



UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

Danilo Leal Maciel

**Sherlock N-Overlap: Normalização invasiva e
coeficiente de sobreposição para análise de
similaridade entre códigos-fonte em disciplinas de
programação**

FORTALEZA – CEARÁ

JULHO 2014

DANILO LEAL MACIEL

Sherlock N-Overlap: Normalização invasiva e coeficiente de sobreposição para análise de similaridade entre códigos-fonte em disciplinas de programação

*Dissertação de Mestrado apresentada à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará como parte dos requisitos para obtenção do grau de **Mestre em Engenharia de Teleinformática**.*

Área de Concentração: Sinais e Sistemas

Orientador: Prof. Dr. José Marques Soares

FORTALEZA – CEARÁ

JULHO 2014

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Pós-Graduação em Engenharia - BPGE

-
- M138s Maciel, Danilo Leal.
 Sherlock N-Overlap: normalização invasiva e coeficiente de sobreposição para análise de
 similaridade entre códigos-fonte em disciplinas de programação / Danilo Leal Maciel. – 2014.
- 105 f. : il. color. , enc. ; 30 cm.
- Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Tecnologia, Programa de Pós-
Graduação em Engenharia de Teleinformática, Fortaleza, 2014.
 Área de concentração: Sinais e Sistemas.
 Orientação: Prof. Dr. José Marques Soares.
1. Teleinformática. 2. Linguagem de programação. I. Título.



UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA
CAMPUS DO PICI, CAIXA POSTAL 6007 CEP 60.738-640
FORTALEZA – CEARÁ - BRASIL
FONE (+55) 85 3366-9467 – FAX (+55) 85 3366-9468

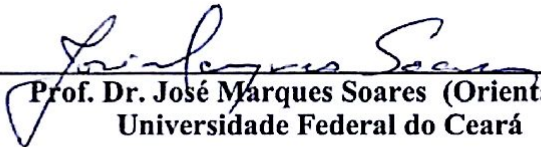
DANILO LEAL MACIEL

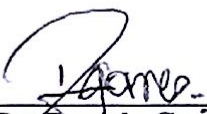
SHERLOCK N-OVERLAP: NORMALIZAÇÃO INVASIVA E COEFICIENTE DE SOBREPOSIÇÃO PARA ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE EM DISCIPLINAS DE PROGRAMAÇÃO

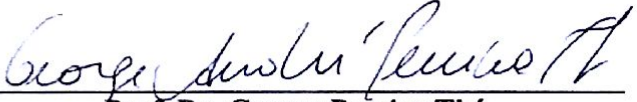
Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Teleinformática.
Área de concentração: Sinais e Sistemas.

Aprovada em: 07/07/2014.

BANCA EXAMINADORA


Prof. Dr. José Marques Soares (Orientador)
Universidade Federal do Ceará


Prof. Dr. Danilo Gonçalves Gomes
Universidade Federal do Ceará


Prof. Dr. George Pereira Thé
Universidade Federal do Ceará


Prof. Dr. Ig Ibert Bittencourt Santana Pinto
Universidade Federal de Alagoas

Resumo

Este trabalho se contextualiza no problema da detecção de plágio entre códigos-fonte em turmas de programação. Apesar da ampla quantidade de ferramentas disponíveis para a detecção de plágio, poucas são capazes de identificar, de maneira eficaz, todas as semelhanças léxicas e semânticas entre pares de códigos, o que se deve à complexidade inerente a esse tipo de análise. Fez-se, portanto, para o problema e o cenário em questão, um estudo das principais abordagens discutidas na literatura sobre detecção de plágio em código-fonte e, como principal contribuição, concebeu-se uma ferramenta aplicável no domínio de práticas laboratoriais. A ferramenta tem por base o algoritmo Sherlock, que foi aprimorado sob duas perspectivas: a primeira, com modificações no coeficiente de similaridade usado pelo algoritmo, de maneira a melhorar a sua sensibilidade para comparação de assinaturas; a segunda, propondo técnicas de pré-processamento invasivas que, além de eliminar informação irrelevante, sejam também capazes de sobrevalorizar aspectos estruturais da linguagem de programação, reunindo ou separando sequências de caracteres cujo significado seja mais expressivo para a comparação ou, ainda, eliminando sequências menos relevantes para destacar outras que permitam melhor inferência sobre o grau de similaridade. A ferramenta, denominada Sherlock N-Overlap, foi submetida a rigorosa metodologia de avaliação, tanto em cenários simulados como em turmas de programação, apresentando resultados superiores a ferramentas atualmente em destaque na literatura sobre detecção de plágio.

Palavras-chaves: detecção de plágio, análise de similaridade, práticas laboratoriais de programação

Abstract

This work is contextualized in the problem of plagiarism detection among source codes in programming classes. Despite the wide set of tools available for the detection of plagiarism, only few tools are able to effectively identify all lexical and semantic similarities between pairs of codes, because of the complexity inherent to this type of analysis. Therefore to the problem and the scenario in question, it was made a study about the main approaches discussed in the literature on detecting plagiarism in source code and as a main contribution, conceived to be a relevant tool in the field of laboratory practices. The tool is based on Sherlock algorithm, which has been enhanced as of two perspectives: firstly, with changes in the similarity coefficient used by the algorithm in order to improve its sensitivity for comparison of signatures; secondly, proposing intrusive techniques preprocessing that, besides eliminating irrelevant information, are also able to overemphasize structural aspects of the programming language, or gathering separating strings whose meaning is more significant for the comparison or even eliminating sequences less relevant to highlight other enabling better inference about the degree of similarity. The tool, called Sherlock N-Overlap was subjected to rigorous evaluation methodology, both in simulated scenarios as classes in programming, with results exceeding tools currently highlighted in the literature on plagiarism detection.

Keywords: plagiarism detection, similarity analysis, laboratory programming practices

Dedico este trabalho aos meus pais.

Agradecimentos

Agradeço primeiramente a Deus por todas as bênçãos, inspiração e proteção.

Aos meus pais, Antônio e Liduina, pelo amor e a educação, que sem a qual este trabalho não teria sido realizado.

Aos meus irmãos, Davi e Lia, pelo apoio moral e incentivo.

À minha noiva e melhor amiga, Jardênia Cavalcante pelo apoio, paciência, dedicação e companheirismo mesmo nos momentos mais difíceis.

Ao Professor Dr. José Marques Soares pelos conselhos científicos, disponibilidade e pelas condições oferecidas para a realização deste trabalho. Agradeço também aos demais professores do PPGETI, pelas experiências e ensinamentos transmitidos.

A todos os amigos que acompanharam o decorrer do curso compartilhando experiências nos momentos de alegria e de dificuldades e assim também contribuíram para a realização deste trabalho, de modo especial Vanessa Viana, Edigleison Carvalho, Rodrigo Teles, Paulo André, Allyson Bonetti, Bruno Monte, Nathan Camargos e Artur Rodrigues.

Por fim, à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

Sumário

Lista de Ilustrações	viii
Lista de Tabelas	xi
1 Introdução	1
1.1 Problematização	3
1.2 Objetivos	4
1.3 Organização do texto	5
2 Fundamentação Teórica	6
2.1 Técnicas de plágio	6
2.2 Algoritmos, Ferramentas e Sistemas para detecção de plágio	10
2.2.1 YAP3	11
2.2.2 JPlag	11
2.2.3 MOSS	14
2.2.4 SIM	15
2.2.5 Sherlock	17
2.3 Coeficientes de similaridade	21
2.3.1 Coeficiente de Jaccard	21
2.3.2 Coeficiente de Sorensen–Dice	22
2.3.3 Coeficiente Overlap	23
2.3.4 Coeficiente de Tversky	23
2.4 Considerações finais	24

3	Sherlock N-Overlap: Normalização invasiva e coeficiente de sobreposição para identificação de similaridade	25
3.1	Normalizando códigos-fonte para comparação com Sherlock	25
3.1.1	Análise da organização das estruturas quanto aos espaços em branco	28
3.1.2	Discussões sobre o alto grau de invasão proporcionado pela Normalização 4	29
3.2	Sherlock N-Overlap: adaptando o algoritmo Sherlock original com o coeficiente de sobreposição	31
3.3	Discussões sobre o Sherlock N-Overlap	32
3.4	Considerações Finais	33
4	Um arcabouço para análise comparativa do Sherlock N-Overlap com outras ferramentas de detecção de plágio com método próprio de comparação	35
4.1	Sistema de Análise de Similaridade	36
4.1.1	Arquitetura	37
4.1.2	Funcionamento e interface	38
4.1.3	Considerações sobre o uso do sistema de Análise de Similaridade . .	43
4.2	Arcabouço metodológico para avaliação de resultados	44
4.2.1	Método tradicional para comparação das ferramentas	44
4.2.2	Nova abordagem para comparação das ferramentas: o método de conformidade	47
5	Análise de Resultados	53
5.1	Primeira Análise: códigos plagiados propositalmente	53
5.1.1	Cenário de experimentação	54
5.1.2	Descrição do processo de aplicação do método tradicional para obtenção dos dados para análise em códigos gerados artificialmente	55
5.1.3	Análise do parâmetro zerobit do Sherlock	61
5.2	Avaliação conformativa: códigos gerados por alunos em práticas laboratoriais	70
5.2.1	Cenário de experimentação	70
5.2.2	Análise preliminar: uma primeira visão sobre as quantidades de ocorrências, ocorrências isoladas e ausências isoladas	72

5.2.3	Análise com aplicação do método de conformidade	75
6	Conclusão	79
6.1	Publicações/Resultados	81
6.2	Perspectivas futuras	82
	Referências	83
	Apêndice A Enunciados dos problemas plagiados propositalmente	88
	Apêndice B Tabelas: dados brutos	90
	Apêndice C Tabelas: precisão, revocação e média harmônica	97
	Apêndice D Artigos Publicados	111

Lista de Ilustrações

Figura 2.1	Técnicas de Plágio	7
Figura 2.2	Exemplo de código Java e os respectivos <i>tokens</i>	12
Figura 2.3	Algoritmo Greedy String Tiling	13
Figura 2.4	Fluxo de funcionamento do Winnowing	16
Figura 2.5	Trecho de Código com Palavras Sublinhadas	18
Figura 3.1	As 50 palavras mais frequentes do Inglês, conforme BNC corpus.	29
Figura 3.2	Exemplo do funcionamento da proposta de Stamatatos (2011) para comparação de 8-grams.	30
Figura 3.3	Interpretação gráfica do coeficiente Jaccard.	32
Figura 4.1	Arquitetura	37
Figura 4.2	Submissão externa - Página de submissão	39
Figura 4.3	Submissão externa - Página de comparação	39
Figura 4.4	Configurações do Sherlock e Normalização Personalizada	40
Figura 4.5	Interface com execução de um algoritmo	41
Figura 4.6	Interface com execução de dois algoritmos	41
Figura 4.7	Interface com execução de três algoritmos	42
Figura 4.8	Análise de pares: arquivos enviados	43
Figura 4.9	Análise de pares: arquivos normalizado	44
Figura 4.10	Print Screen da tabela de comparação	49
Figura 4.11	Print Screen do relatório associado a Figura 4.10	49
Figura 4.12	Print Screen de relatório parcial de quantificação de ocorrências	50

Figura 4.13	Print Screen de relatório completo de quantificação de ocorrências . . .	51
Figura 5.1	Dados brutos - código médio - programador 1 ($Z=0$)	56
Figura 5.2	Média Harmônica calculada por limiar para os algoritmos SIM, MOSS e JPlag para os códigos médios do programador 1 com $Z=0$	57
Figura 5.3	Média Harmônica calculada por limiar para as versões do Sherlock e para os códigos médios do programador 1 com $Z=0$	58
Figura 5.4	Áreas para código médio - programador 1 ($Z=0$, $>10\%$)	58
Figura 5.5	Áreas para código médio - programador 1 ($Z=0$, $>70\%$)	59
Figura 5.6	Áreas para código médio - programador 1	62
Figura 5.7	Áreas para código médio - programador 2	64
Figura 5.8	Áreas para código médio - programador 3	64
Figura 5.9	Áreas para código pequeno - programador 1	66
Figura 5.10	Áreas para código pequeno - programador 2	67
Figura 5.11	Áreas para código pequeno - programador 3	67
Figura 5.12	Áreas para códigos médios	69
Figura 5.13	Áreas para códigos pequenos	70
Figura 5.14	Turma A - Códigos Reais	76
Figura 5.15	Turma B - Códigos Reais	76
Figura 5.16	Turma C - Códigos Reais	77
Figura 5.17	Turma D - Códigos Reais	77
Figura B.1	Código médio - Programador 1 ($Z=0$)	90
Figura B.2	Código médio - Programador 1 ($Z=1$)	90
Figura B.3	Código médio - Programador 1 ($Z=2$)	90
Figura B.4	Código médio - Programador 1 ($Z=3$)	90
Figura B.5	Código médio - Programador 1 ($Z=4$)	91
Figura B.6	Código médio - Programador 2 ($Z=0$)	91
Figura B.7	Código médio - Programador 2 ($Z=1$)	91
Figura B.8	Código médio - Programador 2 ($Z=2$)	91
Figura B.9	Código médio - Programador 2 ($Z=3$)	91
Figura B.10	Código médio - Programador 2 ($Z=4$)	92
Figura B.11	Código médio - Programador 3 ($Z=0$)	92

Figura B.12	Código médio - Programador 3 ($Z=1$)	92
Figura B.13	Código médio - Programador 3 ($Z=2$)	92
Figura B.14	Código médio - Programador 3 ($Z=3$)	92
Figura B.15	Código médio - Programador 3 ($Z=4$)	93
Figura B.16	Código pequeno - Programador 1 ($Z=0$)	93
Figura B.17	Código pequeno - Programador 1 ($Z=1$)	93
Figura B.18	Código pequeno - Programador 1 ($Z=2$)	93
Figura B.19	Código pequeno - Programador 1 ($Z=3$)	93
Figura B.20	Código pequeno - Programador 1 ($Z=4$)	94
Figura B.21	Código pequeno - Programador 2 ($Z=0$)	94
Figura B.22	Código pequeno - Programador 2 ($Z=1$)	94
Figura B.23	Código pequeno - Programador 2 ($Z=2$)	94
Figura B.24	Código pequeno - Programador 2 ($Z=3$)	94
Figura B.25	Código pequeno - Programador 2 ($Z=4$)	95
Figura B.26	Código pequeno - Programador 3 ($Z=0$)	95
Figura B.27	Código pequeno - Programador 3 ($Z=1$)	95
Figura B.28	Código pequeno - Programador 3 ($Z=2$)	95
Figura B.29	Código pequeno - Programador 3 ($Z=3$)	95
Figura B.30	Código pequeno - Programador 3 ($Z=4$)	96

Lista de Tabelas

Tabela 2.1	Exemplo de seleção de assinaturas por zerobit para sequência 3-gram de palavras seguida da respectiva representação binária das assinaturas	19
Tabela 2.2	Resumo das características gerais	21
Tabela 3.1	Descrição e exemplificação das normalizações	27
Tabela 4.1	Representação dos parâmetros de cálculo para precisão e revocação . .	45
Tabela 4.2	Comparação entre precisão e revocação tradicional e de conformidade	51
Tabela 5.1	Dados contabilizados - código médio - programador 1 (Z=0)	56
Tabela 5.2	Contabilização de assinaturas processadas do código original médio de cada programador	64
Tabela 5.3	Contabilização de assinaturas processadas do código original pequeno de cada programador	67
Tabela 5.4	Relação quantitativa dos códigos analisados	71
Tabela 5.5	Somatório dos pares, classificados dentre os 3 tipos de ocorrências, para os 22 problemas propostos e por turma	73
Tabela C.1	Dados contabilizados - código médio - programador 1 (Z=0)	97
Tabela C.2	Dados contabilizados - código médio - programador 1 (Z=1)	97
Tabela C.3	Dados contabilizados - código médio - programador 1 (Z=2)	98
Tabela C.4	Dados contabilizados - código médio - programador 1 (Z=3)	98
Tabela C.5	Dados contabilizados - código médio - programador 1 (Z=4)	98
Tabela C.6	Dados contabilizados - código médio - programador 2 (Z=0)	99

Tabela C.7	Dados contabilizados - código médio - programador 2 ($Z=1$)	99
Tabela C.8	Dados contabilizados - código médio - programador 2 ($Z=2$)	99
Tabela C.9	Dados contabilizados - código médio - programador 2 ($Z=3$)	100
Tabela C.10	Dados contabilizados - código médio - programador 2 ($Z=4$)	100
Tabela C.11	Dados contabilizados - código médio - programador 3 ($Z=0$)	100
Tabela C.12	Dados contabilizados - código médio - programador 3 ($Z=1$)	101
Tabela C.13	Dados contabilizados - código médio - programador 3 ($Z=2$)	101
Tabela C.14	Dados contabilizados - código médio - programador 3 ($Z=3$)	101
Tabela C.15	Dados contabilizados - código médio - programador 3 ($Z=4$)	102
Tabela C.16	Dados contabilizados - código pequeno - programador 1 ($Z=0$)	102
Tabela C.17	Dados contabilizados - código pequeno - programador 1 ($Z=1$)	102
Tabela C.18	Dados contabilizados - código pequeno - programador 1 ($Z=2$)	103
Tabela C.19	Dados contabilizados - código pequeno - programador 1 ($Z=3$)	103
Tabela C.20	Dados contabilizados - código pequeno - programador 1 ($Z=4$)	103
Tabela C.21	Dados contabilizados - código pequeno - programador 2 ($Z=0$)	104
Tabela C.22	Dados contabilizados - código pequeno - programador 2 ($Z=1$)	104
Tabela C.23	Dados contabilizados - código pequeno - programador 2 ($Z=2$)	104
Tabela C.24	Dados contabilizados - código pequeno - programador 2 ($Z=3$)	105
Tabela C.25	Dados contabilizados - código pequeno - programador 2 ($Z=4$)	105
Tabela C.26	Dados contabilizados - código pequeno - programador 3 ($Z=0$)	105
Tabela C.27	Dados contabilizados - código pequeno - programador 3 ($Z=1$)	106
Tabela C.28	Dados contabilizados - código pequeno - programador 3 ($Z=2$)	106
Tabela C.29	Dados contabilizados - código pequeno - programador 3 ($Z=3$)	106
Tabela C.30	Dados contabilizados - código pequeno - programador 3 ($Z=4$)	107
Tabela C.31	Dados para análise de falsos positivos e negativos - código médio para os 3 programadores ($Z=0$)	107
Tabela C.32	Dados para análise de falsos positivos e negativos - código médio para os 3 programadores ($Z=1$)	107
Tabela C.33	Dados para análise de falsos positivos e negativos - código médio para os 3 programadores ($Z=2$)	108
Tabela C.34	Dados para análise de falsos positivos e negativos - código médio para os 3 programadores ($Z=3$)	108

Tabela C.35	Dados para análise de falsos positivos e negativos - código médio para os 3 programadores ($Z=4$)	108
Tabela C.36	Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores ($Z=0$)	109
Tabela C.37	Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores ($Z=1$)	109
Tabela C.38	Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores ($Z=2$)	109
Tabela C.39	Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores ($Z=3$)	110
Tabela C.40	Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores ($Z=4$)	110

Introdução

Professores, com frequência, buscam apoio em tecnologias de vanguarda para o acompanhamento e gerenciamento das atividades dos alunos. Para executar a gestão de cursos e acompanhar atividades gerais, alguns ambientes virtuais de aprendizagem (AVA), como, por exemplo, o Moodle ([DOUGIAMAS; TAYLOR, 2003](#)), oferecem funcionalidades para a disponibilização de notas de aula, proposta e submissão de trabalhos e registro de notas de atividades. Além disso, muitos AVAs podem ser estendidos com funcionalidades de domínio específico ([SILVA et al., 2011](#)) ([TAVARES et al., 2010](#)) ou mecanismos de avaliação especiais ([SALES; BARROSO; SOARES, 2012](#)).

