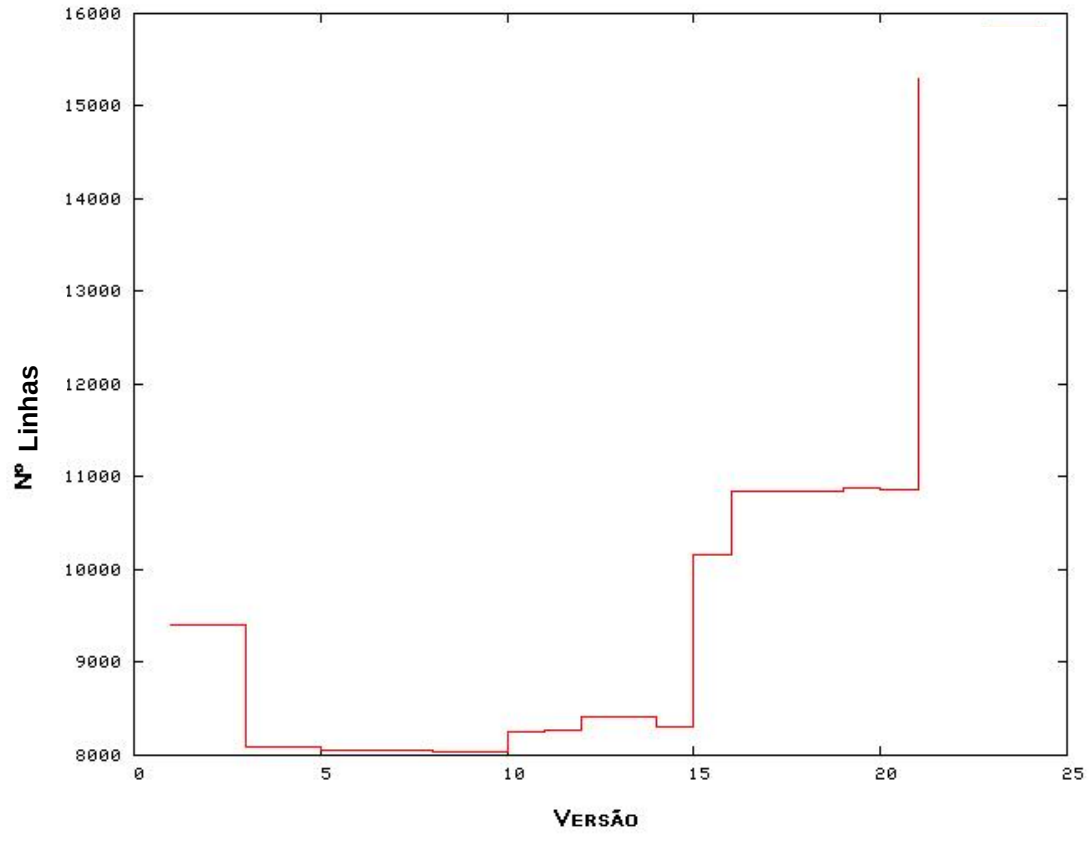
JDBCLoadRelationCommand.java



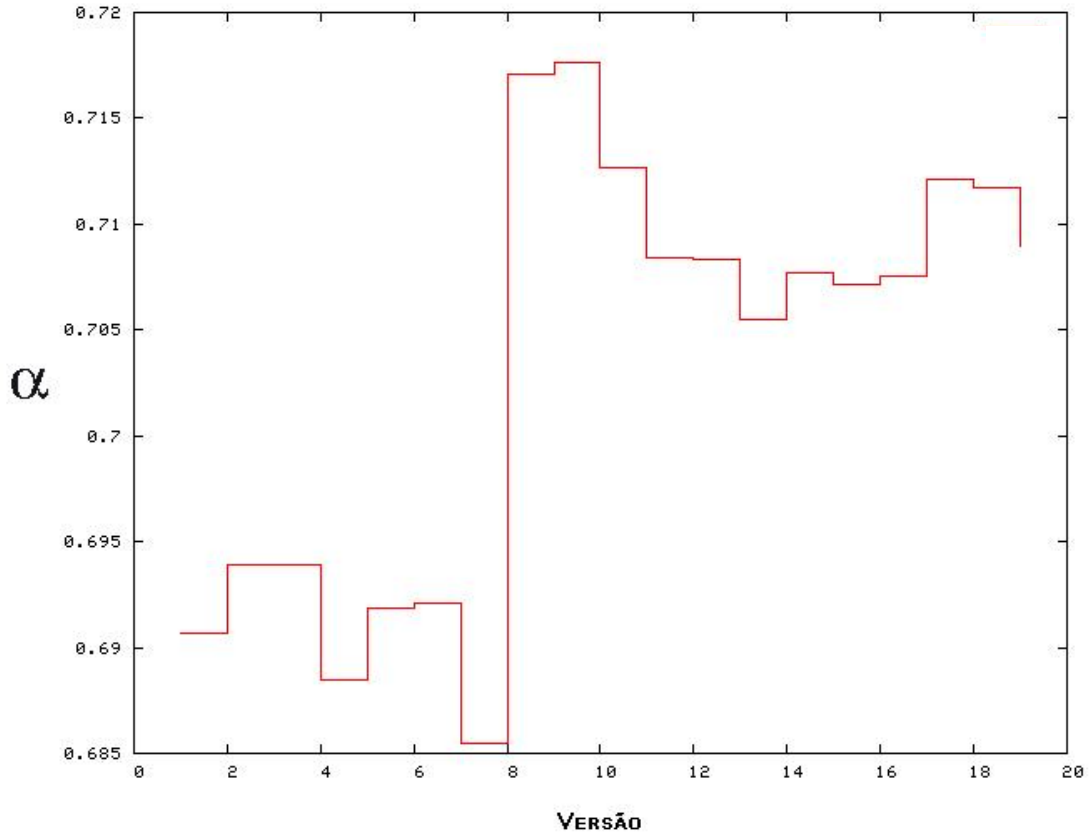