De especial interesse no contexto deste trabalho, algumas ferramentas, como o [VPL \(2013\)](#), o [Onlinejudge \(2013\)](#) e o BOCALAB ([FRANÇA; SOARES, 2011](#)), são particularmente adequadas para auxiliar nas tarefas de compilação, execução e avaliação de programas de computador desenvolvidos por alunos como solução para práticas laboratoriais em disciplinas de programação propostas pelo professor. Com a automatização de parte do processo de verificação e validação de resultados de atividades de programação, espera-se que o tempo investido em atenção aos alunos por professores e monitores seja aumentado em quantidade e em qualidade.

Entretanto, ao mesmo tempo que este tipo de ferramenta pode contribuir com aspectos de natureza organizacional, como a conferência de resultados em práticas laboratoriais, ele pode também potencializar a cópia de trabalhos entre os alunos por meio de transmissão de códigos via Internet ou por outro mecanismo eletrônico qualquer. Em laboratórios de programação, mesmo em turmas presenciais e sem uso de tecnologias de

apoio para envio e correção de problemas, não é raro encontrar situações de cópia total ou parcial de soluções entre colegas, frequentemente com a mudança de nomes de variáveis ou com a inserção de comentários, visando dificultar a percepção da ação. Em cenários de turmas numerosas, o controle e a detecção deste tipo de conduta se torna ainda mais difícil.

Este trabalho se contextualiza no problema da detecção de plágio entre códigos-fonte em turmas de programação. Entretanto, embora a detecção de plágio seja um aspecto importante no referido contexto, os estudos e as experiências realizadas ao longo deste trabalho revelaram que os resultados da comparação entre códigos podem indicar também outros aspectos que são de natureza irrepreensível. Ou seja, dependendo da perspectiva e do contexto da atividade, a semelhança entre soluções pode ser indicativa de parceria, de trabalho colaborativo, de referência ou de apoio sobre solução encontrada em livros ou exemplos fornecidos pelo professor, entre outros motivos. Por essa razão, ao invés de “detecção de plágio”, adota-se prioritariamente neste trabalho a expressão “análise de similaridade”.

Krokosz (2011) investigou como as maiores universidades de cada continente, incluindo as do Brasil, tratam o plágio entre alunos, sob vários critérios, concluindo que, no Brasil, estamos atrasados em termos de medidas preventivas, de adoção de código de ética e apoio institucional para lidar com esse tipo de problema. Esse tipo de verificação decorre apenas de iniciativas isoladas de alguns professores, sendo, portanto, a prática de copiar trabalhos ilicitamente banalizada, com prejuízos na formação acadêmica e pessoal.

Tendo em vista que a proposição de atividades em meio digital para turmas numerosas facilita a cópia de soluções entre alunos, torna-se útil o emprego de meios computacionais para desmotivar essa postura, quando não autorizada, ou para acompanhar eventuais trocas de experiências, quando for o caso. Recursos que permitam esse tipo de verificação devem estar ao alcance do professor de maneira eficiente e prática, de preferência, funcionando conjuntamente com os sistemas que permitem a proposição e a verificação automatizadas das soluções desenvolvidas pelos alunos.

O desenvolvimento das tecnologias associadas a este trabalho foi realizado como desdobramento do módulo de análise de similaridade do ambiente BOCALAB (FRANÇA; SOARES, 2011), que tem como finalidade principal a de atender os requisitos do fluxo:

submissão, compilação, execução, avaliação e *feedback* das atividades de modo remoto e distribuído. A detecção de plágio foi proposta e integrada de maneira preliminar na primeira versão do BOCALAB, usando-se, para isso, o algoritmo Sherlock (PIKE; LOKI, 2013). A escolha dessa solução como técnica para a identificação de similaridade foi devido ao fato de ser o Sherlock uma ferramenta livre, de código aberto e de propósito geral, o que permite para um mesmo executável, comparar códigos em diversas linguagens. Embora para a construção de sua versão original, não tenham sido realizados estudos mais aprofundados sobre a detecção de plágio.

Assim, iniciou-se o presente trabalho com um estudo da sensibilidade do algoritmo Sherlock para códigos submetidos a técnicas de pré-processamento. Tais técnicas visam a normalização dos códigos-fonte antes da comparação efetiva, buscando remover os aspectos irrelevantes e destacar os importantes, permitindo, assim, uma comparação mais dirigida e objetiva. Além disso, estudou-se as métricas comumente utilizadas por algoritmos de comparação e detecção de plágio, buscando-se propor melhorias ao Sherlock original.

Por fim, os resultados do algoritmo Sherlock modificado, incorporando técnicas de normalização de código, foram comparados com os obtidos com o uso do JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2002), do MOSS (SCHLEIMER; WILKERSON; AIKEN, 2003) e do SIM (GRUNE, 2013b), ferramentas amadurecidas e destacadas na literatura atual sobre detecção de plágio entre códigos-fonte.

As investigações e experimentações realizadas visam o desenvolvimento de uma solução para permitir ao professor identificar pares de alunos que possuem fortes indícios de trabalho em colaboração ou de plágio.

1.1 Problematização

A detecção de plágio (e a análise de similaridade) em código-fonte tem sido estudada em diversos trabalhos, tais como Hage, Rademaker e Vugt (2010), Bin-Habtoor e Zaher (2012) e Duric e Gasevic (2012). Algumas ferramentas foram disponibilizadas especialmente para este fim, como visto em Prechelt, Malpohl e Philippsen (2002) e Schleimer, Wilkerson e Aiken (2003), encontrando-se, não raramente, avaliações comparativas entre elas. Apesar da ampla quantidade de ferramentas disponíveis, devido à complexidade inerente à análise

de similaridade entre códigos-fonte, poucas delas são capazes de identificar de maneira eficaz todas as semelhanças léxicas e semânticas entre pares de códigos. [Cosma e Joy \(2006\)](#) endossam essa visão, afirmando que o valor de similaridade entre arquivos e fragmentos de códigos é uma questão subjetiva, sendo as divergências de resultados acentuadas mesmo entre as ferramentas mais bem conceituadas, o que evidencia limitações em quase todas as técnicas de detecção de similaridade. Tem-se neste tema, portanto, um assunto de pesquisa em aberto.

O principal problema sob investigação constitui-se da pesquisa de técnicas bem como a proposição de um ambiente computacional que favoreça e facilite ao professor a análise de similaridade entre códigos-fonte, tanto para coibir a prática do plágio, como para permitir, de alguma maneira, avaliar o comportamento e o aprendizado dos alunos por meio da comparação de suas soluções para os problemas propostos.

Entretanto, devido à densidade do problema da detecção de plágio, esta pesquisa não aborda problemas de natureza pedagógica, concentrando-se prioritariamente na proposição de uma ferramenta para análise de similaridade entre códigos-fonte que seja aplicável, da maneira mais eficiente possível, no tipo de código-fonte que compõe o domínio específico das práticas laboratoriais de programação propostas em cursos de Engenharias e de Ciência da Computação. Espera-se, portanto, que o todo ou parte do *software* resultante deste trabalho seja reintegrado ao BOCALAB ou possa contribuir com outros ambientes virtuais que apoiem o professor em suas práticas acadêmicas.

1.2 Objetivos

Objetivos Gerais

Avaliar, conceber e desenvolver técnicas e ferramenta para viabilizar a verificação de similaridade entre códigos de alunos de programação de maneira eficaz.

Objetivos Específicos

São objetivos específicos:

- Estudar e avaliar algoritmos que possam ser utilizados para a construção de ferramentas de apoio à análise de similaridade entre códigos-fonte submetidos por

alunos como solução a problemas propostos, identificando aos mais adequados ao problema tratado nesta dissertação;

- ▶ Estudar e propor técnicas de pré-processamento que sejam capazes de preparar o código, removendo informações irrelevantes, ou normalizando estruturas, visando destacar as características importantes para o código a ser comparado;
- ▶ Estudar métodos para avaliar os resultados obtidos por ferramentas de detecção de plágio com o uso técnicas de pré-processamento;
- ▶ Construir um ambiente computacional para a submissão de conjuntos de código-fonte para análise de similaridade, bem como a configuração de múltiplas ferramentas de detecção de plágio, permitindo a visualização e comparação dos resultados obtidos. O ambiente deverá ser aberto e viabilizar a adaptação de novos algoritmos de comparação e novas técnicas de processamento.

1.3 Organização do texto

A dissertação está organizada em 6 capítulos. No [Capítulo 2](#), apresenta-se a fundamentação teórica desta dissertação. Discutem-se os principais conceitos relativos a técnicas de plágio, ao funcionamento de algoritmos e de ferramentas para análise de similaridade e detecção de plágio, assim como os principais trabalhos relacionados.

No [Capítulo 3](#) expõe-se a modificação empregada no algoritmo Sherlock e descrevem-se também as regras de normalização concebidas como técnicas de pré-processamento para uso do algoritmo Sherlock modificado. No [Capítulo 4](#), disserta-se sobre a ferramenta desenvolvida para apresentar os resultados do Sherlock com as normalizações, bem como para comparar os resultados com os de outras abordagens. No mesmo capítulo apresenta-se o método de conformidade que foi desenvolvido para a análise dos resultados levando em consideração a convergência de resultados de múltiplas ferramentas.

No [Capítulo 5](#), apresentam-se os resultados alcançados com dois testes: um com códigos gerados manualmente e outro com códigos gerados por alunos.

Por fim, no [Capítulo 6](#) são realizadas as considerações finais, bem como as perspectivas de trabalhos para a continuidade deste projeto de pesquisa.

Fundamentação Teórica

Neste capítulo, são apresentados os principais conceitos que envolvem a área de análise de similaridade em código fonte.

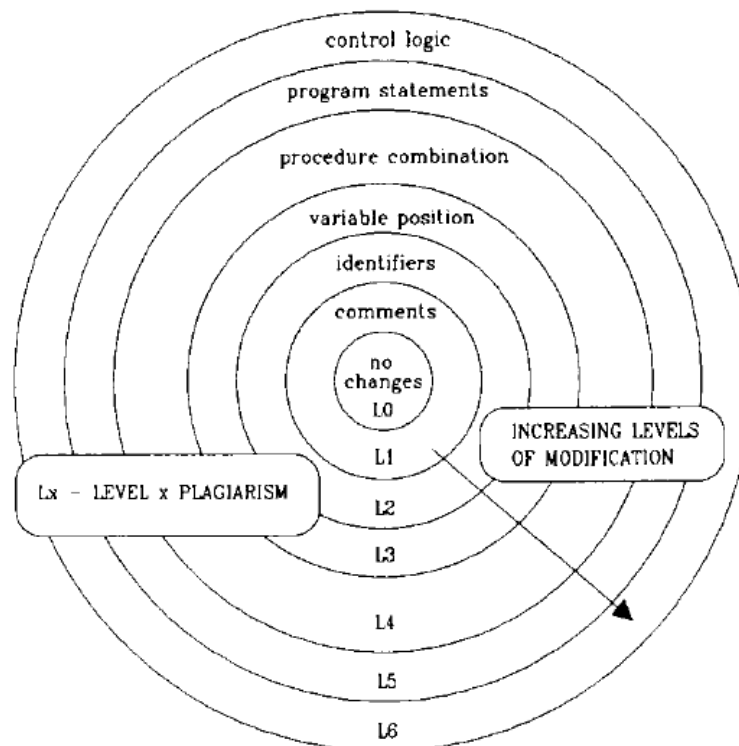
Antes de explorar o funcionamento das ferramentas para detecção de plágio ou análise de similaridade, discute-se sobre os meios mais comuns empregados para realizar o plágio. Em seguida, apresentam-se as ferramentas de detecção de plágios mais utilizadas, bem como alguns coeficientes de similaridades empregados por algoritmos de comparação.

2.1 Técnicas de plágio

A capacidade de plagiar códigos em programação aumenta de maneira proporcional ao conhecimento do aluno sobre essa disciplina. A experiência em sala de aula revela que, nas primeiras semanas do aprendizado de programação, não é raro encontrar códigos com nomes de variáveis que são, ao mesmo tempo, pouco usuais e idênticos, revelando certa ingenuidade na tentativa de dissimular a cópia. Com o amadurecimento e maior domínio da linguagem de programação, os alunos são desafiados com problemas mais complexos, também se tornando mais difícil identificar o nível de similaridade entre pares de código. A minoria dos alunos que possuem a intenção de copiar ilicitamente o trabalho dos colegas, o fazem principalmente por falta de tempo ou por incapacidade técnica de resolver o problema proposto (JOY; LUCK, 1999). Pelos motivos expostos, não são esperadas modificações muito elaboradas em códigos plagiados.

Em programação, define-se Técnicas de Plágio, ou Padrões de Plágio, como as possíveis técnicas utilizadas para encobrir a detecção do plágio. [Faidhi e Robinson \(1987\)](#) estão entre os primeiros autores a caracterizar essas modificações, ilustrando-as conforme a [Figura 2.1](#), que enumera níveis de plágio de acordo com o tipo de modificação empregada. O primeiro nível corresponde a códigos sem modificação e o último refere-se ao tipo de modificação mais complexa: alteração lógica.

Figura 2.1 – Técnicas de Plágio



Fonte: [Faidhi e Robinson \(1987\)](#)

De acordo com [Whale \(1990\)](#), as técnicas de modificações mais empregadas são:

- i. Alteração de comentários e/ou formatação;
- ii. Modificação de nomes de identificadores;
- iii. Alteração da ordem de operandos e expressões;
- iv. Alteração de tipos de dados;
- v. Substituição de expressões por equivalentes;

- vi. Adição de instruções redundantes ou variáveis (por exemplo, inicialização desnecessária);
- vii. Alteração na ordem de instruções que não alteram o funcionamento;
- viii. Alteração das estruturas de *loop* (exemplos: “repeat” por “while”; “while” por “for”);
- ix. Alteração das estruturas das instruções de seleção (linearizar “ifs” cascadeados; trocar “ifs” por “switch-case”);
- x. Substituição de chamadas a funções pelo respectivo conteúdo;
- xi. Adicionar instruções que não influenciam o fluxo do programa (por exemplo, funções de impressão);
- xii. Combinação de código copiado com código original.

Joy e Luck (1999) classificaram as alterações quanto às características léxicas e estruturais. As modificações léxicas são aquelas que não dependem da estrutura sintáticas de uma linguagem. As mudanças estruturais, por outro lado, estão associadas às regras de sintaxe de uma linguagem específica.

Segundo eles, as modificações léxicas são:

- i. Reescrita, adição ou omissão de comentários;
- ii. Alteração de formatação;
- iii. Modificação de nomes de identificadores;
- iv. Alteração do número de linhas, para linguagens como o FORTRAN.

E as estruturais são:

- i. Substituição de estruturas de *loops* (por exemplo, “for” por “while”);
- ii. Substituição de “ifs” cascadeados por “switch-case”;
- iii. Alteração na ordem de instruções que não afetam o funcionamento do programa;

- iv. Substituição de múltiplas chamadas de procedimentos por chamadas de uma função única, ou *vice-versa*;
- v. Substituição de chamada de procedimento pelo conteúdo do procedimento;
- vi. Alteração na ordem de operandos (por exemplo “ $x < y$ ” por “ $y > x$ ”).

Mais recentemente, [Mozgovoy \(2006\)](#) também elaborou uma lista de transformações para esconder o plágio:

- i. Alteração de comentários (reescrita, adição, alteração se sintaxe e omissão);
- ii. Alteração de espaços em branco e layout;
- iii. Modificação de nomes de identificadores;
- iv. Reordenação de blocos de código;
- v. Reordenação de instruções dentro de blocos de códigos;
- vi. Alteração na ordem de operadores/operandos em expressões;
- vii. Mudança de tipos de dados;
- viii. Adição de instruções redundantes ou variáveis;
- ix. Substituição de estruturas de controle por equivalentes (“**while**” por “**do-while**”, “**ifs**” por “**switch-case**”);
- x. Substituição da chamada a uma função pelo conteúdo da mesma.

Percebe-se que não existe uma grande variação entre as principais técnicas defendidas por cada autor. Para o contexto deste trabalho, é importante reunir uma seleção dessas técnicas, de forma a elaborar conjuntos de teste com códigos que representem as principais modificações, segundo a literatura.

Para as análises de similaridade realizadas sobre conjuntos de códigos controlados neste trabalho, modificações foram realizadas sobre códigos originais inspirando-se sobre um subconjunto de ações elencadas nessa seção. Para isso, levou-se em consideração o contexto no qual se realizou a análise, especificamente em turmas de primeiro ano de

programação. As ações utilizadas para a produção dos códigos controlados são apresentadas na [subseção 5.1.1](#), juntamente com o cenário de experimentação.

2.2 Algoritmos, Ferramentas e Sistemas para detecção de plágio

Os algoritmos, ferramentas e sistemas de detecção de plágio variam não só relativamente à lógica de comparação, mas também em termos de disponibilidade de código e restrições quanto a linguagens de programação aceitas.

Neste trabalho, entende-se por algoritmo o fluxo lógico utilizado para a realização da comparação. Por ferramenta entende-se a implementação de um algoritmo com otimizações e adequações a uma linguagem de programação específica. Por sistema, entende-se a utilização de uma ferramenta com acréscimo de funcionalidades, como credencial de acesso, recursos de submissão de arquivos para comparação, renderização dos resultados e armazenamento de resultados anteriores. Dessa forma, a rigor, o Sherlock é uma ferramenta com algoritmo próprio, RKR-GST ([WISE, 1996](#)) é o algoritmo presente na ferramenta Yap3 ([WISE, 1996](#)) e com implementação adaptada no sistema JPlag ([PRECHELT; MALPOHL; PHILIPPSEN, 2002](#)), com variações de otimização. Por motivos de simplificação, utiliza-se o termo ferramenta de maneira genérica, em especial no momento da análise de resultados, para identificar a solução computacional utilizada para medir a similaridade entre pares de códigos-fonte.