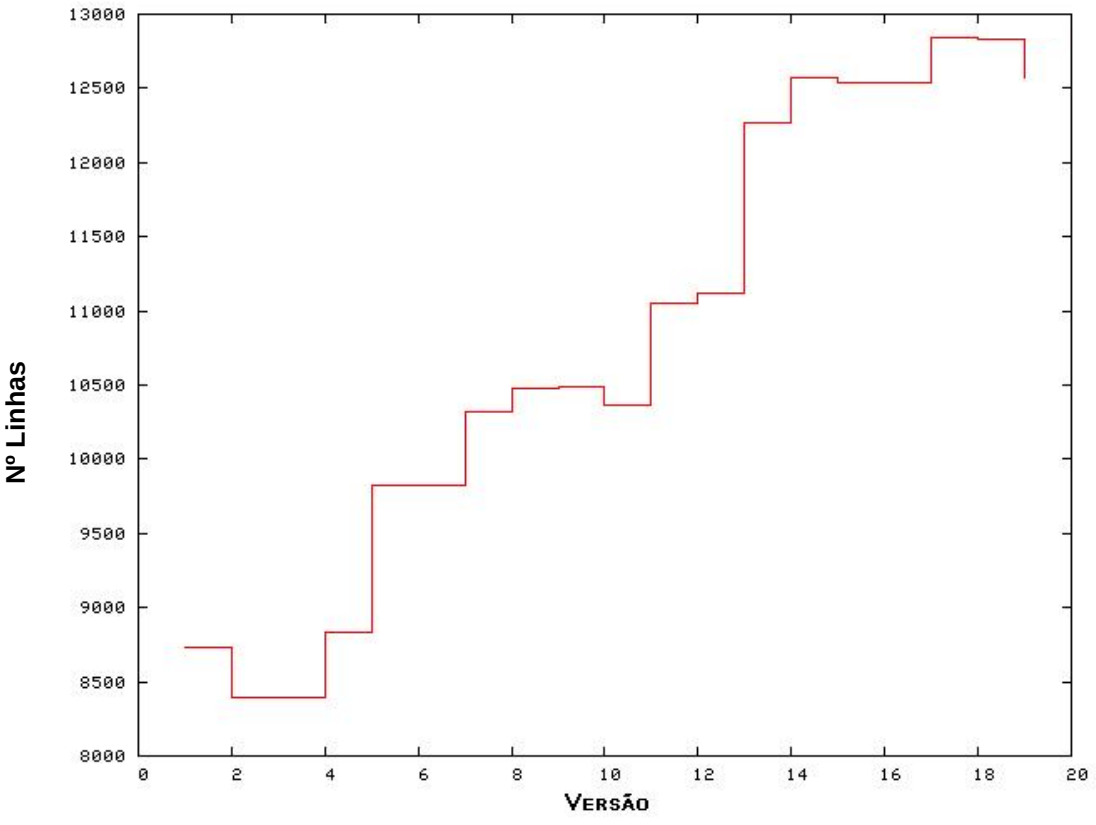
CMPFilePersistenceManager.java