Diversas ferramentas podem ser utilizadas para realizar a análise de similaridade (ou detecção de plágio) entre códigos-fonte, tais como SIM ([GRUNE, 2013b](#)), YAP ([WISE, 1996](#)), JPlag ([PRECHELT; MALPOHL; PHILIPPSEN, 2002](#)), SID (Chen et al., 2004), Plaggie (Ahtiainen, 2006) e MOSS ([BOWYER; HALL, 1999](#)) ([SCHLEIMER; WILKERSON; AIKEN, 2003](#)). Estudos comparativos sobre algumas destas ferramentas foram realizados por [Hage, Rademaker e Vugt \(2010\)](#), [Kleiman \(2007\)](#) e [Green et al. \(2012\)](#). De modo geral, ferramentas especificamente construídas para esta finalidade permitem encontrar semelhanças no caso de alterações de nomes de variáveis, nomes de funções, comentários ou, ainda, alterando-se a ordem de partes do código. Segundo [Burrows, Tahaghoghi e Zobel \(2007\)](#), JPlag e MOSS são as mais importantes e mais citadas ferramentas para detecção de plágio, sendo, por isso, abordadas com maiores detalhes, seguidos pelo Sherlock, sobre o qual se apóiam as contribuições deste trabalho.

Dos algoritmos que serão discutidos a seguir, o JPlag, o MOSS, o SIM e o Sherlock foram integrados à ferramenta de análise apresentada no [Capítulo 4](#).

2.2.1 YAP3

O Yap3 ([WISE, 1996](#)) consiste na 3a versão de uma série de sistemas para detecção de plágio em código fonte. O YAP, primeira versão, utiliza a ferramenta *sdiff*, que é similar ao *diff*, o Yap2 utiliza o algoritmo de Heckel ([HECKEL, 1978](#)) e, por fim, o Yap3 utiliza o algoritmo Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) ([WISE, 1996](#)).

Devido ao fato de o JPlag, algoritmo apresentado na próxima subseção, ser baseado no YAP3, este não foi integrado ao conjunto de ferramentas exploradas neste trabalho.

2.2.2 JPlag

O JPlag ([PRECHELT; MALPOHL; PHILIPPSEN, 2002](#)) é uma ferramenta desenvolvida em Java para análise de similaridade entre códigos-fonte. É disponibilizada exclusivamente por meio de um *webservice* e seus detalhes de implementação não são completamente divulgados. De acordo com [Prechelt, Malpohl e Philippsen \(2002\)](#), sabe-se que o funcionamento central do JPlag é baseado no mesmo algoritmo presente no YAP3, o RKR-GST com otimizações.

O uso do sistema é livre, mas para ter acesso ao serviço, é necessário requisitar autorização. A submissão de arquivos para comparação é realizada por meio de um *applet* Java, disponibilizado pelos desenvolvedores. Como alternativa, existe ainda a possibilidade de desenvolver um cliente próprio para o acesso ao *webservice*. As instruções de cadastro e acesso podem ser obtidas no endereço *web*¹ do sistema.

O JPlag foi originalmente concebido para encontrar pares similares entre conjuntos de arquivos de códigos fonte, especificamente nas linguagens Java, C#, C, C++ e Scheme. Também é possível comparar arquivos de texto em linguagem natural.

Sendo um serviço remoto, ao enviar um conjunto de códigos, a requisição entra em uma fila de espera, sendo o desempenho dependente do volume de concorrência. O JPlag

¹ <http://jplag.ipd.kit.edu/>

apresenta o resultado da comparação em forma de arquivos HTML, que pode ser baixado ou visualizado *online*.

Segundo [Prechelt, Malpohl e Philippsen \(2002\)](#), o algoritmo do JPlag funciona em duas etapas:

- i. Os códigos fontes enviados para comparação são submetidos a um analisador que interpreta as estruturas da linguagem e gera *tokens* correspondentes;
- ii. Os *tokens* gerados na primeira etapa são comparados par a par, sendo, na sequência, calculado um índice de similaridade por par.

Figura 2.2 – Exemplo de código Java e os respectivos *tokens*

Java source code	Generated tokens
1 public class Count {	BEGIN_CLASS
2 public static void main(String[] args)	VAR_DEF, BEGIN_METHOD
3 throws java.io.IOException {	
4 int count = 0;	VAR_DEF, ASSIGN
5	
6 while (System.in.read() != -1)	APPLY, BEGIN_WHILE
7 count++;	ASSIGN, END_WHILE
8 System.out.println(count+" chars.");	APPLY
9 }	END_METHOD
10 }	END_CLASS

Fonte: [Prechelt, Malpohl e Philippsen \(2002\)](#).

O analisador utilizado na primeira etapa é dependente da linguagem. Assim, o JPlag está limitado às linguagens que possuem o respectivo analisador implementado. Na [Figura 2.2](#) é apresentado um exemplo de tradução em tokens para um código Java.

A fase de comparação é baseada no algoritmo Greedy String Tiling ([WISE, 1993](#)), o qual tem como objetivo encontrar, para duas cadeias de caracteres (dois códigos), o conjunto com substrings contínuas que seja: (i) a maior possível, (ii) comum aos dois códigos e (iii) que já não tenha sido coberta por outra substring. O algoritmo que corresponde a essa lógica é apresentado na [Figura 2.3](#). Nessa figura, conforme apresentado por [Prechelt, Malpohl e Philippsen \(2002\)](#), a função $match(a, b, l)$ representa a associação entre as substrings idênticas de A e B , que são iniciadas nas posições A_a e B_b , respectivamente, de tamanho l . No fim, o algoritmo retorna uma lista de ladrilhos (*tiles*), que correspondem às substrings em comum necessárias para encontrar a similaridade $sim(A, B)$, que é calculada

Figura 2.3 – Algoritmo Greedy String Tiling

```

0  Greedy-String-Tiling(String A, String B) {
1      tiles = {};
2      do {
3          maxmatch = M;
4          matches = {};
5          Forall unmarked tokens Aa in A {
6              Forall unmarked tokens Bb in B {
7                  j = 0;
8                  while (Aa+j == Bb+j &&
9                      unmarked(Aa+j) && unmarked(Bb+j))
10                     j ++;
11                  if (j == maxmatch)
12                     matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                     matches = {match(a, b, j)};
15                     maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 . . . (maxmatch - 1) {
21                  mark(Aa+j);
22                  mark(Bb+j);
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > M);
27      return tiles;
28  }

```

Fonte: [Prechelt, Malpohl e Philippsen \(2002\)](#).

como uma fração da quantidade de *tokens* comuns aos dois códigos em relação à quantidade total de *tokens*, conforme [Equação 2.1](#).

$$sim(A, B) = 2 \times \frac{\sum_{match(a,b,length) \in tiles} length}{(|A| + |B|)} \quad (2.1)$$

De acordo com [Wise \(1993\)](#), o Greedy String Tiling obtém melhoras significativas de performance ao incorporar ideias do algoritmo Karp-Rabin ([KARP; RABIN, 1987](#)). A ideia central do algoritmo Karp-Rabin ([KARP; RABIN, 1987](#)) é encontrar todas as ocorrências de uma determinada string P em um texto T , a partir de uma função *hash*. O funcionamento se baseia na característica da função *hash* de representar grande quantidade de informação em um formato conciso. Assim, em lugar de comparar a informação original, comparam-se os respectivos valores *hashs*. Dessa forma, para procurar em T a string P de

tamanho s , gera-se um *hash* de todas as substrings de tamanho s em T . Compara-se, em seguida, o *hash* de P com todos os *hash* obtidos de T .

O algoritmo originalmente proposto por [Wise \(1993\)](#), baseado no Karp-Rabin, busca obter a maior cadeia possível e calcula o *hash* com base em uma cadeia de comprimento variável, cujo valor inicial é igual a M . Quando um casamento de assinaturas é descoberto, o algoritmo busca ampliar a cadeia encontrada, o que gera um novo cálculo de assinaturas e é bastante custoso. [Prechelt, Malpohl e Philippsen \(2002\)](#), no entanto, fixam o comprimento das cadeias com tamanho M . Assim, o JPlag calcula todos os *hashs* apenas uma vez antes do processo de comparação, enquanto a versão anterior recalcula todos a cada iteração. Nota-se que para cada *hash* calculado, armazena-se também a respectiva posição referente ao texto original. Dessa forma é possível identificar no texto original as substrings que o algoritmo considerou iguais.

Outras informações e análises sobre o funcionamento do JPlag podem ser encontradas nos trabalhos de [Kleiman \(2007\)](#), [Burrows, Tahaghoghi e Zobel \(2007\)](#), [Hage, Rademaker e Vugt \(2010\)](#).

2.2.3 MOSS

O MOSS (*Measure of Software Similarity*) ([BOWYER; HALL, 1999](#)) ([SCHLEIMER; WILKERSON; AIKEN, 2003](#)), à semelhança do JPlag, também é acessado exclusivamente por meio de um *webservice*, disponibilizado na Universidade da Califórnia, necessitando-se requisitar autorização para usá-lo. A resposta desta requisição contém um *script perl*, por meio do qual as submissões para comparação devem ser realizadas. Após a submissão, é gerada uma URL que fornece o resultado da comparação, que permanece disponível para consulta por 14 dias. O MOSS funciona para programas escritos nas linguagens C, C++, Java, Pascal, Ada, LISP, entre outras.

Apesar de ter sido desenvolvido em 1994 por Alex Aiken, somente em 2003, [Schleimer, Wilkerson e Aiken \(2003\)](#) apresentaram alguns detalhes sobre o algoritmo que emprega a técnica Winnowing [Schleimer, Wilkerson e Aiken \(2003\)](#). O funcionamento interno do MOSS, entretanto, ainda é confidencial. Não se sabe, por exemplo, como é calculado o valor de similaridade entre dois códigos ([KLEIMAN, 2007](#)). De acordo com [Kleiman \(2007\)](#), que a ausência de detalhes sobre o funcionamento do MOSS é para

dificultar o descobrimento de formas de burlar o algoritmo. A confidencialidade também pode ser justificada como garantia de segredos comerciais, já que, para utilização local ou comercial do MOSS, é necessário contato com a empresa Similix Corporation.

No MOSS, sabe-se que o código fonte é convertido em *tokens*, e o WInnowing é utilizado para selecionar um subconjunto de *hashs*, relativo aos *tokens* gerados, visando criar um tipo de assinatura (*fingerprint*), que se constitui de uma sequência de valores *hashs* que representa a estrutura do documento. O cálculo da distância entre os *fingerprints* determina o quanto os arquivos correspondentes são similares (SCHLEIMER; WILKERSON; AIKEN, 2003) (APIRATIKUL, 2004) (MOZGOVOY, 2006).

Na Figura 2.4, Schleimer, Wilkerson e Aiken (2003) representam o funcionamento do WInnowing. Na Figura 2.4(b) alguns aspectos do código são removidos a fim de eliminar ruídos. Em seguida, o texto é dividido em sequências de substrings com 5-grams, Figura 2.4(c). Para cada substring é gerado um valor *hash*, como exemplificado na Figura 2.4(d). Todos esses *hashs* são distribuídos em janelas de tamanho 4, como na Figura 2.4(e). Em seguida, os menores *hashs* de cada janela são selecionados para compor a *fingerprint* representativa do documento Figura 2.4(f). É importante observar que os valores de 5-gram e a janela de tamanho 4 são hipotéticos, pois não se sabe como e quais desses valores são efetivamente utilizados no MOSS.

Outras informações sobre o MOSS podem ser encontradas nos trabalhos de Kleiman (2007), Apiratikul (2004), Hage, Rademaker e Vugt (2010).

2.2.4 SIM

O SIM (GRUNE, 2013b) (GRUNE; VAKGROEP, 1989) foi desenvolvido na Universidade de Amsterdã em 1989 por Dick Grune, autor do versionador de arquivos CVS. A última versão (SIM 2.77) é de 2012 e está disponível² para *download* dos códigos-fontes e dos binários. É gerado um binário para cada linguagem suportada: C, Java, Pascal, Modula-2, Lisp, Miranda, e linguagem natural.

A única documentação encontrada sobre o funcionamento do SIM é o próprio código fonte e uma descrição resumida realizada pelo autor (GRUNE, 2013a). É explicitado que cada arquivo é dividido em *tokens* de acordo com a linguagem do arquivo de entrada. Os

² http://dickgrune.com/Programs/similarity_tester/

Figura 2.4 – Fluxo de funcionamento do Winkowing

A do run run run, a do run run

(a) Some text.

adorunrunrunadorunrun

(b) The text with irrelevant features removed.

adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun

(c) The sequence of 5-grams derived from the text.

77 74 42 17 98 50 17 98 8 88 67 39 77 74 42
17 98

(d) A hypothetical sequence of hashes of the 5-grams.

(77, 74, 42, 17)	(74, 42, 17, 98)
(42, 17, 98, 50)	(17, 98, 50, 17)
(98, 50, 17, 98)	(50, 17, 98, 8)
(17, 98, 8, 88)	(98, 8, 88, 67)
(8, 88, 67, 39)	(88, 67, 39 , 77)
(67, 39, 77, 74)	(39, 77, 74, 42)
(77, 74, 42, 17)	(74, 42, 17, 98)

(e) Windows of hashes of length 4.

17 17 8 39 17

(f) Fingerprints selected by winnowing.

Fonte: adaptado de [Schleimer, Wilkerson e Aiken \(2003\)](#).

tokens são armazenados em um *array* que é realocado quando ocorre *overflow*. Em outro *array*, são armazenadas as posições referentes ao começo da *string* a qual *token* corresponde. Representadas por *tokens*, cada *substring* é comparada com todas as *substrings* a sua direita. Para acelerar o processo, são consideradas apenas as *substrings* de um tamanho mínimo, determinado, enquanto as demais são descartadas. A partir dos *hashs* são geradas duas tabelas: uma para armazenar as posições (A_n) das *substrings* e a outra os elementos (B_n) que contém os *hashs* das respectivas *substrings*. Assim, um índice A_n representa a posição no texto da *substring* cujo *hash* refere-se ao índice B_n . Essa correspondência entre as duas tabelas, em termos da posição das *substrings* iguais, é necessária para quantificar as ocorrências de emparelhamento combinados por número de linhas. O autor sugere que

a similaridade seja calculada em função das *substrings* emparelhadas e da quantidade de linhas por arquivo.

É importante diferenciar o SIM de outra ferramenta com mesmo nome ([GITCHELL; TRAN, 1999](#)). Apesar de referências que tratam da disponibilidade do código ([KLEIMAN, 2007](#)), não foi possível encontrá-lo.

2.2.5 Sherlock

É necessário diferenciar os homônimos Sherlock de [Pike e Loki \(2013\)](#) e de [Joy e Luck \(1999\)](#). O Sherlock ([PIKE; LOKI, 2013](#)) utilizado neste trabalho é de Pike e Loki, o mesmo empregado por [Hage, Rademaker e Vugt \(2010\)](#) e [Zakova, Pistej e Bistak \(2013\)](#).

O Sherlock ([PIKE; LOKI, 2013](#)) é uma alternativa às ferramentas apresentadas precedentemente por ser de código aberto, permitindo modificações e melhorias. Além disso, foi possível implantá-lo no BOCALAB, sem exigir conectividade com serviços de terceiros, o que representaria considerável desvantagem em termos tempo de resposta. Desenvolvido em linguagem C, apresenta bom desempenho e realiza a análise de semelhança léxica entre documentos textuais. Para encontrar trechos duplicados de cada documento, é gerada uma assinatura digital calculando valores *hash* para sequência de palavras. Ao final, comparam-se uma amostragem das assinaturas geradas e identifica-se o percentual de semelhança.

O Sherlock se constitui da composição de dois programas: o Sig e o Comp, criados por Rob Pike, engenheiro de software canadense. O programa Sig gera assinaturas digitais que representam cada código-fonte analisado. O programa Comp, a partir das assinaturas geradas pelo Sig, calcula as similaridades entre os arquivos. Posteriormente, Loki combinou os dois programas em um único, denominado Sherlock. A assinatura digital é a representação de uma estrutura retirada do código, o que enquadra o Sherlock na abordagem de comparação de estruturas. O processo de detecção da ferramenta pode ser resumido em duas etapas: geração de assinaturas e comparação de assinaturas. As seções seguintes descrevem melhor essas duas etapas.

Geração de assinaturas

O início dessa etapa se dá pela extração de características, que retira do código-fonte estruturas menores chamadas de palavras. Uma palavra é qualquer substring delimitada por caracteres de espaço, tabulação ou nova linha. O trecho de código da [Figura 2.5](#) tem as palavras sublinhadas.

Figura 2.5 – Trecho de Código com Palavras Sublinhadas

```
int sum(int *v, int n) {
    int i, result = 0;
    for (i = 0; i < n; i++) {
        result += v[i];
    }
    return result;
}
```

Fonte: [Camargos \(2013\)](#).

Na sequência, um conjunto de palavras é convertido em assinatura por meio da função *hash* apresentada na [Equação 2.2](#). As assinaturas são uma representação numérica do conteúdo do texto a ser posteriormente comparado.

$$h(K) = K \bmod M \quad (2.2)$$

O inteiro K , que representa um *token*, é mapeado em um valor inteiro dentro do intervalo $[0, M - 1]$, onde $M = 2^{32}$. O inteiro K é obtido com o somatório:

$$K = \sum_{i=0}^{n-1} t[i] \times 31^{n-1-i} \quad (2.3)$$

em que n é o número de caracteres no *token* e $t[i]$ corresponde à representação ASCII do i -ésimo caractere. No final dessa etapa, o código-fonte é representado por um vetor de assinaturas selecionadas. Essa seleção ocorre de acordo com o parâmetro zerobit, que indica a quantidade de zeros que deve existir ao final da representação binária da assinatura a ser selecionada.

Na [Tabela 2.1](#) exemplifica-se o processo de seleção de assinaturas para um documento com o texto “Sherlock N-Overlap: Normalização invasiva e coeficiente

de sobreposição para análise de similaridade entre códigos-fonte em disciplinas de programação”. Com o parâmetro número de palavras igual a três, é gerada uma assinatura para cada sequência 3-gram. Para zerobit igual a zero, nenhuma assinatura é descartada. Para zerobit igual a 2, somente são selecionadas as assinaturas que contenham ao menos 2 zeros consecutivos no final da representação binária.

Tabela 2.1 – Exemplo de seleção de assinaturas por zerobit para sequência 3-gram de palavras seguida da respectiva representação binária das assinaturas

Sequência 3-gram de palavras/representação binária da assinatura			
Seleção por zerobit	zerobit = 0	zerobit = 1	zerobit = 2
Sherlock N-Overlap: Normalização - 1101110100000110001101000001	1101110100000110001101000001		
N-Overlap: Normalização invasiva - 10010010110001100011001001000101	10010010110001100011001001000101		
Normalização invasiva e - 1100001101110101001111010010010	1100001101110101001111010010010	1100001101110101001111010010010	
invasiva e coeficiente - 101000001000101000001001000	101000001000101000001001000	101000001000101000001001000	101000001000101000001001000
e coeficiente de - 100011111000100101111001111000	100011111000100101111001111000	100011111000100101111001111000	100011111000100101111001111000
coeficiente de sobreposição - 100101001000010111011111011010	100101001000010111011111011010	100101001000010111011111011010	
de sobreposição para - 100000010111100101001110111110	100000010111100101001110111110	100000010111100101001110111110	
sobreposição para análise - 100001010000011011110001110111	100001010000011011110001110111		
para análise de - 1110001010001011110001000011011	1110001010001011110001000011011		
análise de similaridade - 1001011001000100000110001010111	1001011001000100000110001010111		
de similaridade entre - 11100001001001011100011100001	11100001001001011100011100001		
similaridade entre códigos-fonte - 10100110100110100111110	10100110100110100111110	10100110100110100111110	
entre códigos-fonte em - 1000010001010101001111110000010	1000010001010101001111110000010	1000010001010101001111110000010	
códigos-fonte em disciplinas - 10110001000100101000000011100011	10110001000100101000000011100011		
em disciplinas de - 100110100100110100001110110100000	100110100100110100001110110100000	100110100100110100001110110100000	100110100100110100001110110100000
disciplinas de programação - 1100011110100111111110011100110	1100011110100111111110011100110	1100011110100111111110011100110	

A próxima etapa utiliza esses vetores de assinaturas, selecionadas por zerobit, para comparações.

Comparação de assinaturas

Na etapa de comparação, as assinaturas que identificam um código-fonte são comparadas às assinaturas de outro até que todos os códigos analisados sejam comparados, exigindo um número de comparações igual à combinação sem repetição de n códigos 2 a 2, comparando assim todos os pares de código, conforme a [Equação 2.4](#).

$$C_2^n = \frac{n \times (n - 1)}{2} \quad (2.4)$$

A similaridade é definida como a porcentagem de semelhança entre as assinaturas correspondentes aos dois arquivos A e B que são comparados, como formulado na equação [Equação 2.5](#).

$$SIM(A, B) = 100 \times \frac{a}{(l_1 + l_2 - a)} = 100 \times \frac{a}{(a + b + c)} \quad (2.5)$$

em que

$$l_1 = \text{length}(A) = a + b \quad (2.6)$$

$$l_2 = \text{length}(B) = a + c \quad (2.7)$$

sendo a é o número de assinaturas similares encontradas em ambos os arquivos (A e B), b é o número de assinaturas encontradas exclusivamente em A e c o número de assinaturas encontradas exclusivamente em B .

É possível observar que o cálculo final empregado pelo Sherlock (Equação 2.5) é idêntico ao coeficiente de Jaccard (Equações 2.8 e 2.9). Os coeficientes de similaridade são discutidos na seção 2.3.

Parâmetros do Sherlock

O Sherlock possui três parâmetros que influenciam diretamente o seu comportamento, são eles:

- i. Zerobits (z): controla a granularidade da comparação. Esse controle atua na seleção de assinaturas que são comparadas. Quando zerobit = 0, todas as assinaturas geradas serão comparadas. Quanto menor o número, mais exata a comparação, porém mais lenta. O valor padrão para este parâmetro é 4, mas pode-se adotar qualquer número no intervalo [0-31];
- ii. Número de palavras (n): define a quantidade de palavras que forma uma assinatura digital. Isto também contribui para a granularidade da comparação. Quanto maior o número, maior a exatidão, entretanto, a comparação será mais lenta. O valor definido como padrão é 3, que funciona bem em muitos casos;
- iii. Limiar ou *threshold* (t): indica um valor de corte, em que apenas os percentuais de semelhança superiores a esse valor são exibidos. O valor definido como padrão é 20%.

Na [Tabela 2.2](#) resume-se as características gerais das ferramentas mencionadas. O Sherlock destaca-se das demais por ser de propósito geral, assim o mesmo executável pode ser utilizado para várias linguagens sem modificar qualquer parâmetro da execução.

Tabela 2.2 – Resumo das características gerais

Ferramenta	Código Fonte	Acesso	Aplicação
Yap3	Aberto	Livre	C, Pascal, Lisp
MOSS	Fechado	Cadastro	C, C++, Java, C#,
		Web Service	Python, Visual Basic, Perl e 19 outras
JPlag	Fechado	Cadastro	C, C++, C#, Java,
		Web Service	Scheme e linguagem natural
SIM	Aberto	Livre	C, Java, Pascal, Modula-2,
			Lisp, Miranda e linguagem natural
Sherlock	Aberto	Livre	Propósito geral

2.3 Coeficientes de similaridade

No cálculo de similaridade entre códigos, algumas técnicas utilizam coeficientes de similaridades originalmente aplicados a conjuntos ([GREEN et al., 2012](#)). Para os coeficientes apresentados nesta seção, A e B representam dois conjuntos distintos. A interseção $|A \cap B|$ representa o conjunto de trechos de códigos em comum, dado os conjuntos de arquivos A e B .

Em seguida são elencados alguns coeficientes de similaridade encontrados com frequência em estudos relativos à detecção de plágio.

2.3.1 Coeficiente de Jaccard

O coeficiente de [Jaccard \(1901\)](#) é usado para calcular a similaridade entre dois conjuntos, baseando-se nas cardinalidades de interseção e união desses conjuntos, conforme apresentado na [Equação 2.8](#):

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2.8)$$

É comum encontrar na literatura o mesmo coeficiente expresso na forma da [Equação 2.9](#), em que $a = |A \cap B|$, $b = |A - B|$ e $c = |B - A|$, já que $|A \cup B| = |A \cap B| + |A - B| + |B - A|$.

$$J = \frac{a}{a + b + c} \quad (2.9)$$

Percebe-se, dessa maneira, a semelhança entre o coeficiente de Jaccard e a métrica utilizada no Sherlock de [Pike e Loki \(2013\)](#).

Levando em conta a sua formulação, o coeficiente de Jacquard se revela sensível ao aumento de elementos incomuns aos conjuntos comparados, aumentando o denominador e, assim, reduzindo a semelhança entre os conjuntos à medida que cresce o número de elementos distintos. Trazendo para o contexto da comparação entre códigos fonte, uma das técnicas de plágio que consiste na inclusão de código inútil no programa. Nestes casos, percebe-se considerável redução dos índices de similaridade entre os códigos comparados.

2.3.2 Coeficiente de Sorensen–Dice

[Dice \(1945\)](#) e [Sorensen \(1948\)](#), de forma independente publicaram o seguinte coeficiente:

$$Sorensen_Dice(A, B) = 2 \times \frac{|A \cap B|}{|A| + |B|} \quad (2.10)$$

Sabendo que $|A| + |B| = |A - B| + |B - A| + 2 \times |A \cap B|$, chega-se à [Equação 2.11](#).

$$D = \frac{2a}{2a + b + c} \quad (2.11)$$

Neste índice, embora haja uma sobrevalorização da quantidade de elementos em intersecção entre os conjuntos separados, ainda percebe-se a sensibilidade a existência de elementos incomuns, o que pode dificultar a análise de similaridade entre códigos-fonte à medida que são incluídas instruções inúteis no código plagiado.

2.3.3 Coeficiente Overlap

O coeficiente Overlap tem sido utilizado em vários trabalhos ([CHARIKAR, 2002](#)), ([MATSUO et al., 2007](#)), ([CESARE; XIANG, 2012](#)). Em nenhum deles é mencionada a autoria do coeficiente, que é expresso pela [Equação 2.12](#).

$$Overlap(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)} \quad (2.12)$$

Diferentemente dos coeficientes de similaridade de Jaccard e Sorensen-Dice, através deste é possível identificar a quantidade de elementos coincidentes em relação ao menor conjunto, o que reduz de maneira significativa a sensibilidade à inclusão de estruturas e instruções inúteis para dissimular diferenças entre programas.

Observa-se novamente que, considerando a análise de similaridade em um contexto mais amplo e não só buscando a identificação de plágio, a inclusão de código inútil pode ser devida ao reaproveitamento de programas anteriores que, por serem aplicados a problema diferente, podem carregar código que não seja apropriado à solução do novo problema proposto pelo professor. Como exemplo, observa-se o uso recorrente de inclusões de bibliotecas que não são utilizadas pelo novo programa, ou de variáveis que eram utilizadas no programa anterior e não são necessárias no novo programa.

2.3.4 Coeficiente de Tversky

O coeficiente de [Tversky \(1977\)](#) pode ser expresso pelas [Equações 2.13 e 2.14](#).

$$Tversky(A, B, \alpha, \beta) = \frac{|A \cap B|}{|A \cap B| + \alpha|A - B| + \beta|B - A|} \quad (2.13)$$

$$T = \frac{a}{a + \alpha \times b + \beta \times c} \quad (2.14)$$

em que $\alpha \geq 0$ e $\beta \geq 0$. Este coeficiente é uma generalização de vários outros. Por exemplo, para $\alpha = \beta = 1$, tem-se o coeficiente de Jaccard, para $\alpha = \beta = \frac{1}{2}$ tem-se o coeficiente de Sorensen-Dice, entre outros, segundo o próprio [Tversky \(1977\)](#).

Outras informações relacionadas podem ser encontradas nos trabalhos de [Wolda \(1981\)](#), [Charikar \(2002\)](#), [Zobel e Moffat \(1998\)](#) e [Cesare e Xiang \(2012\)](#).

2.4 Considerações finais

A principal contribuição deste trabalho é a concepção e implementação de uma ferramenta para análise de similaridade aplicável no domínio de práticas laboratoriais realizadas em disciplinas de programação. Depois de um estudo sistemático das principais abordagens discutidas na literatura sobre detecção de plágio em código-fonte, devido à disponibilidade do programa em forma de código aberto e de seu potencial para o domínio de aplicação em questão, decidiu-se investir em melhorias na proposta do algoritmo Sherlock de Pike ([PIKE](#); [LOKI, 2013](#)). As melhorias foram orientadas em duas perspectivas: a primeira, realizando modificações no coeficiente de similaridade usado pelo algoritmo, de maneira a melhorar a sua sensibilidade para comparação de assinaturas; a segunda, propondo técnicas de pré-processamento que, além de eliminar informação irrelevante, sejam também capazes de sobrevalorizar aspectos estruturais da linguagem de programação, reunindo ou separando sequências de caracteres cujo significado seja mais expressivo para a constituição das assinaturas ou, ainda, eliminando sequências menos relevantes para destacar outras que permitam melhor inferência sobre o grau de similaridade.

O resultado dessas modificações realizadas no Sherlock e seu uso sistemático apoiado em técnicas de pré-processamento deram origem ao Sherlock-N Overlap, apresentado no [Capítulo 3](#).

Sherlock N-Overlap: Normalização invasiva e coeficiente de sobreposição para identificação de similaridade

A motivação inicial deste trabalho foi avaliar a detecção de plágio presente no ambiente BOCALAB ([FRANÇA; SOARES, 2011](#)). Ao longo desse estudo, verificou-se a existência de lacunas no funcionamento da ferramenta Sherlock original e, a fim de minimizá-las, foram propostas e desenvolvidas algumas técnicas de pré-processamento de código, para melhor prepará-los à operação de comparação. Simultaneamente, o funcionamento do algoritmo Sherlock foi analisado quanto ao coeficiente de comparação, a fim de que pudesse ser avaliado com a utilização de outros comumente encontrados na literatura. A incorporação do coeficiente que apresentou melhores resultados e a utilização conjunta de técnicas de normalização constituem uma nova ferramenta denominada Sherlock N-Overlap.

3.1 Normalizando códigos-fonte para comparação com Sherlock

Propondo-se a identificar semelhanças léxicas, o Sherlock possui especial sensibilidade para espaços em branco, o que pode ser desejável para comparação de texto em linguagem natural. Entretanto, essa característica é limitante para a comparação de códigos-fonte. Por exemplo, para o compilador, é irrelevante escrever

`“for(i=0;i<n;i++){result+=v[i];}”` ou `“for (i = 0 ; i < n ; i++) { result + = v [i] ; }”`. Entretanto, para o Sherlock, o número de palavras usadas nesse segmento é absolutamente diferente e, como o algoritmo se apoia nesse conceito para a construção das assinaturas que serão comparadas, os índices de similaridade ficam completamente comprometidos. Assim, evidenciou-se a necessidade de normalizar os códigos quanto aos espaços em branco, o que gerou uma melhoria significativa nos resultados. A contiguidade ou não de algumas estruturas podem favorecer a comparação realizada pelo Sherlock, dando origem à concepção de um conjunto de normalizações.

Numa primeira abordagem, buscou-se um padrão uniforme de separação de palavras que são identificadas na fase de geração de assinaturas, dando origem às normalizações 1, 2 e 3, cujas regras são apresentadas na [Tabela 3.1](#).

Entretanto, observou-se que a ação de normalizar ultrapassa o aspecto de simples uniformização dos códigos-fonte, sendo possível alcançar melhores resultados com a captura de aspectos do código que são dificilmente fraudáveis. Nesse sentido, buscou-se identificar e destacar no processo de normalização as estruturas do código que preservam a essência da lógica empregada para resolver o problema ao qual ele se destina.

Pensando dessa forma, foi idealizada a normalização 4, apresentada por último na [Tabela 3.1](#), que preserva apenas o que não é texto literal. Com essa normalização, a maioria das ferramentas tradicionais sequer apresentam resultados, pois elas estão preparadas para receber um código completo e não sequências de caracteres com aparência aleatória. Isso ocorre especialmente para as ferramentas que realizam a “tokenização” do código. É interessante destacar que grande parte das mudanças realizadas pelos alunos que tentam efetuar plágio, ou que se baseiam em uma solução padronizada, não afeta as estruturas comparadas pela normalização 4. Por fim, tem-se com essa normalização uma representação da estrutura do código.

Cada uma das quatro técnicas de normalização descritas e exemplificadas na [Tabela 3.1](#) contém regras específicas de aproximação ou afastamento de caracteres.

As regras descritas nas quatro normalizações apresentadas na [Tabela 3.1](#) são aplicadas de maneira cumulativa, mas com diferenças pontuais quanto à remoção de espaços em branco entre elementos da linguagem, organizadas em quatro grupos. Portanto, as técnicas de normalização 1, 2, 3 e 4, respectivamente, vão da menos invasiva à mais invasiva,

Tabela 3.1 – Descrição e exemplificação das normalizações

Código Original	// Imprime resposta
<Sem modificação>	for (i=1;i<= n;i++){ if(%3== 1){ printf("Resp %d", v[i]/x+z); } }
Normalização 1	for(i=1; i<=n; i++) {
Remoção de linhas e espaços vazios; Remoção de todos os comentários; Remoção das referências aos arquivos externos (bibliotecas); Inclusão (ou remoção) de espaços em branco entre expressões, declaração de variáveis e outras estruturas. Regras específicas para aproximar e afastar caracteres para normalização 1.	if(i % 3 ==1) { printf (" Resp %d ", v[i] / x +z); } } }
Normalização 2	for(i=1; i<=n; i++) {
Aplicação da normalização 1; Remoção de todos os caracteres situados entre aspas. Regras específicas para aproximar e afastar caracteres para normalização 2.	if(i % 3 ==1) { printf(, v[i] / x +z); } } }
Normalização 3	for(= ; <= ; ++) {
Aplicação da normalização 2; Remoção de todos os valores literais e variáveis. Regras específicas para aproximar e afastar caracteres para normalização 3.	if(% ==) { printf(, [] / +); } } }
Normalização 4	(=; <=; ++) {
Aplicação da normalização 3; Remoção de todas as palavras reservadas. Regras específicas para aproximar e afastar caracteres para normalização 4.	(%==) { (, [] / +); } } }

considerando-se o grau de modificação efetuado no código-fonte antes de submetê-lo à comparação.

O nível de invasão no código original pode ser examinado na [Tabela 3.1](#), em que, na coluna da esquerda, são apresentadas as regras aplicadas para cada tipo de normalização e, na coluna da direita, os resultados da aplicação da normalização em um exemplo de trecho de código-fonte. Observa-se, portanto, que o nível de invasão oferecido pela normalização 4 impede a compreensão do mesmo por um leitor humano, embora guarde elementos suficientemente importantes para a identificação de similaridade entre a estrutura de códigos semelhantes. Por outro lado, a normalização 1 é aquela que oferece menor dificuldade de leitura do código após a sua aplicação.

3.1.1 Análise da organização das estruturas quanto aos espaços em branco

Observando a técnica de normalização 1 aplicada ao código exemplo, verificou-se empiricamente que palavras reservadas indicativas de funções parametrizadas e estruturas condicionais, como: `“printf (”,` e `“if (”,` são melhor comparadas quando se eliminam os espaços antes do parêntese, resultando em `“printf(”` e `“if(”`.

Entretanto, as consequências ocorridas na comparação feita pelo Sherlock em relação à separação pelo espaço em branco de elementos de estruturas logicamente associadas requer que, na normalização 4, os caracteres especiais sejam unidos, como em `“(,);”,` ao invés de `“(,) ;”,` para se obter o melhor resultado.

Em alguns experimentos, observou-se empiricamente que, para expressões como `“i%3==1”,` obtêm-se melhores resultados quando o valor numérico está junto ao operador do lado direito, e os elementos do lado esquerdo estão separados, com a seguinte configuração: `“i % 3 ==1”`. Contudo, para declaração de variáveis, por exemplo, `“int i=1;”,` a organização que resulta em melhores resultados é `“int i = 1;”`.

Cada normalização representa em si uma estratégia particular, que herda da anterior algumas propriedades, mas que podem ser diferenciadas em relação às regras que levam a aproximar ou afastar determinados caracteres. Por exemplo, para a normalização 3, os resultados são melhores quando os operadores `“% ==”` ficam separados, mas na normalização 4 é necessário deixá-los, como em `“%==”`.

3.1.2 Discussões sobre o alto grau de invasão proporcionado pela Normalização 4

A normalização 4, que preserva apenas o que não é texto literal, apesar de mais invasiva, de maneira geral, é a técnica com a qual o Sherlock apresenta os melhores resultados, conforme resultados do [Capítulo 5](#). Técnicas invasivas são utilizadas também em outros domínios. Apesar de não ter servido de inspiração para o presente trabalho, tratando-se apenas de uma coincidência e publicada um ano antes dos estudos sobre a normalização 4 ([MACIEL et al., 2012](#)), [Stamatatos \(2011\)](#) utilizou um método similar ao da normalização 4 com Sherlock, no domínio da linguagem natural. Na "normalização" proposta por Stamatatos são retiradas dos textos a serem comparados todas as palavras que não compõem a lista das 50 palavras mais comuns do Inglês, segundo o *British National Corpus*¹. Assim, apenas as palavras da [Figura 3.1](#) são preservadas, para cara texto. Na etapa seguinte, uma quantidade *n-gram* de palavras consecutivas são selecionadas e comparadas.

Figura 3.1 – As 50 palavras mais frequentes do Inglês, conforme BNC corpus.