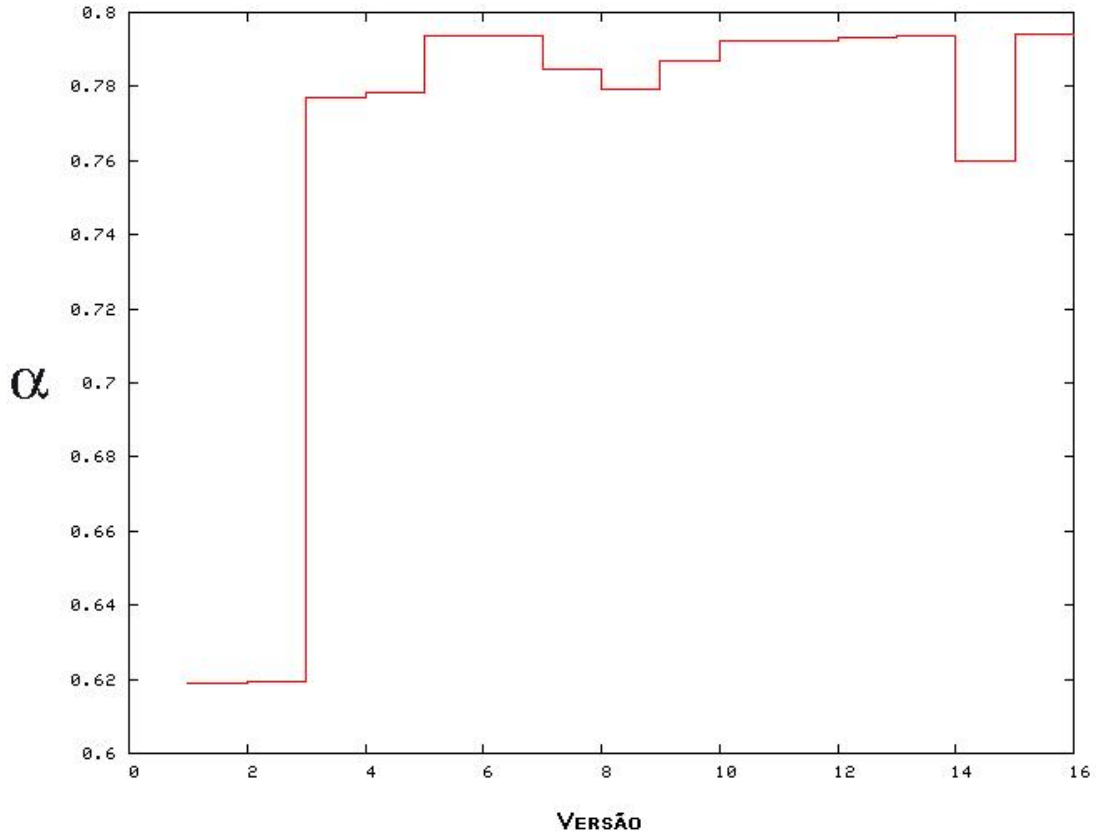


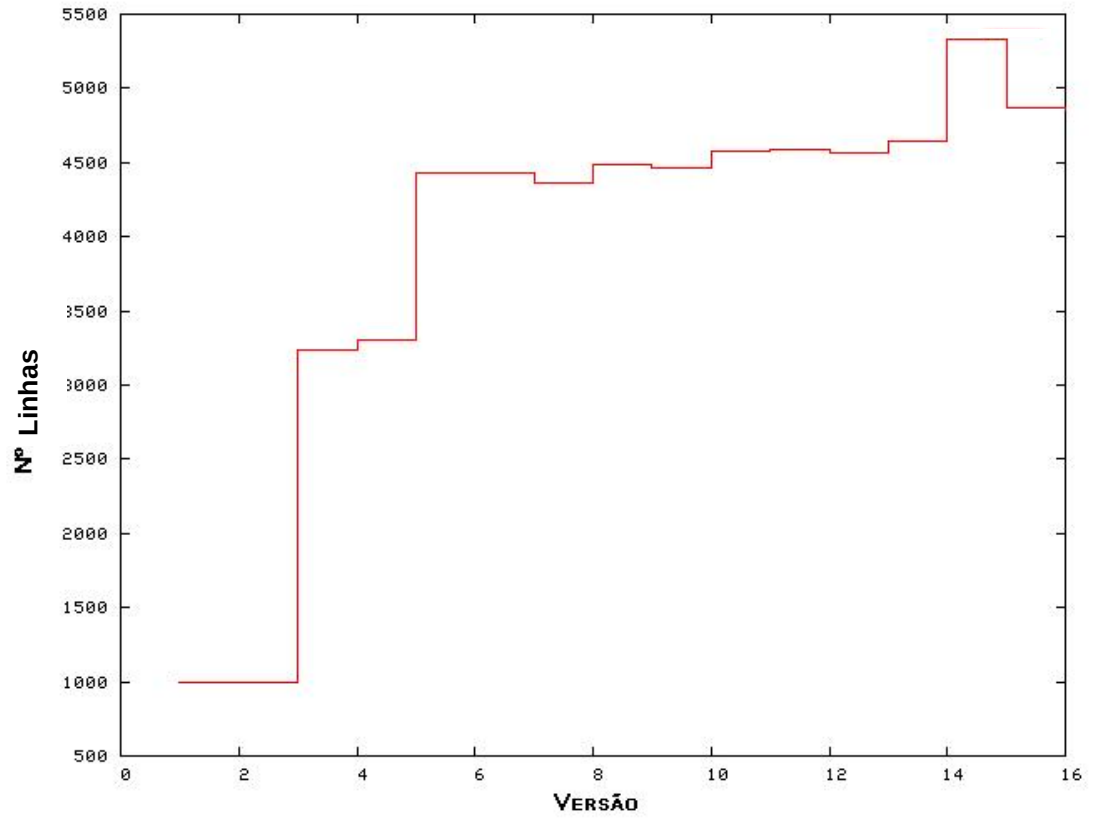
JDBCAbstractCMPFieldBridge.java



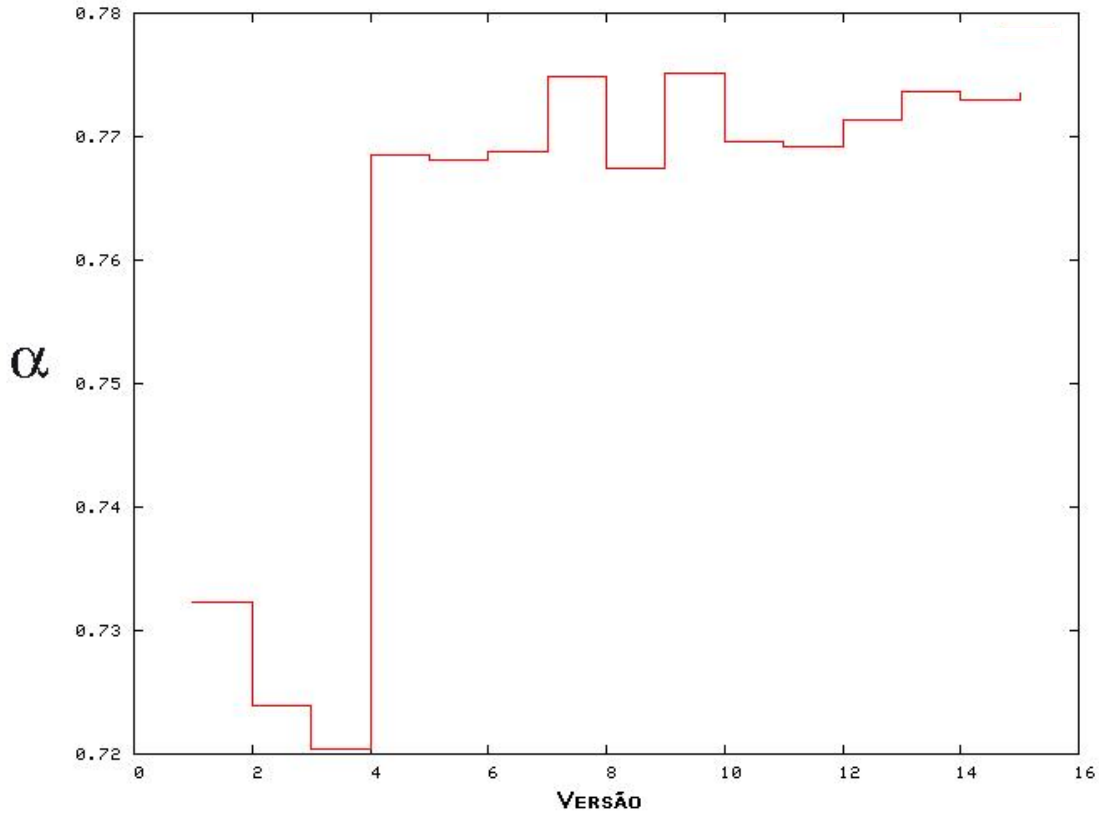


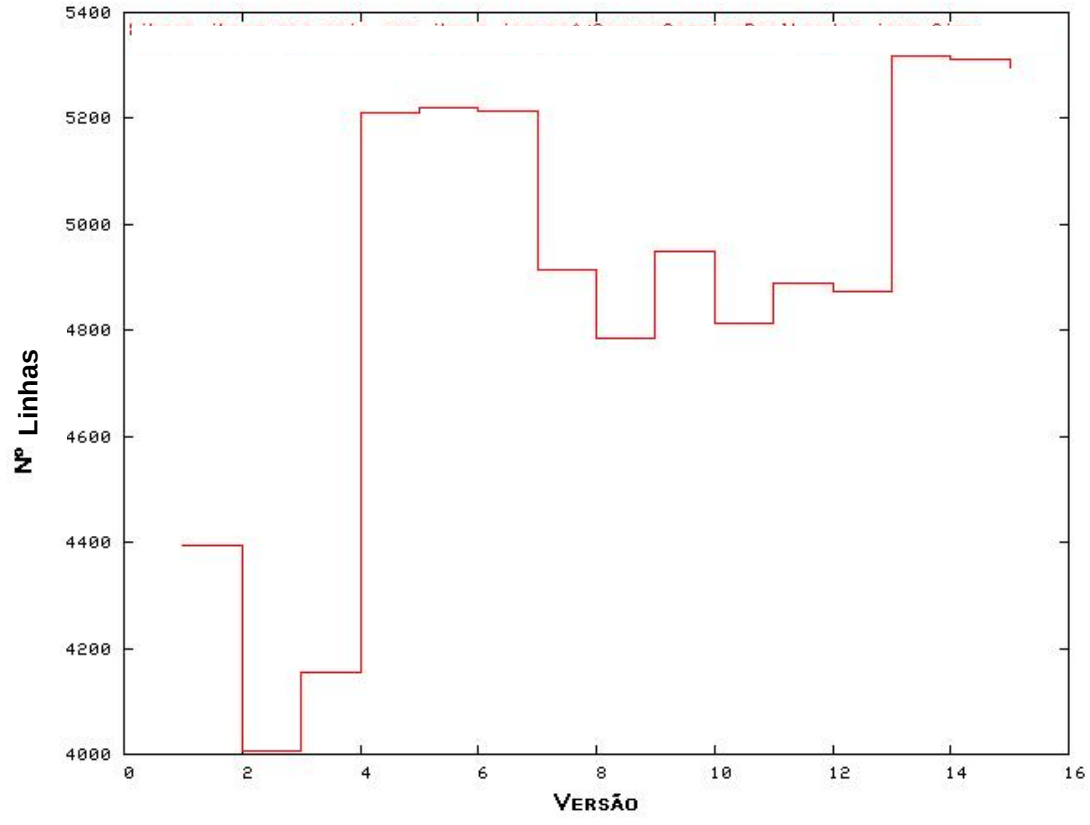