1. the	11. with	21. are	31. or	41. her
2. of	12. he	22. not	32. an	42. n't
3. and	13. be	23. his	33. were	43. there
4. a	14. on	24. this	34. we	44. can
5. in	15. I	25. from	35. their	45. all
6. to	16. that	26. but	36. been	46. as
7. is	17. by	27. had	37. has	47. if
8. was	18. at	28. which	38. have	48. who
9. it	19. you	29. she	39. will	49. what
10. for	20. 's	30. they	40. would	50. said

Fonte: [Stamatatos \(2011\)](#).

A [Figura 3.2](#) exemplifica esse processo, em que dois trechos em inglês parafraseados são representados apenas pelas palavras da [Figura 3.1](#) e então comparados. Essa abordagem é bem interessante por dois motivos principais: pela semelhança da abordagem utilizada

¹ <http://corpus.byu.edu/bnc/>

Figura 3.2 – Exemplo do funcionamento da proposta de Stamatatos (2011) para comparação de 8-grams.

Suspicious passage:	Original passage:
<i><u>This</u> came into existence likely <u>from the</u> deviance <u>in the</u> time-period <u>of the</u> particular billet. <u>As the</u> premier <u>is to be</u> nominated <u>for not</u> more than <u>a period of</u> four years, <u>it can</u> infrequently happen <u>that an</u> ample wage, fixed <u>at the</u> embarkation <u>of that</u> period, <u>will not</u> endure <u>to be</u> such <u>to</u> its end.</i>	<i><u>This</u> probably arose <u>from the</u> difference <u>in the</u> duration <u>of the</u> respective offices. <u>As the</u> President <u>is to be</u> elected <u>for</u> no more than four years, <u>it can</u> rarely happen <u>that an</u> adequate salary, fixed <u>at the</u> commencement <u>of that</u> period, <u>will not</u> continue <u>to be</u> such <u>to</u> its end.</i>
SWNG representation:	SWNG representation:
[this,from,the,in,the,of,the,as]	[this,from,the,in,the,of,the,as]
[from,the,in,the,of,the,as,the]	[from,the,in,the,of,the,as,the]
[the,in,the,of,the,as,the,is]	[the,in,the,of,the,as,the,is]
[in,the,of,the,as,the,is,to]	[in,the,of,the,as,the,is,to]
[the,of,the,as,the,is,to,be]	[the,of,the,as,the,is,to,be]
[of,the,as,the,is,to,be,for]	[of,the,as,the,is,to,be,for]
[the,as,the,is,to,be,for,not]	[the,as,the,is,to,be,for,it]
[as,the,is,to,be,for,not,a]	[as,the,is,to,be,for,it,can]
[the,is,to,be,for,not,a,of]	[the,is,to,be,for,it,can,that]
[is,to,be,for,not,a,of,it]	[is,to,be,for,it,can,that,an]
[to,be,for,not,a,of,it,can]	[to,be,for,it,can,that,an,at]
[be,for,not,a,of,it,can,that]	[be,for,it,can,that,an,at,the]
[for,not,a,of,it,can,that,an]	[for,it,can,that,an,at,the,of]
[not,a,of,it,can,that,an,at]	[it,can,that,an,at,the,of,that]
[a,of,it,can,that,an,at,the]	[can,that,an,at,the,of,that,will]
[of,it,can,that,an,at,the,of]	[that,an,at,the,of,that,will,not]
[it,can,that,an,at,the,of,that]	[an,at,the,of,that,will,not,to]
[can,that,an,at,the,of,that,will]	[at,the,of,that,will,not,to,be]
[that,an,at,the,of,that,will,not]	[the,of,that,will,not,to,be,to]
[an,at,the,of,that,will,not,to]	
[at,the,of,that,will,not,to,be]	
[the,of,that,will,not,to,be,to]	

Fonte: Stamatatos (2011).

para código fonte com a Normalização 4, e pela facilidade de aplicá-la para o Português por meio de normalização específica e o Sherlock.

Observa-se que, embora a normalização 4 seja bastante eficiente quando usada em conjunto com o Sherlock, a maioria das ferramentas conhecidas para detecção de plágio entre códigos-fonte não apresentam bons resultados, pois elas estão preparadas para receber um código completo, possuindo analisadores sintáticos específicos. Isso ocorre

especialmente para as ferramentas que realizam a tokenização do código. Em geral, devido à grande maioria dos algoritmos/ferramentas utilizarem tokenização, não faz sentido aplicar a eles as abordagens aqui propostas de normalização, particularizando-as, portanto, para aplicação em algoritmos como o Sherlock.

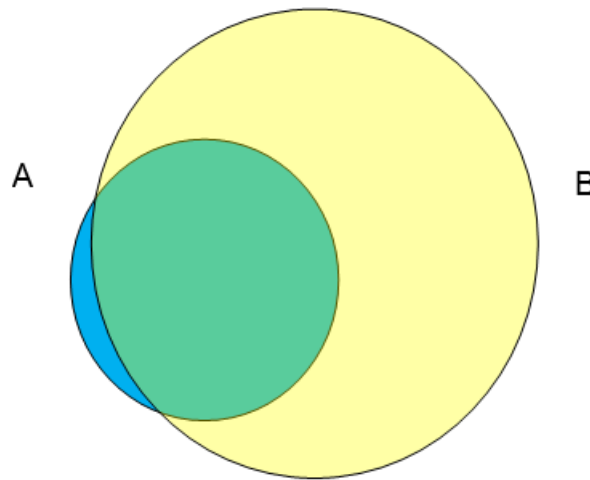
Apesar do grau elevado de invasão da normalização 4, foi observado que grande parte das mudanças realizadas pelos alunos que tentam efetuar plágio ou que se apoiam em soluções de referência não afeta as estruturas mantidas após o pré-processamento. Esse fato é corroborado pelos resultados apresentados no [Capítulo 5](#).

3.2 Sherlock N-Overlap: adaptando o algoritmo Sherlock original com o coeficiente de sobreposição

Conforme visto na [subseção 2.2.5](#), o Sherlock utiliza métrica de similaridade análoga ao coeficiente de similaridade de Jaccard ([JACCARD, 1901](#)). Dessa forma, outras métricas foram estudadas a fim de verificar possíveis melhorias nos indicadores de similaridade. Os coeficientes de Sorensen-Dice ([DICE, 1945](#)), ([SORENSEN, 1948](#)) e Overlap foram avaliados, tendo este último apresentado os melhores resultados.

Analisando as equações [2.8](#) do coeficiente de similaridade de Jaccard e a [2.12](#) do coeficiente de similaridade Overlap, pode-se inferir que este último torna o código-fonte mais robusto ao enxerto de código inútil. A [Figura 3.3](#) ilustra uma situação, em que o conjunto A representa o conjunto das assinaturas geradas pelo Sherlock para o código-fonte original e o conjunto B aquelas geradas para o código-fonte plagiado.

Percebe-se na [Figura 3.3](#) que, com o incremento de assinaturas no conjunto B devido ao enxerto de código inútil, o cálculo do coeficiente de similaridade de Jaccard, que considera em seu denominador o número de assinaturas dos dois conjuntos, implicará sempre em percentuais inferiores ao índice de similaridade Overlap, cujo denominador corresponde ao número de assinatura do menor dos dois conjuntos.

Figura 3.3 – Interpretação gráfica do coeficiente Jaccard.

3.3 Discussões sobre o Sherlock N-Overlap

Um aspecto positivo encontrado no Sherlock é a simplicidade e a versatilidade do algoritmo. De código aberto, permite a verificação de grande variedade de situações para estudo, o que não é possível com quase todas as ferramentas estudadas. Outra ferramenta com bastante potencial para estudo é o SIM ([GRUNE; VAKGROEP, 1989](#)), por ser *stand-alone* e ter o código aberto. Contudo, em comparação com o Sherlock, O SIM apresenta funcionamento mais complexo. Para exemplificar, enquanto o Sherlock é constituído de um único arquivo .c e com menos de 400 linhas, o SIM é dividido em 52 arquivos entre .c e .h.

Além disso, a possibilidade de empregar o Sherlock com linguagem natural, em conjunto com a aplicação das técnicas estudadas, amplia a sua relevância para uso em ambientes virtuais para comparar respostas subjetivas de alunos.

O Sherlock N-Overlap incorpora tanto os aspectos positivos do Sherlock original quanto algumas de suas limitações. Dentre elas, devido à característica de funcionar por linha de comando, ele não oferece uma interface apropriada para avaliação dos resultados pelo professor. Outra limitação é a impossibilidade de destacar, nos códigos comparados, os trechos semelhantes encontrados, diferentemente do que ocorre com JPlag e MOSS. Para adicionar tal funcionalidade, é necessária uma modificação no Sherlock para armazenar as posições das substrings na etapa de geração das assinaturas *hashs*.

Uma outra limitação do Sherlock é potencializada pela aplicação da normalização 4 em códigos muito pequenos. Devido à grande redução do tamanho do código por essa normalização, é necessária atenção ao empregar o valor do parâmetro *zerobit*, que também atua como um elemento redutor das assinaturas que serão comparadas. Esses dois fatores de redução do código podem gerar assinaturas em quantidade insuficiente para uma boa comparação. Existem várias formas de mitigar esse problema. A primeira é estudar modificações da normalização 4 para que a diminuição no código resultante seja menor. A segunda é consequência dos resultados apresentados no [Capítulo 5](#), em que descobriu-se que para códigos com até 20 linhas, a utilização do *zerobit* com valor 2 é suficiente para obter bons resultados. Uma terceira forma é desenvolver uma análise automática do código que relacione seu tamanho com a quantidade de assinaturas geradas e, com isso, configure automaticamente os melhores parâmetros para execução do Sherlock N-Overlap.

Apesar das conhecidas limitações, os resultados obtidos com o Sherlock N-Overlap não podem ser negligenciados e, por isso, foram comparados com ferramentas de referência bastante conhecidas e empregadas para a detecção de plágio entre códigos-fonte. O resultado da avaliação da ferramenta são apresentados e discutidos no [Capítulo 4](#).

3.4 Considerações Finais

Para avaliar as modificações propostas no Sherlock original, foi necessário compará-las com o estado da arte em termos de detecção de plágio em códigos-fonte. A comparação de todas essas técnicas não é trivial por vários motivos. Primeiramente, não existe um banco de dados ou repositório de códigos-fonte cujos *status* plágio ou não-plágio sejam previamente conhecidos, na qualidade do que ocorre para estudo de plágio em linguagem natural ([POTTHAST et al., 2010](#)) ([CLOUGH; STEVENSON, 2011](#)) ([CEGLAREK, 2013](#)). Fora do escopo deste trabalho, foi encontrado um dataset² para linguagem Java, utilizado por [Poon et al. \(2012\)](#). Além disso, não foi encontrada uma ferramenta que permita a comparação simultânea, seja analítica ou quantitativa, dos índices de similaridade para pares de código gerados por múltiplas ferramentas. A variedade de ferramentas e suas respectivas configurações exige que essa avaliação seja realizada, de preferência, com a execução repetida dessas combinações sobre um mesmo conjunto

² <http://wing.comp.nus.edu.sg/downloads/SSID/>

de dados. Para manter o controle dessas informações, também é desejável tabular e documentar os resultados de cada execução de forma apropriada. Os desafios intrínsecos à análise de resultados se tornaram, assim, requisitos e objetivos adicionais deste trabalho, suscitando a concepção de um arcabouço metodológico de análise que permite gerar e documentar a comparação de ferramentas de detecção de plágio. Este arcabouço, que é apresentado detalhadamente no [Capítulo 4](#), tornou imprescindível a construção de uma aplicação para compilar os resultados de múltiplas ferramentas de detecção de plágio em uma mesma interface. Assim, essa aplicação que dá apoio à análise de resultados representa uma outra contribuição deste trabalho. Além de integrar vários algoritmos para detecção de plágio e de pré-processamento de código (normalização), a aplicação pode ser facilmente estendida para a incorporação de outras ferramentas/algoritmos.

A integração de alguns importantes algoritmos de detecção de plágio nesta ferramenta de análise exigiu a concepção de soluções particulares. Por exemplo, apesar da disponibilidade do *plugin* PCPP (2012) para Moodle (DOUGIAMAS; TAYLOR, 2003), a integração do JPlag e do MOSS, que são algoritmos de código fechado e disponibilizados apenas via *webservices*, precisou ser adaptada de maneira a possibilitar a comparação simultânea dos resultados. Outra limitação na abordagem presente no *plugin* é a de apresentar uma interface específica para cada algoritmo (JPlag e MOSS), o que dificulta a integração para apresentação de resultados de maneira conjunta aos de outras ferramentas. Além disso, a utilização desse *plugin* é realizada exclusivamente via Moodle.

No [Capítulo 4](#), é apresentado o arcabouço desenvolvido para análise comparativa do Sherlock N-Overlap com outras ferramentas de detecção de plágio.

Capítulo 4

Um arcabouço para análise comparativa do Sherlock N-Overlap com outras ferramentas de detecção de plágio com método próprio de comparação

Este capítulo apresenta o arcabouço metodológico utilizado para avaliar a eficácia do algoritmo Sherlock N-Overlap. Para a extração de resultados, foi desenvolvida uma ferramenta que integra múltiplos algoritmos e as técnicas de normalização apresentadas no [Capítulo 3](#). Essa ferramenta, denominada neste trabalho Sistema de Análise de Similaridade, foi direcionada inicialmente ao professor para análise da semelhança entre as soluções apresentadas pelos alunos para problemas propostos em laboratório e resolvidos via BOCALAB. Em sua primeira versão, apenas o Sherlock com as quatro técnicas de normalização estavam disponíveis. Com a integração das ferramentas SIM, MOSS e JPlag, além do Sherlock N-Overlap, ampliou-se o espectro de sua utilização para estudo comparativo dos resultados dos diversos algoritmos.

Contudo, analisar os índices de semelhança para os diversos algoritmos, alguns combinados às quatro técnicas de normalização, não é um trabalho trivial. Verificou-se, portanto, a necessidade de utilização de uma abordagem exclusivamente voltada para

análise comparativa dos índices de semelhança gerados pelos diversos algoritmos.

Assim, neste capítulo são apresentados, nessa ordem, o sistema de análise de similaridade e o arcabouço metodológico para a análise de resultados. Ambos dão suporte à discussão de resultados do [Capítulo 5](#).

4.1 Sistema de Análise de Similaridade

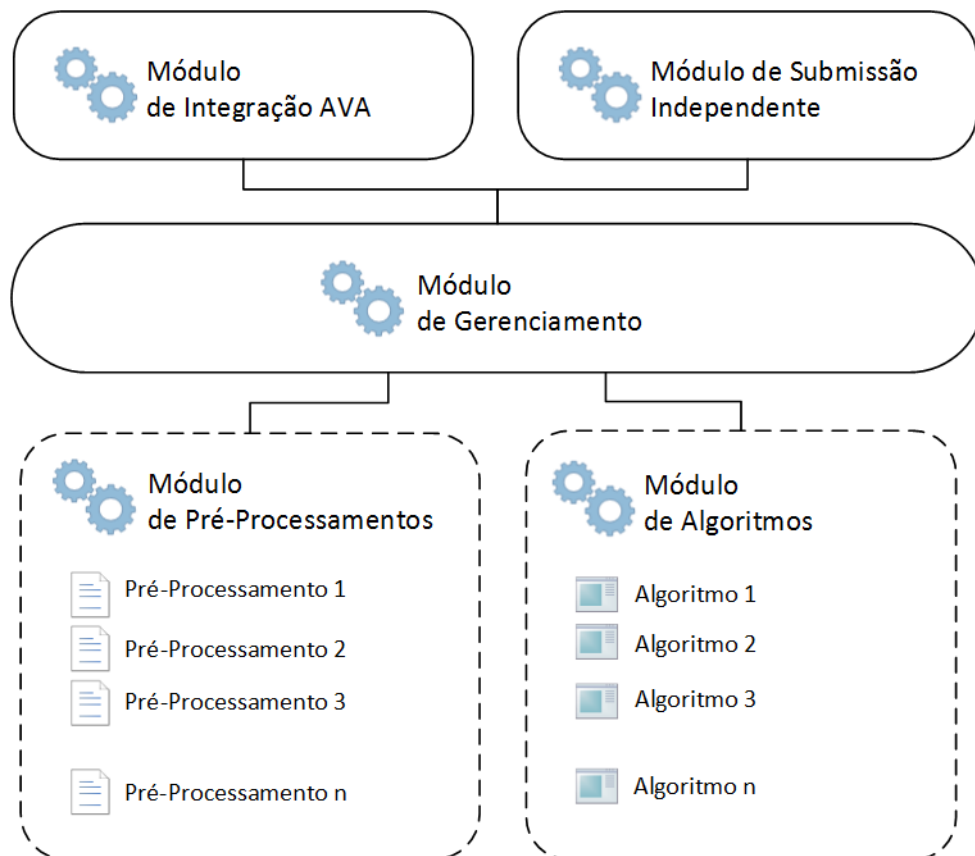
Cada sistema ou ferramenta de detecção disponível possui seu modo próprio de submissão de arquivos e apresentação de resultados. Para comparar todos os resultados de forma eficaz, desenvolveu-se uma ferramenta, a qual denominamos “Sistema de Análise de Similaridade”, que integra os principais sistemas de detecção de plágio disponíveis: JPlag ([PRECHELT; MALPOHL; PHILIPPSSEN, 2002](#)), MOSS ([SCHLEIMER; WILKERSON; AIKEN, 2003](#)) e SIM ([GRUNE; VAKGROEP, 1989](#)). Nem todas as tentativas de integração obtiveram sucesso. Por exemplo, a integração do Yap3 não foi possível devido à incapacidade de fazê-lo funcionar apropriadamente em ambiente Linux. Contudo, isso não foi considerado uma limitação importante, pois o JPlag já é apresentado na literatura como uma versão melhorada do Yap3. O código do SIM ([GITCHELL; TRAN, 1999](#)), diferentemente do SIM ([GRUNE; VAKGROEP, 1989](#)), não foi encontrado e, portanto, não foi integrado.

A ferramenta de Análise de Similaridade gerencia a localização dos arquivos a serem comparados, executa as técnicas de normalização desenvolvidas e os algoritmos de comparação à escolha do usuário, tabula todos os resultados em uma interface única e implementa o método de avaliação proposta, conforme [subseção 4.2.2](#). O sistema de Análise de Similaridade é extensível, permitindo incorporar outros algoritmos e técnicas de normalização e, até o momento em que essa dissertação foi redigida, constitui a única ferramenta do gênero conhecida.

4.1.1 Arquitetura

O sistema de análise proposto é subdividido em 5 módulos: Módulo de Algoritmos, Módulo de Pré-Processamento, Módulo de Integração com Ambientes Virtuais de Aprendizagem (AVAs), Módulo de Submissão Independente e Módulo de Gerenciamento. A [Figura 4.1](#) representa a arquitetura composta por esses módulos inter-relacionados.