JDBCStopCommand.java



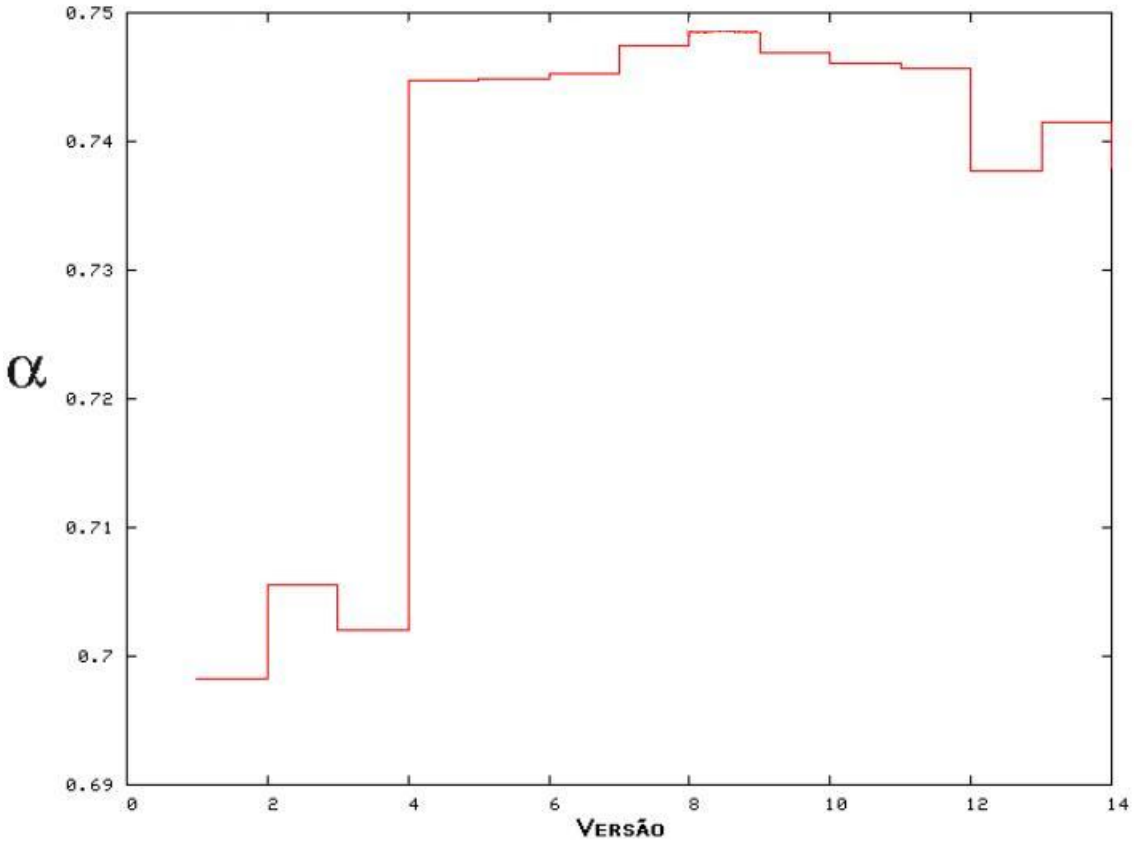


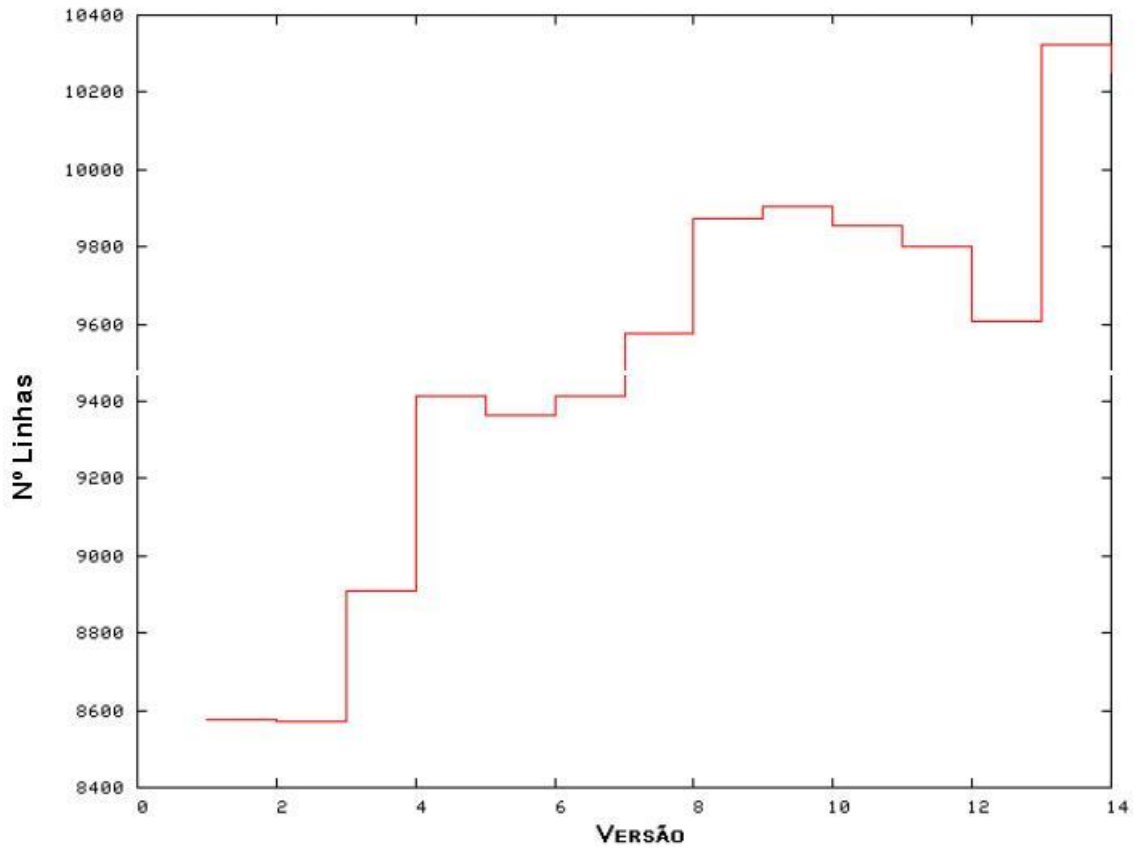