Figura 4.1 – Arquitetura



O Módulo de Algoritmos contém a integração das ferramentas de comparação. Na versão atual, está disponível integração com os ferramentas JPlag, MOSS, SIM, Sherlock e Sherlock N-Overlap. A concepção e integração de novos algoritmos é facilitada pela ferramenta. Por exemplo, é possível desenvolver uma abordagem a partir da combinação do resultado de outros algoritmos, calculando-se a média do resultado de 3 algoritmos já integrados. Uma nova abordagem pode ser desenvolvida diretamente em linguagem PHP no Módulo de Algoritmos ou ser integrada na forma de um programa executável pré-existente, à semelhança do SIM e do Sherlock, ou ainda via *webservices*, como foi o caso do JPlag e do MOSS.

O Módulo de Pré-Processamento reúne os algoritmos relativos às técnicas de normalização. Na versão atual, estão presentes as 4 normalizações.

O Módulo de Gerenciamento é o núcleo do sistema de análise. Por intermédio dele registram-se as configurações de cada algoritmo, recebem-se os arquivos para comparação e coordena-se a execução dos algoritmos.

No que se refere à interface com usuário, dois módulos estão disponíveis: o Módulo de Integração e o Módulo de Submissão Independente. Esses módulos alimentam a localização dos arquivos a serem comparados, apresentam ao usuário a interface *web* com os algoritmos e normalizações disponíveis e formatam os resultados. Dessa forma, o Módulo de Integração AVA integra-se a um ambiente virtual de aprendizagem, que pode ser o Moodle ou algum outro. Na versão atual, tem-se uma implementação de cada, um módulo de integração AVA que captura as submissões realizadas via BOCALAB e outra que possibilita a submissão de arquivo de tarefa do próprio Moodle. O Módulo de Submissão Independente funciona para submissão direta de quaisquer arquivos para comparação. O funcionamento é análogo ao das ferramentas via *webservice*, com a diferença de que o código é disponibilizado e a submissão é realizada por meio de uma página *web*. Todos esses módulos de interface com o usuário podem coexistir para um mesmo Módulo de Gerenciamento.

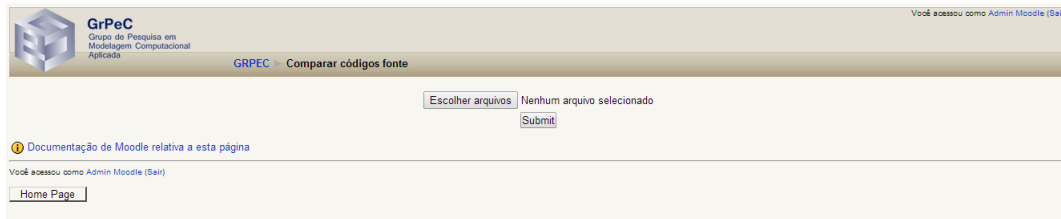
4.1.2 Funcionamento e interface

A interface para detecção de plágio do BOCALAB (FRANÇA; SOARES, 2011) foi a base para a implementação do novo sistema. O sistema de Análise de Similaridade possui cinco funcionalidades: Selecionar arquivos, Adicionar/Selecionar novo algoritmo, Selecionar Normalização, Executar Comparação e Analisar pares individualmente.

Selecionar arquivos

A [Figura 4.2](#) mostra a página para submissão de arquivos para análise de similaridade, por meio da qual os arquivos são selecionados de um diretório e submetidos para comparação.

Figura 4.2 – Submissão externa - Página de submissão



De modo complementar, quando o sistema de Análise está integrado a um ambiente virtual de aprendizagem (AVA), como o Moodle, podem ser diretamente submetidos para comparação os arquivos submetidos ao AVA como uma tarefa proposta. Nessa situação, a tela de seleção de arquivos é substituída por outra, com uma lista das atividades desenvolvidas pelos alunos.

Definição do algoritmo/ferramenta de comparação e Seleção da normalização

A cada adição de um novo algoritmo/ferramenta, também é possível selecionar a técnica de Normalização. Essas duas configurações são feitas por meio da interface mostrada na [Figura 4.3](#). Em seguida, configuram-se os parâmetros e as técnicas de normalização a serem empregadas sobre os códigos submetidos.

Figura 4.3 – Submissão externa - Página de comparação



Para cada algoritmo a ser aplicado, algumas configurações são possíveis. Também é possível configurar aspectos específicos para a construção de uma normalização

personalizada. As opções para algoritmos e normalizações são mostradas na [Figura 4.4](#), com as possibilidades de configuração do Sherlock, Sherlock Overlap e uso da normalização personalizada. Para as versões do Sherlock, é possível alterar o número de palavras, *zerobit* e tolerância aceita por arquivo (*Threshold*). A normalização personalizada contém grande variedade de opções. Na versão inicial, integrada ao BOCALAB, apenas as configurações do Sherlock podiam ser alteradas por meio da interface, sendo as demais ferramentas executadas com as respectivas configurações padrão.

Figura 4.4 – Configurações do Sherlock e Normalização Personalizada

The figure consists of two screenshots of the GrPeC web interface, showing the configuration options for the Sherlock algorithm and the Personalized Normalization tool.

Top Screenshot: The interface shows the 'Parâmetros do Sherlock' section with the following options:

- Número de palavras: 3
- Zerobits - Granularidade (0-31): 2
- Tolerância aceita por arquivo: 20%

 The 'Algoritmo' section shows 'Sherlock' selected, with 'Normalização 1' chosen from a dropdown. A 'Destacar apenas percentagens superior a:' dropdown is set to 5%. A 'COMPARAR' button is visible at the bottom.

Bottom Screenshot: This screenshot shows the 'Personalizar Normalização' section, which is expanded to show various options:

- ☒ Organizar Código
- ☐ Caracteres em Caixa Baixa
- ☐ Remover Literais (Textos entre aspas)
- ☐ Remover Variáveis
- ☐ Remover Valores Numéricos
- ☐ Remover Chaves
- ☐ Deixar Apenas Palavras Reservadas
- ☐ Permitir Caracteres Específicos
- ☐ Remover Palavras Reservadas
- ☐ Aproximar Caracteres Específicos 01
- ☐ Aproximar Caracteres Específicos 02
- ☐ Afastar Caracteres Específicos 01
- ☐ Afastar Caracteres Específicos 02

 The 'Algoritmo' section shows 'Sherlock' selected, with 'Normalização Personalizada' chosen from a dropdown. A 'Destacar apenas percentagens superior a:' dropdown is set to 5%. A 'COMPARAR' button is visible at the bottom.

Execução da comparação

Após a configuração dos algoritmos e das normalizações, pode-se executar a comparação dos códigos-fonte submetidos. Na [Figura 4.5](#) é apresentado o resultado da comparação com apenas um algoritmo selecionado. Nessa configuração, todos os códigos são combinados 2 a 2, apresentando a similaridade para cada par de código em uma célula. Os pares com similaridade superior a 95% são destacados com a cor vermelha, os pares com similaridade igual a 0% são marcados com a cor verde e todos os demais com a cor laranja.

É possível configurar a ferramenta de modo que todos os pares com similaridade inferior a um determinado limiar sejam desconsiderados. Dessa maneira, todos os

Figura 4.5 – Interface com execução de um algoritmo

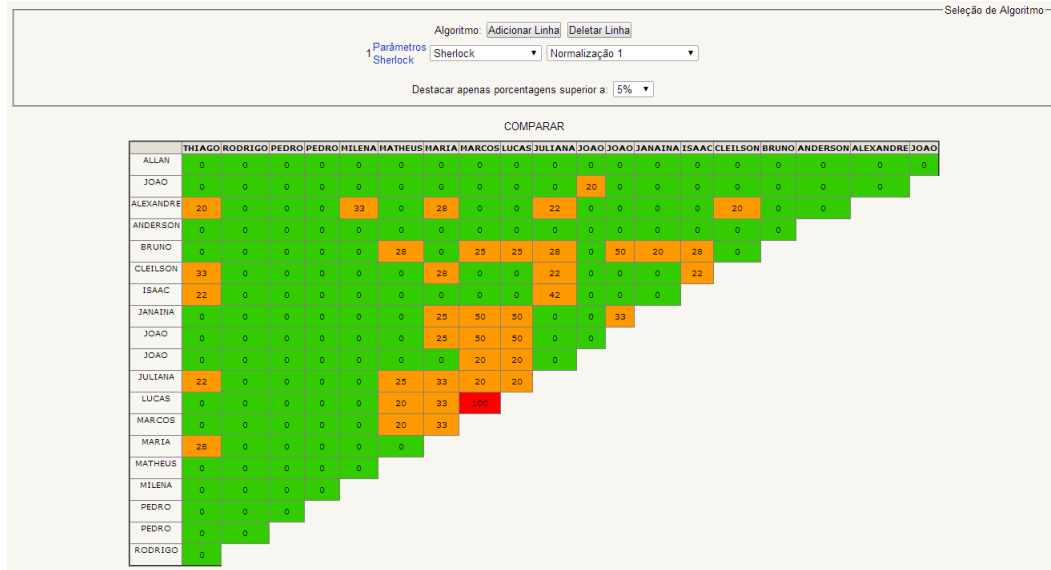
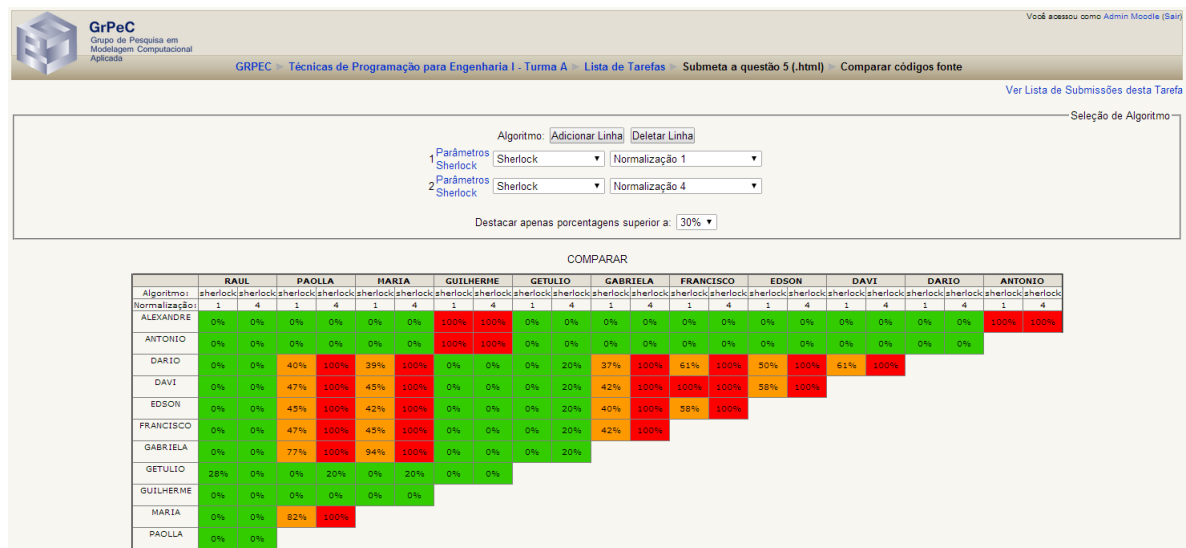


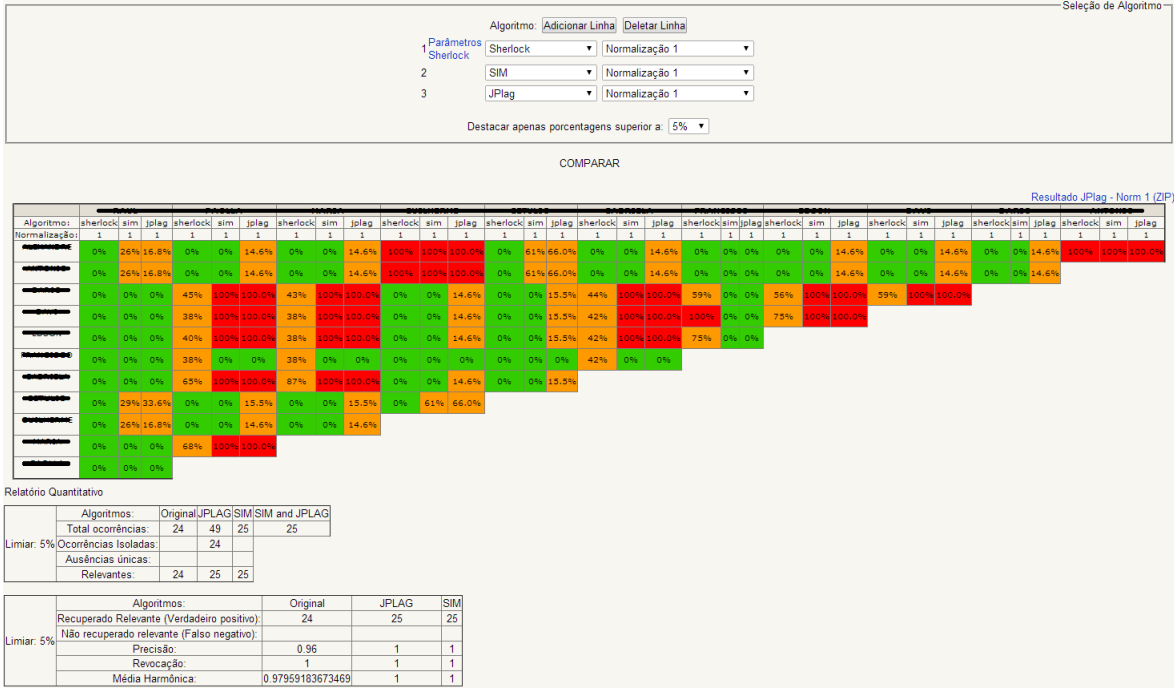
Figura 4.6 – Interface com execução de dois algoritmos



valores inferiores a esse limiar são marcados com a cor verde. Essa funcionalidade facilita a visualização, especialmente quando ocorrem muitos valores com baixo índice de similaridade.

A Figura 4.6 apresenta o resultado de uma comparação de um mesmo algoritmo com duas diferentes técnicas de normalização. Com essa configuração, a tabela de similaridade ganha duas novas informações para cada par: a identificação do respectivo algoritmo e a normalização utilizada. Essa diferença também pode ser observada na Figura 4.7, desta vez com a execução de 3 algoritmos diferentes e a adição de um relatório quantitativo, que permite contabilizar uma comparação mais efetiva para quando se executa algoritmos

Figura 4.7 – Interface com execução de três algoritmos



diferentes. As informações presentes nesse relatório estão detalhadas na [subseção 4.2.2](#).

Com a interface proposta, é simples verificar os resultados da detecção de similaridade de diferentes algoritmos sobre um mesmo par.

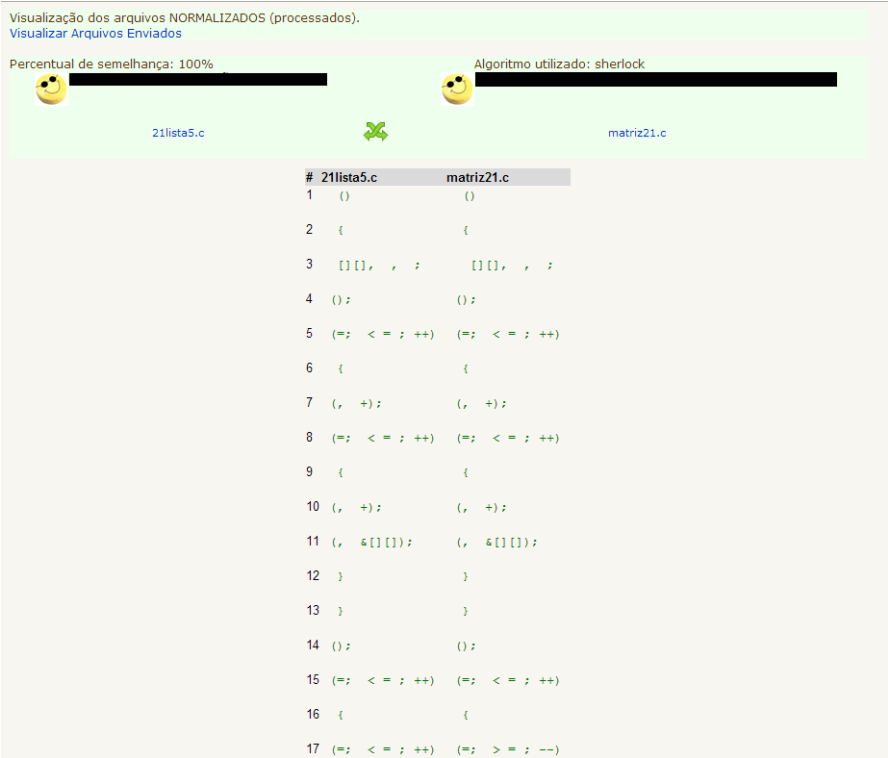
Analisar pares individualmente

Para uma análise mais apurada sobre a similaridade dos pares de código, é possível ter acesso aos códigos originais e/ou processados, lado a lado, com um simples clique sobre o percentual de similaridade apresentado em uma célula.

Na [Figura 4.8](#) representa-se a comparação lado a lado de dois códigos originais, portanto sem considerar a etapa de normalização. Nessa tela é possível observar o percentual de similaridade calculado para os códigos originais, bem como o horário de submissão dos arquivos, que é disponibilizado na versão do sistema de Análise de Similaridade integrado a um AVA. Na região superior esquerda dessa página existe um [link](#) para visualizar os respectivos códigos normalizados, configurados de acordo com as etapas anteriores à comparação.

Na [Figura 4.9](#) são mostrados exemplos de códigos pré-processados pela normalização 4.

Figura 4.9 – Análise de pares: arquivos normalizado



4.2 Arcabouço metodológico para avaliação de resultados

Ainda que o sistema de Análise de Similaridade simplifique e sumarie o acesso ao resultado das comparações para os diversos algoritmos, a análise dos dados para múltiplos pares constitui um trabalho intenso, necessitando de abordagens que consigam sistematizar a avaliação para o operador humano. As próximas subseções discutem abordagens tradicionalmente utilizada para medir a eficiência de ferramentas de detecção de plágio e uma proposição concebida no contexto deste trabalho.

4.2.1 Método tradicional para comparação das ferramentas

Os valores de similaridade retornados pelas diferentes ferramentas não são facilmente comparáveis, devido a utilizarem medidas de similaridades que consideram, muitas vezes, propriedades diferentes ou realizam diferentemente o trabalho de preparação do código antes da comparação. A maioria dos autores considera o cálculo do desempenho de ferramentas de detecção de plágio como um problema equivalente ao de avaliar algoritmos de classificação de dados. Em geral, utilizam-se de medidas de precisão e revocação, assim como a média harmônica entre elas, conforme encontrado nos trabalhos de [Prechelt](#),

Malpohl e Philippsen (2002), Burrows, Tahaghoghi e Zobel (2007), Duric e Gasevic (2012), Cosma e Joy (2012), e Ohmann (2013).

Antes de definir essas medidas, é importante compreender os quatro tipos possíveis de resultados de um sistema de classificação, conforme apresentado na [Tabela 4.1](#). Contabilizados, esses resultados determinam a eficácia da resposta de um algoritmo com base na precisão e na revocação.