ServerSessionPoolLoader.java



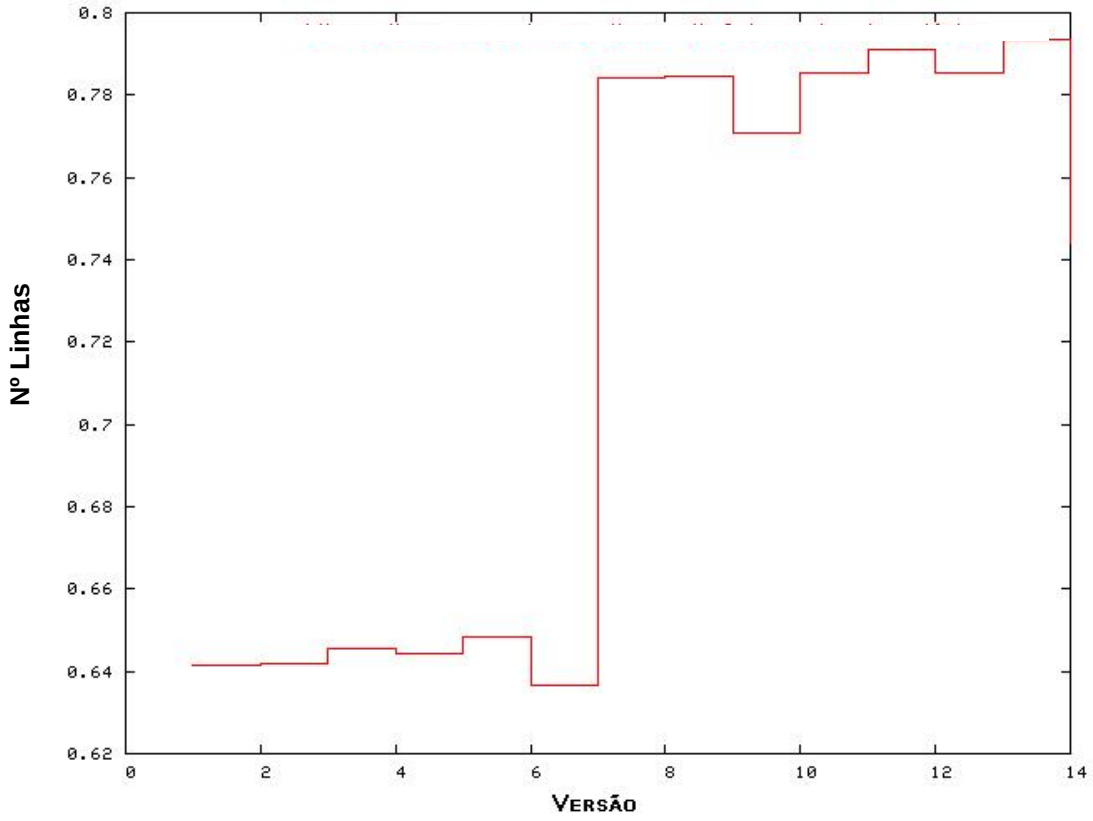