Considerando os casos recuperados como similares, estes podem ser compostos pelos que são de fato similares (A) e por aqueles que não o são, mas foram assim considerados pelo algoritmo de classificação, os que chamaremos de **falsos positivos** (B). Por outro lado, nos casos em que não foram detectadas similaridades, podem existir os que de fato não são similares (C) e os que são similares, mas não foram considerados assim pelo algoritmo, os que chamaremos de **falsos negativos** (D). O sistema ideal é aquele que identifica a similaridade, considerando certo limiar, para todos os verdadeiros positivos e apenas para eles (A).

Tabela 4.1 – Representação dos parâmetros de cálculo para precisão e revocação

Pares sem indicação de similaridade para um dado limiar	Pares com indicação de similaridade para um dado limiar
Pares relevantes não recuperados (D) (Falso negativo)	Pares relevantes recuperados (A) (Verdadeiro positivo)
Pares não relevantes não recuperados (C) (Verdadeiro negativo)	Pares não relevantes recuperados (B) (Falso positivo)

A **precisão**, denominada por P , onde $P \in [0, 1]$, é a proporção dos documentos propriamente relevantes recuperados dentre o total dos que foram recuperados, incluindo os falsos positivos, sendo calculada pela [Equação 4.1](#):

$$P = \frac{\text{Total de Pares Relevantes Recuperados (A)}}{\text{Total de Pares Recuperados (A + B)}} \quad (4.1)$$

Dessa maneira, o valor da precisão é 1 (um) quando não há falso positivo.

A **revocação**, denominada por R , onde $R \in [0, 1]$, é a proporção de documentos que foram identificados como similares em um total que considera também os falsos negativos, sendo calculada pela [Equação 4.2](#):

$$R = \frac{\text{Total de Pares Relevantes Recuperados (A)}}{\text{Total de Pares Relevantes (A + D)}} \quad (4.2)$$

Dessa forma, o valor da revocação é 1 (um) quando todos os pares similares, para determinado limiar, são detectados completamente ou, em outras palavras, quando não há falso negativo.

Assim, se uma ferramenta recupera **todos** os pares similares ($R = 1$), não significa que a ferramenta é ideal, pois podem existir falsos positivos. De outro modo, se uma ferramenta recupera **apenas** os pares similares ($P = 1$), não significa que a ferramenta é ideal, pois podem existir falsos negativos. O desempenho global de uma ferramenta é obtido pela combinação dessas duas medidas. Uma boa forma de se obter delas um valor único é por meio do cálculo da média harmônica ([DURIC; GASEVIC, 2012](#)), ([CEDEÑO, 2012](#)), ([OHMANN, 2013](#)). A média harmônica da precisão e da revocação é calculada pela [Equação 4.3](#):

$$F = 2 \times \frac{P \times R}{P + R} \quad (4.3)$$

em que $F \in [0, 1]$. Quanto mais próximo de 1 (um) for o valor de F , melhor o desempenho da ferramenta de detecção.

Tendo por base as contribuições em termos do método de avaliação do desempenho dos algoritmos e das normalizações desenvolvidas, foram realizados testes em duas etapas: a primeira, denominada **manual**, usando códigos cuja a classificação de plágio era conhecida *a priori*; a segunda, denominada **automática**, foi realizada sobre códigos em que não se conhece antecipadamente a classificação de plágio. No segundo caso, foi necessário adaptar a maneira de calcular o desempenho dos algoritmos, conforme detalhado na [subseção 4.2.2](#).

4.2.2 Nova abordagem para comparação das ferramentas: o método de conformidade

A tarefa de avaliar a similaridade dos códigos gerados por alunos nas submissões de atividades/provas não é trivial. A grande quantidade de códigos, assim como o fato de uma parte deles apresentar tamanho muito reduzido, além da subjetividade envolvida para avaliar similaridade, são fatores complicadores dessa tarefa. Para aplicar o método de cálculo definido na [subseção 4.2.1](#), é necessário realizar previamente a classificação dos pares, conforme a [Tabela 4.1](#), em termos de falsos positivos e falsos negativos. Por isso, conclui-se não ser viável aplicar o método de análise baseado no cálculo da precisão e revocação, visto que não se conhece *a priori* a situação de pares de código como plágio ou não-plágio.

Para superar essa limitação, cogitou-se delegar essa classificação para um conselho de professores experientes. Contudo a quantidade de pares e a subjetividade envolvida seriam impeditivas. Outra alternativa seria assumir o resultado de uma das ferramentas tradicionais, isoladamente, como referência para distinguir os pares. Contudo observa-se que não é incomum uma ferramenta ser melhor do que a outra em situações alternadas, assim uma única ferramenta não pode ser a referência para a análise de outra.

Em um segundo momento, refletiu-se sobre a possibilidade de empregar as 3 ferramentas que apresentam melhores resultados nos testes manuais. Quando ao menos duas apresentarem um resultado em comum, assumir-se-ia esse resultado como o “relevante”. Observa-se, entretanto, nas Figuras [B.11](#), [B.26](#), [B.16](#) que essa estratégia não seria adequada, pois poderia levar a contabilizar verdadeiros positivos como falsos positivos, e falsos positivos como verdadeiros positivos, o que seria absurdo.

Considerando que o JPlag e o MOSS são algoritmos cuja eficiência na detecção de similaridade é destacada pela comunidade científica, e que o SIM, assim como os dois anteriores, apresentou resultados satisfatórios em testes controlados (em que se conhecia *a priori* o status plágio ou não-plágio), neste trabalho, todos os três passaram a integrar um conjunto de ferramentas de controle em uma abordagem particularizada. Assim, partindo da hipótese que os resultados obtidos pelos três algoritmos, considerados em conjunto, podem ser confiáveis quando comparados a um quarto algoritmo, definiu-se um novo método para avaliação do Sherlock N-Overlap que se apoia nos resultados por elas gerados.

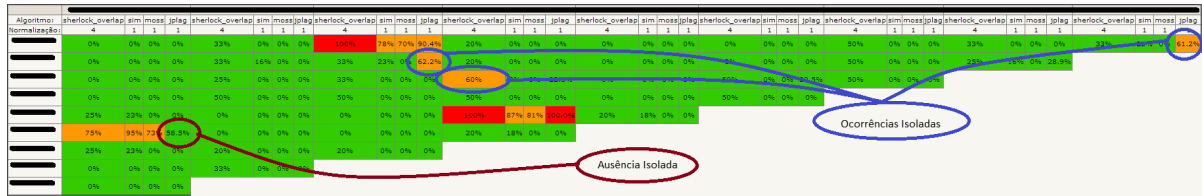
Esse método foi denominado **Método de Conformidade**.

Essa forma de avaliação representa uma alternativa devido à inexistência, ao menos em toda a literatura consultada até o momento em que essa dissertação foi escrita, de qualquer outro método de avaliação que não se baseie apenas em julgamentos subjetivos. O método proposto leva em consideração a conformidade entre os resultados das três ferramentas de referência (neste trabalho foram utilizados o JPlag, o MOSS e o SIM) e de uma ferramenta em análise (o Sherlock N-Overlap com normalização 3 e 4). Essa estratégia é fundamentada com base na contagem das medidas de precisão (Equação 4.1) e revocação (Equação 4.2). Enquanto essas medidas contabilizam os valores nas classes de atributos definidos na Tabela 4.1, analogamente, é necessário definir classes de valores de conformidade. Devem ser contabilizados:

- i. as **quantidades de ocorrências** de similaridade, correspondendo ao número de pares que determinada ferramenta contabilizou como similar para um determinado limiar predefinido;
- ii. as quantidades de **ocorrências (de similaridade) isoladas**, correspondendo à quantidade de pares em que determinada ferramenta foi a única a contabilizar similaridade para um determinado limiar;
- iii. as quantidades de **ausências (de similaridade) isoladas**, correspondendo à quantidade de pares em que determinada ferramenta foi a única a NÃO contabilizar similaridade, para um certo limiar.

As duas últimas medidas variam de acordo com o resultado das demais ferramentas.

A Figura 4.10 apresenta um exemplo de ocorrências e de ausências isoladas usando programas submetidos por uma turma de programação de 2012 (a identidade dos alunos foi ocultada). A tabela corresponde ao resultado retornado pelo sistema de Análise de Similaridade para a execução do Sherlock Overlap com normalização 4, JPLag, MOSS e SIM, os três últimos sem uso de normalização, e para um limiar de 60%. Todas as células com resultado inferior a esse limiar possuem a cor verde e a ocorrência relacionada não é contabilizada no relatório apresentado na Figura 4.11, que sumariza a contagem dos tipos de ocorrências identificadas para cada ferramenta.

Figura 4.10 – Print Screen da tabela de comparação**Figura 4.11** – Print Screen do relatório associado a Figura 4.10

Limiar: 60%	Algoritmos:	Overlap	N4	JPLAG	SIM	MOSS
	Total ocorrências:	4	4	3	3	
	Ocorrências Isoladas:	1	2			
	Ausências isoladas:		1			

De acordo com as Figuras 4.10 e 4.11, é possível observar adotando o limiar de 60% que, para o JPlag, foi contabilizada uma ausência isolada. Considerando que as demais ferramentas são confiáveis, pode-se inferir que este tipo de ocorrência é um falso negativo. Assim, quanto maior a quantidade de ausências isoladas, pior será considerado o desempenho de determinado algoritmo relativamente às ferramentas analisadas.

De outro modo, as ocorrências isoladas podem ser justificadas por duas situações distintas. Inicialmente, é imediato acreditar se tratar de um caso de falso positivo. Entretanto, considerando a fragilidade de alguns algoritmos na identificação de similaridades para algumas situações, é necessário considerar que pode se tratar de um verdadeiro positivo, mas que as outras ferramentas não foram capazes de identificar.

Cálculo da similaridade no Método de Conformidade

Como o conjunto das ocorrências isoladas são compostos por dois tipos de valores, para efeitos de cálculo, considera-se como total de verdadeiros positivos os pares que não são contabilizados como ocorrência isolada, conforme Equação 4.4.

$$Total\ de\ Verdadeiro\ Positivo = Total\ de\ Ocorrências - Ocorrências\ Isoladas \quad (4.4)$$

Usando o exemplo da Figura 4.12, para o JPlag, foram contabilizados 12 ocorrências, das quais 7 foram isoladas, ou seja, em 7 ocorrências, nenhuma das outras ferramentas também indicou similaridade. Tendo em vista que esta validação baseia-se na conformidade

com as demais ferramentas, pode-se considerar como verdadeiro positivo apenas 5 dessas ocorrências.

Figura 4.12 – Print Screen de relatório parcial de quantificação de ocorrências

Limiar: 60%	Algoritmos:	Overlap N3	JPLAG	SIM	MOSS
	Total ocorrências:	2	12	4	1
	Ocorrências Isoladas:	1	7		
	Ausências isoladas:	1			

No cálculo da precisão apresentado na [subseção 4.2.1 \(Equação 4.1\)](#), divide-se o total de pares verdadeiros positivos pelo total de pares recuperados, sendo que esse total de recuperados é igual à quantidade de verdadeiros positivos somada aos falsos positivos.

No Método de Conformidade, o total de pares recuperados é igual ao total de verdadeiros positivos somado às ocorrências isoladas, que são compostas por pares de verdadeiros positivos e falsos positivos. Logo, não é possível aplicar diretamente o cálculo original da precisão. Na aplicação direta da precisão original, seria considerado o total de pares recuperados como o total de ocorrências encontradas. Dessa forma, porém, não existiria inter-relação entre as ferramentas avaliadas. Melhores resultados são obtidos ao adotar um denominador único, sendo este o maior valor entre os verdadeiros positivos encontrados. Lembrando que por verdadeiros positivos considera-se os pares em comum de ao menos duas ferramentas, consequência da [Equação 4.4](#). Então, na precisão de conformidade, o valor máximo de verdadeiros positivos corresponde à estimativa dos valores que deveriam ter sido recuperados, logo, análogo ao total de pares recuperados considerado no método de cálculo tradicional. Assim, define-se precisão de conformidade, denominada P_{conf} , em que $P_{conf} \in [0, 1]$, pela [Equação 4.5](#).

$$P_{conf} = \frac{\text{Total de Verdadeiro Positivo}}{\max_{\text{entre as ferramentas}}(\text{Total de Verdadeiro Positivo})} \quad (4.5)$$

No cálculo da revocação apresentado na [subseção 4.2.1 \(Equação 4.2\)](#), divide-se o total de pares verdadeiros positivos pelo total de pares relevantes, calculado pelo somatório dos pares verdadeiros positivos com os falsos negativos. Na abordagem de conformidade, conforme discutido, consideram-se todas as ausências isoladas como falsos negativos.

Assim, define-se revocação de conformidade, denominada R_{conf} , onde $R_{conf} \in [0, 1]$, pela Equação 4.6.

$$R_{conf} = \frac{\text{Total de Verdadeiro Positivo}}{\text{Total de Verdadeiro Positivo} + \text{Total de Ausências Isoladas}} \quad (4.6)$$

A Tabela 4.2 compara as expressões da precisão e revocação nos métodos tradicional e de conformidade.

Tabela 4.2 – Comparação entre precisão e revocação tradicional e de conformidade

	Cálculo tradicional	Cálculo de conformidade
P	$\frac{\text{Verdadeiro Positivo}}{\text{Verdadeiro Positivo} + \text{Falso Positivo}}$	$\frac{\text{Total Ocor.} - \text{Ocor. Isoladas}}{\max_{\text{entre as ferramentas}}(\text{Total Ocor.} - \text{Ocor. Isoladas})}$
R	$\frac{\text{Verdadeiro Positivo}}{\text{Verdadeiro Positivo} + \text{Falso Negativo}}$	$\frac{\text{Total Ocor.} - \text{Ocor. Isoladas}}{(\text{Total Ocor.} - \text{Ocor. Isoladas}) + \text{Aus. Isolada}}$

Por fim, com a mesma definida na Equação 4.3, calcula-se a média harmônica para obter uma representação única dos valores obtidos com as Equações 4.5 e 4.6. Pode-se, com as adaptações propostas para o cálculo da precisão e da revocação, avaliar o desempenho relativo de uma ferramenta usando um conjunto de ferramentas de referência, reduzindo as consequência de não se conhecerem *a priori* os falsos positivos e negativos.

Figura 4.13 – Print Screen de relatório completo de quantificação de ocorrências

Limiar: 60%	Algoritmos:	Overlap N3	JPLAG	SIM	MOSS
	Total ocorrências:	2	12	4	1
	Ocorrências Isoladas:	1	7		
	Ausências isoladas:	1			
	Recuperado Relevante - Verdadeiro Positivo:	1	5	4	1
	Não recuperado relevante - Falso negativo:	1	0	0	0
	P	0.2	1	0.8	0.2
	R	0.5	1	1	1
	Média Harmônica	0.28571429	1	0.88888889	0.33333333

A Figura 4.13 exemplifica como o valor da média harmônica permite relacionar o desempenho entre as ferramentas estudadas com o uso do método de conformidade. No

exemplo, o JPlag registrou a maior quantidade de ocorrências e também a maior quantidade de ocorrências isoladas, não tendo apresentado ausências isoladas (falsos negativos). Observando-se o cálculo da média harmônica, tem-se que o JPlag obteve o melhor resultado, com o valor máximo 1, seguido pelo SIM, MOSS e Sherlock N-Overlap com normalização 3. O MOSS registrou menos ocorrências, apenas uma, obtendo média harmônica 0.33, superando o Sherlock N-Overlap com normalização 3, cuja média harmônica foi apenas 0.28. Apesar de ter registrado mais ocorrências que o MOSS, o registro de uma ausência isolada o colocou o Sherlock N-Overlap com Normalização 3 em situação inferior ao MOSS.

No [Capítulo 5](#) são apresentados os resultados comparativos entre o Sherlock N-Overlap (com as normalizações 3 e 4) e as demais ferramentas. As comparações foram realizadas tanto com o uso do método tradicional (para um grupo de códigos com plágios propositalmente produzidos) como com o uso do método de conformidade para códigos de alunos de práticas de programação.

Análise de Resultados

Visando validar o algoritmo Sherlock N-Overlap, neste capítulo é apresentada a avaliação comparativa entre os resultados de todos os algoritmos estudados. Para compreendê-los, foram realizadas duas abordagens. Na primeira, a análise sobre um conjunto de códigos controlados, que foram plagiados propositalmente. Conhecendo a situação de cada par de código previamente, foi utilizada para essa primeira abordagem o método tradicional, descrito na [subseção 4.2.1](#). Na segunda abordagem, foram usados códigos produzidos por alunos do curso de Engenharia de Teleinformática, em atividades práticas de programação. Os códigos usados na segunda abordagem foram registrados no Moodle por meio da ferramenta BOCALAB ([FRANÇA; SOARES, 2011](#)) ao longo do ano de 2012. Para esta última abordagem, foi utilizado o método de conformidade, conforme [subseção 4.2.2](#).

5.1 Primeira Análise: códigos plagiados propositalmente

Apesar dos esforços empreendidos, não foi encontrado qualquer conjunto de códigos recomendado ou disponível em repositório público para avaliação de ferramentas de detecção de plágio. [Duric e Gasevic \(2012\)](#) também registrou a mesma dificuldade em encontrar códigos desenvolvidos especificamente para essa finalidade ou produzidos por alunos com classificação previamente conhecida (ex.: plágio/não-plágio, similar/não-similar). Assim, foi necessário criar um repositório de códigos próprio, de forma similar ao trabalho realizado

por Duric e Gasevic (2012), Prechelt, Malpohl e Philippsen (2002), Burrows, Tahaghoghi e Zobel (2007) e Mozgovoy, Karakovskiy e Klyuev (2007).

5.1.1 Cenário de experimentação

Os códigos utilizados foram gerados considerando-se as principais modificações realizadas pelos alunos para dissimular a cópia de código-fonte, conforme apresentado na seção 2.1.

Para a análise de códigos gerados manualmente, portanto, foram utilizados códigos gerados adotando-se os seguintes critérios:

- ▶ 0. Cópia, sem alterações, do código original;
- ▶ 1. Inclusão/modificação de comentários;
- ▶ 2. Mudança de nomes de identificadores;
- ▶ 3. Troca de posição de variáveis e funções;
- ▶ 4. Mudança de escopo de variável;
- ▶ 5. Alteração na indentação;
- ▶ 6. Inclusão de informação sem relevância: incluir bibliotecas, variáveis sem utilização, comentários, etc.;
- ▶ 7. Rearranjo de expressões matemáticas;
- ▶ 8. Todas as alterações anteriores combinadas;
- ▶ 9. Código gerado sem plágio.