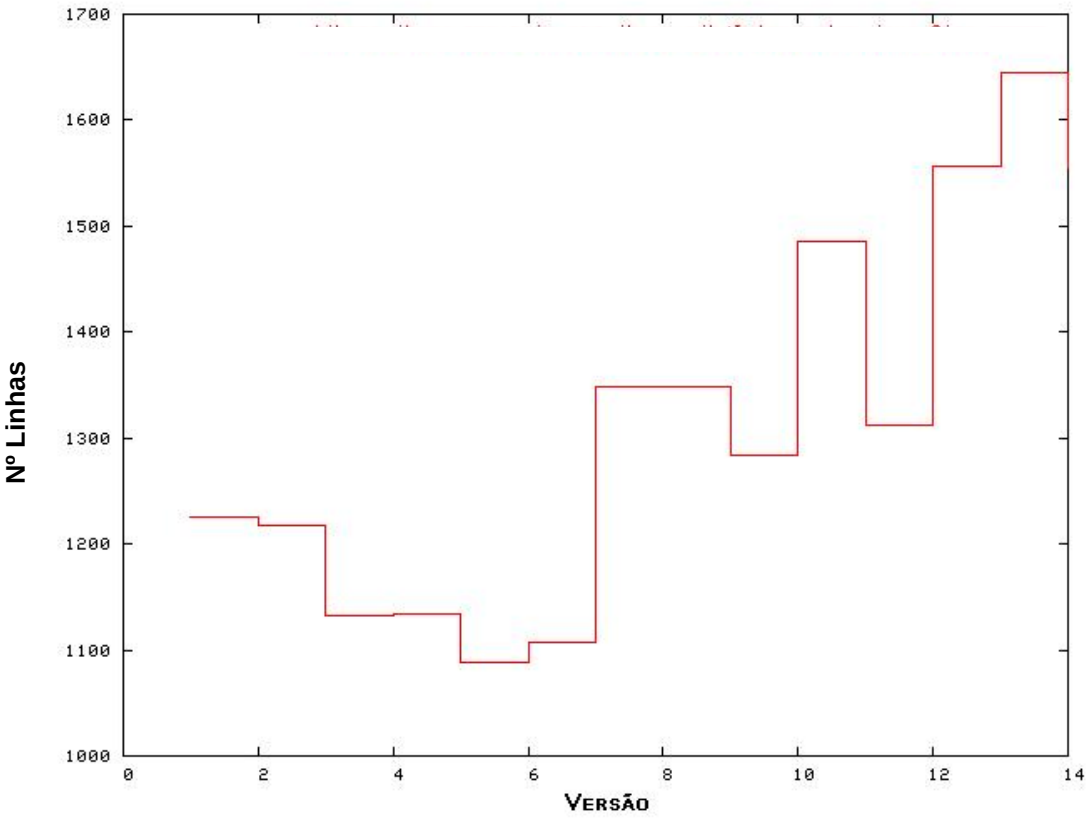
DLQHandler.java



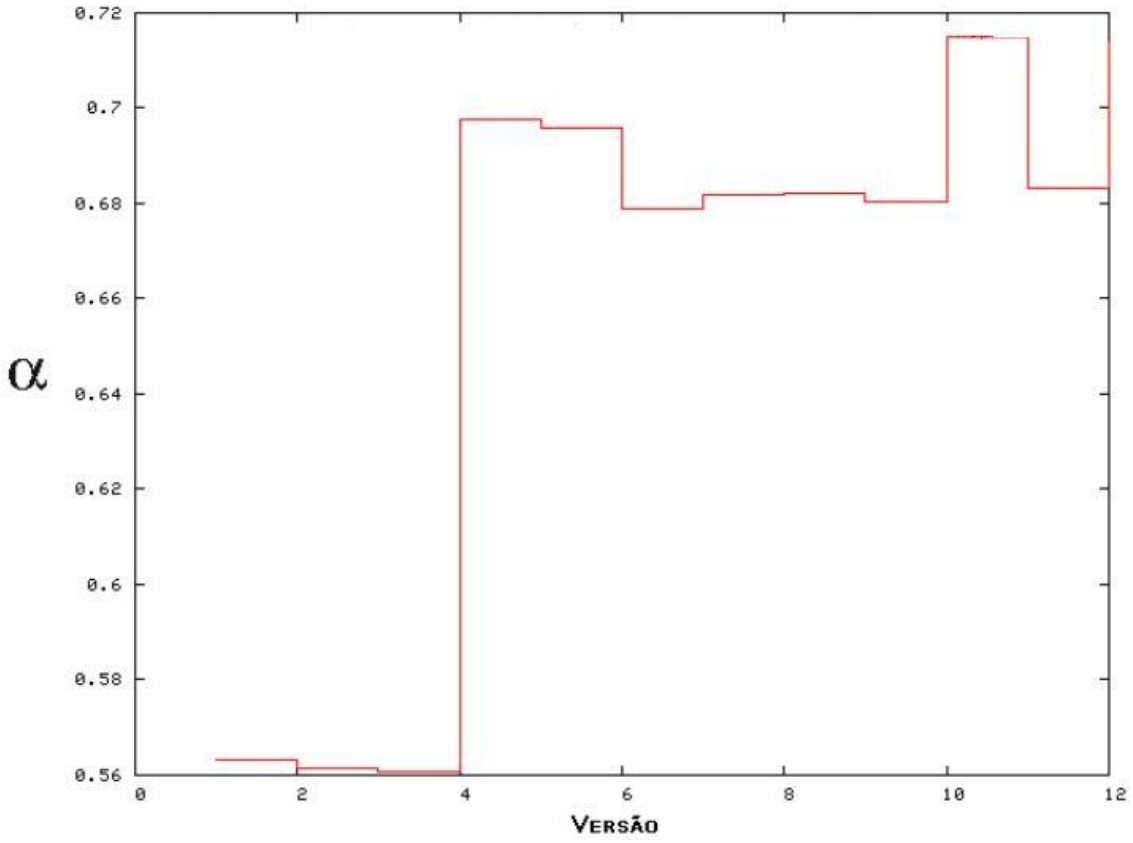


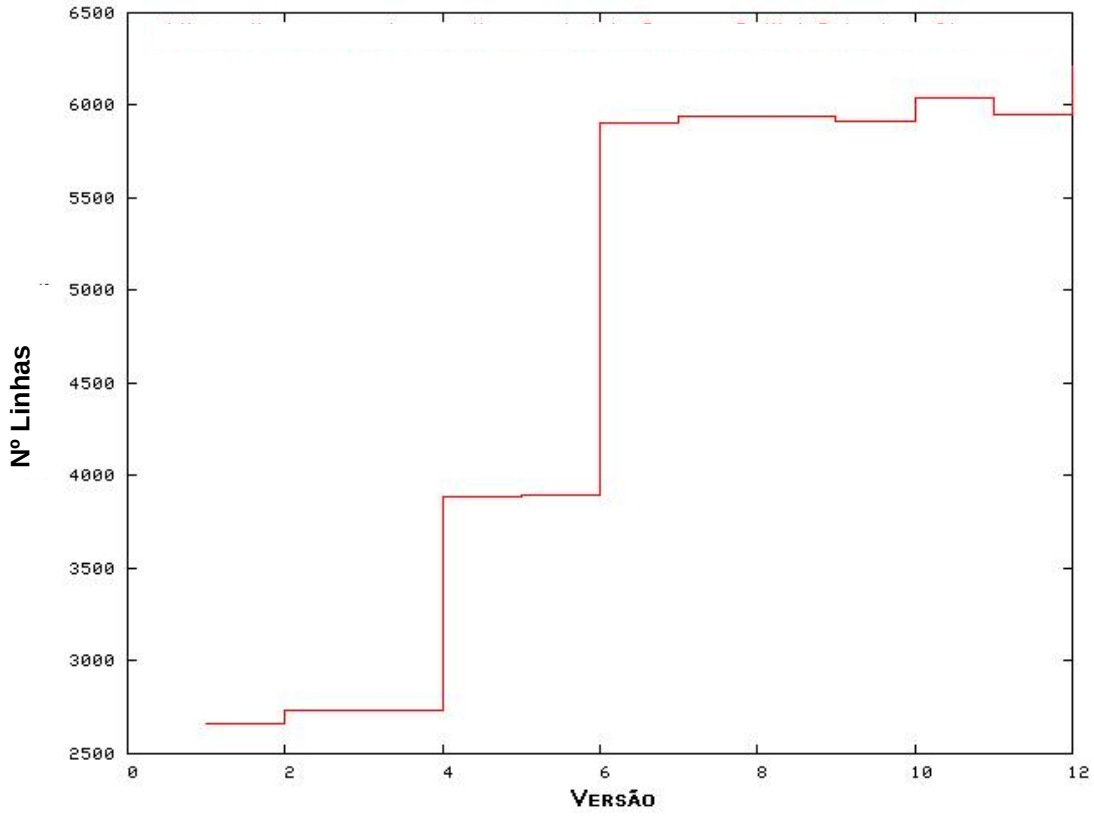
Interceptor.java





ResourceRefMetaData.java





Índice