Os conjuntos de códigos foram desenvolvidos por diferentes alunos de graduação que participaram como monitores das práticas de laboratório. De maneira a guardar o sigilo sobre a identidade dos alunos, os mesmos são identificados neste trabalho como programador 1, 2 e 3. Embora todos os alunos envolvidos tenham ampla capacidade para solucionar os problemas propostos, classificou-se a experiência dos alunos como baixa, média ou alta em função de suas experiências pregressas (por exemplo, participação dos

mesmos em projetos acadêmicos e profissionais). A importância dessa classificação é que ela pode ser determinante no momento da dissimulação das modificações realizadas com o objetivo de plagiar o código original, o que pode interferir nos indicadores de similaridade apresentados como resultado da comparação pelos algoritmos usados. Os enunciados dos códigos desenvolvidos estão no Apêndice A.

Cada programador desenvolveu dois códigos bases: um menor, com cerca de 10-20 linhas, e outro maior, com cerca de 60-70 linhas. Em seguida, cada um gerou um novo código para cada uma das 8 modificações aplicadas no seu respectivo código base e sem consultar os colegas. Além disso, também ocorreu uma permuta entre os enunciados dos códigos para cada um desenvolver um código não plagiado (critério 9). Os códigos modificados não contêm modificações lógicas, compilando corretamente e executando para mesmas entradas, e gerando os mesmos resultados dos códigos originais. Estes experimentos foram executados gerando resultados para os algoritmos Sherlock (PIKE; LOKI, 2013) e Sherlock N-Overlap com as normalizações 3 e 4. Também foram gerados resultados para as ferramentas SIM, MOSS e JPlag, aplicando-se apenas a normalização 1, visto que elas já possuem pré-processamentos específicos.

Nesses experimentos, como já se conhece *a priori* o *status* plágio/não-plágio dos códigos utilizados, é esperada a alta similaridade em todos os resultados, exceto para o código gerado pelo critério 9. Para cada modificação, o mesmo código é testado com todas as ferramentas nas configurações mencionadas. As representações dos resultados mostrados neste capítulo foram geradas pela ferramenta descrita no Capítulo 4.

5.1.2 Descrição do processo de aplicação do método tradicional para obtenção dos dados para análise em códigos gerados artificialmente

A Figura 5.1 apresenta o resultado da comparação dos códigos médios desenvolvidos pelo programador 1, construídos conforme especificado na subseção anterior, mostrando os índices de semelhança (em percentual) que foram verificados pelas diversas ferramentas em análise entre os códigos plagiados e o código original. É importante ressaltar que a variação do parâmetro Z, que define a granularidade, afeta apenas os resultados para as versões do Sherlock, visto ser um parâmetro particular deste algoritmo. Os resultados das ferramentas SIM, MOSS e JPlag estão na mesma tabela para efeitos de comparação dos

resultados finais em relação ao Sherlock configurado para um valor de Z específico. No Apêndice B estão dispostas todas as tabelas com dados brutos dos códigos propositalmente plagiados.

A Tabela 5.1 apresenta os cálculos da precisão (P), da revocação (R) e da média harmônica (F) para cada ferramenta, considerando-se, em cada linha, um limiar que estabelece um percentual mínimo para descarte por irrelevância. Dessa maneira, são considerados não relevantes os resultados inferiores a 90% na primeira linha, os resultados inferiores a 85% na segunda linha e assim sucessivamente. No Apêndice C estão presentes todas as tabelas com os valores da precisão, revocação e média harmônica geradas a partir das tabelas do Apêndice B.

Figura 5.1 – Dados brutos - código médio - programador 1 (Z=0)

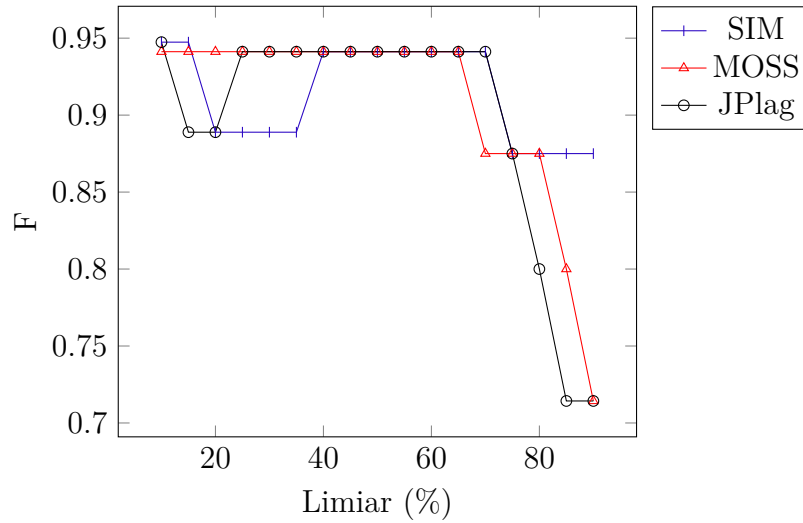
	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	0%	94%	34%	97%	97%	95%	100.0%
mod3.c	88%	78%	94%	89%	94%	83%	70.6%
mod4.c	95%	89%	96%	92%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	90%	100%	98%	100%	99%	98%	99.1%
mod7.c	72%	65%	80%	74%	71%	67%	78.3%
mod8.c	0%	27%	22%	39%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	27%	39%	0%	21.4%

Tabela 5.1 – Dados contabilizados - código médio - programador 1 (Z=0)

Limiar (%)	Sherlock Norm3			Sherlock Norm4			Overlap Norm3			Overlap Norm4			SIM			MOSS			JPlag		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.5556	0.714286	1	0.5556	0.7143	1	0.6667	0.8	1	0.6667	0.8	1	0.7778	0.875	1	0.5556	0.7143	1	0.5556	0.7143
85	1	0.6667	0.8	1	0.6667	0.8000	1	0.6667	0.8	1	0.7778	0.875	1	0.7778	0.875	1	0.6667	0.8000	1	0.5556	0.7143
80	1	0.6667	0.8	1	0.6667	0.8000	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.8750	1	0.6667	0.8000
75	1	0.6667	0.8	1	0.7778	0.8750	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.8750	1	0.7778	0.8750
70	1	0.7778	0.875	1	0.7778	0.8750	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.7778	0.8750	1	0.8889	0.9412
65	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
60	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
55	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
50	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
45	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
40	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
35	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	1	1	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
30	1	0.7778	0.875	1	0.8889	0.9412	1	0.8889	0.9412	1	1	1	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
25	1	0.7778	0.875	1	1	1	1	0.8889	0.9412	0.9	1	0.94737	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
20	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.8889	0.8889	0.8889	1	0.8889	0.9412	0.8889	0.8889	0.8889
15	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.9	1	0.9474	1	0.8889	0.9412	0.8889	0.8889	0.8889
10	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.9	1	0.9474	1	0.8889	0.9412	0.9000	1	0.9474

Assim, na Tabela 5.1, como o valor da precisão varia com a quantidade de falsos positivos, a maioria dos resultados das respectivas colunas é 1 (um), pois o universo de

Figura 5.2 – Média Harmônica calculada por limiar para os algoritmos SIM, MOSS e JPlag para os códigos médios do programador 1 com $Z=0$



códigos em estudo nesta seção, exceto o último, é constituído apenas de códigos plagiados. Ocorre redução deste índice apenas com o uso de limiares muito baixos, como no caso do Sherlock Overlap com a normalização 4 com limiar em 25%, o SIM com limiar em 35% e o JPlag com limiar em 20%. Observa-se que os valores da revocação tendem a crescer à medida que se diminui o limiar, pois amplia-se a quantidade de pares recuperados. Por fim, a média harmônica tende a estabilizar em um valor comum abaixo de determinados limiares, variando de algoritmo para algoritmo, com tendência a aumentar à medida que mais valores corretos são recuperados e a diminuir caso contrário.

A partir dos dados dessa tabela, são plotados os resultados da média harmônica (F), conforme a [Equação 4.3](#), nos gráficos presentes na [Figura 5.2](#), para os algoritmos SIM, MOSS e JPlag, e na [Figura 5.3](#) para os algoritmos Sherlock, Sherlock Overlap, ambos com normalizações 3 e 4. Com esses gráficos é possível avaliar, para cada limiar, o comportamento de cada ferramenta.

Por exemplo, na [Figura 5.2](#), para limiares superiores a 80%, a ferramenta SIM tem melhor desempenho. Para limiares inferiores a 40%, o SIM apresenta uma forte queda no valor da média harmônica. Isto ocorre devido a atribuição de similaridade para o arquivo com não plágio. Este falso positivo pode ser observado na última linha da [Figura 5.1](#), onde se observa que ele vai influenciar negativamente todos os valores do SIM para os limiares inferiores a 40%. O mesmo ocorre com o JPlag para limiares inferiores a 25%. No caso do MOSS, por outro lado, por não contabilizar falsos positivos, não é apresentada diminuição

Figura 5.3 – Média Harmônica calculada por limiar para as versões do Sherlock e para os códigos médios do programador 1 com $Z=0$

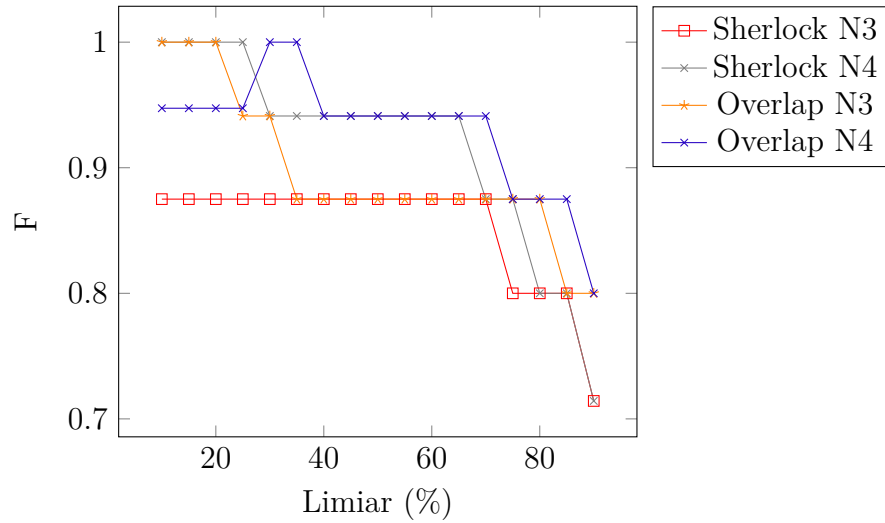
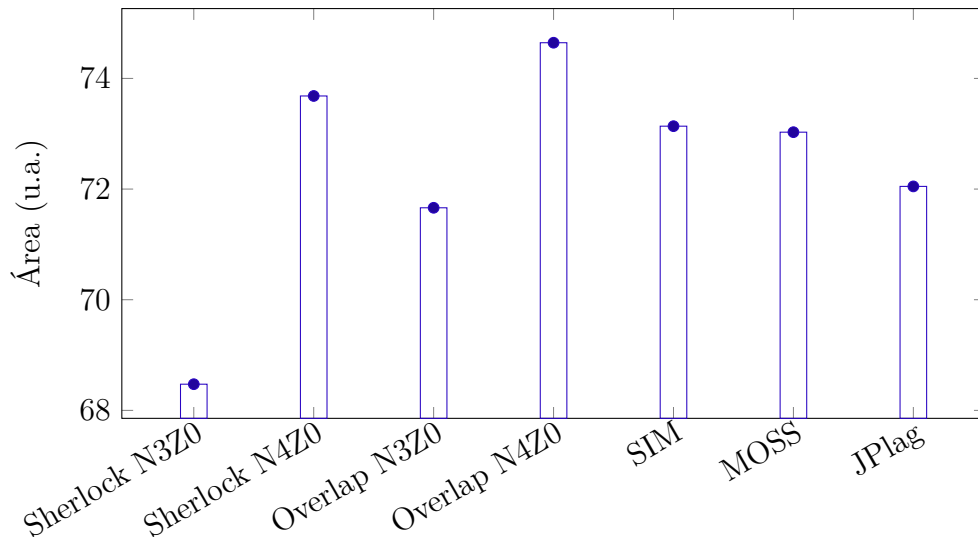
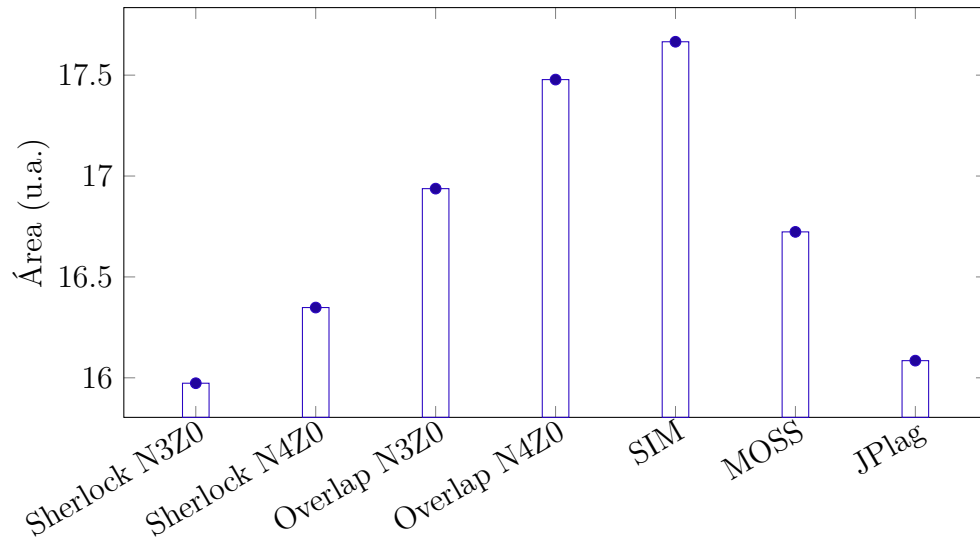


Figura 5.4 – Áreas para código médio - programador 1 ($Z=0$, $>10\%$)



do valor da média harmônica ao se reduzir o limiar.

Na [Figura 5.3](#), observa-se que o Sherlock com a normalização 3 tem o pior desempenho, pois, para todos os limiares, apresenta resultados abaixo das outras 3 abordagens. O Sherlock N-Overlap com normalização 4, para limiares superiores a 30%, apresenta sempre resultado igual ou superior às outras versões. Contudo, o Sherlock N-Overlap com normalização 4 é afetado por um falso positivo e sofre queda para limiares inferiores a 30%, sendo então superado pelo Sherlock com normalização 4 e Sherlock N-Overlap com normalização 3. A ocorrência de falsos positivos pode ser tolerada, desde que não ocorra com alta frequência e que seus valores fiquem restritos a baixos limiares.

Figura 5.5 – Áreas para código médio - programador 1 ($Z=0$, $>70\%$)

De modo geral, nota-se que o Sherlock com a normalização 3 apresenta resultados inferiores às demais abordagens. Entretanto, é necessário um pouco mais de atenção para afirmar algo sobre o desempenho do Sherlock N-Overlap com normalização 3 e do Sherlock com normalização 4. Além disso, é importante considerar que os dados das Figuras 5.2 e 5.3 referem-se apenas aos resultados da execução do Sherlock e Sherlock N-Overlap com o parâmetro $Z=0$. Como avaliar o desempenho de todas as abordagens ao variar o valor de Z ?

Para comparar todos esses resultados de uma forma objetiva, conjunta e evitando gráficos com excesso de informação, propõe-se, como métrica de qualidade, calcular a área sob a curva de interpolação entre os valores da média harmônica, admitindo-se que na região inter-limiar calculados, essa média cresça ou decresça de maneira linear. Assim, dados dois pares ordenados subsequentes: $P_A(x_i, y_i)$ e $P_B(x_{i+1}, y_{i+1})$, a área é calculada pelo somatório do produto entre os módulos das diferenças entre os respectivos valores de x e y , dividido por 2, conforme a Equação 5.1.

$$\hat{Area} = \sum_{i=1}^n \frac{|x_i - x_{i-1}| \times |y_i - y_{i-1}|}{2} \quad (5.1)$$

No gráfico da Figura 5.4 são comparadas as áreas referentes ao gráfico das Figuras 5.2 e 5.3. É possível observar de forma imediata que o Sherlock N-Overlap com normalização 4 detém o melhor resultado dentre as versões do Sherlock, para o parâmetro $Z=0$, sendo superior, inclusive, aos algoritmos SIM, MOSS e JPlag. Confirma-se também o desempenho

ruim do Sherlock com normalização 3.

Para efeitos de comparação do desempenho de vários algoritmos, o cálculo das áreas pode ser realizado com a adoção de um limiar único de referência, por exemplo, ignorando resultados abaixo de 70%, ou utilizado-se o acumulado da faixa completa de limiares, adotando-se o menor limiar: 10%. Como os falsos positivos tendem a ocorrer com mais frequências com limiares baixos, é interessante considerar no cálculo da área ampla faixa de limiares para análise comparativa entre diferentes algoritmos, o que permite levar em conta a ocorrência de falsos positivos. Por outro lado, para a análise de práticas de laboratório, não é prático e nem necessário ao professor considerar faixas com limiares baixos, visto que, a caracterização do plágio ocorre na ocorrência da alta similaridade. Logo, também é importante avaliar o desempenho das ferramentas para faixa de limiares altos. Assim, a [Figura 5.4](#) não é suficiente.

A [Figura 5.5](#) apresenta resultados para os mesmos códigos-fonte e algoritmos que serviram de base para a [Figura 5.4](#), mas considerando apenas os índices de similaridades superiores a 70%. A modificação dos resultados é coerente aos dados apresentados no gráfico da [Figura 5.2](#), em que se percebe que o SIM, para limiar de 70%, tem média harmônica constante entre 0.85 e 0.9, superando o MOSS e o JPlag. Apenas para limiar de 80% o MOSS acompanha esse valor, superando o JPlag.

As Figuras [5.4](#) e [5.5](#) apresentam diferentes perspectivas de análise. Contudo, é inquestionável que o pior resultado nas duas perspectivas é o Sherlock com normalização 3. Por outro lado, considerando as configurações do teste apresentado, percebe-se que o Sherlock N-Overlap com a normalização 4 apresenta melhores resultados com o uso de faixas amplas de limiares, enquanto que considerando-se apenas limiares altos, o SIM apresenta maior regularidade.

Em resumo, as etapas enumeradas em seguida descrevem o fluxo de atividades necessárias para obter os resultados apresentados. Na próxima seção elas são aplicadas para avaliar o parâmetro Z do Sherlock.

- Etapa 1. Execução dos algoritmos e obtenção das tabelas com as similaridades calculadas por cada algoritmo (conforme o exemplo da [Figura 5.1](#)).

- ▶ Etapa 2. Contagem dos falsos positivos e falsos negativos. Cálculo da precisão, revocação e da média harmônica (conforme o exemplo da [Tabela 5.1](#)).
- ▶ Etapa 3. Geração dos gráfico da média harmônica (conforme os exemplos das Figuras [5.2](#) e [5.3](#)).
- ▶ Etapa 4. Cálculo das áreas de cada algoritmo da Etapa 3, para as similaridades superiores a 10% e a 70% (conforme os exemplos das Figuras [5.4](#) e [5.5](#)).

5.1.3 Análise do parâmetro zerobit do Sherlock

Segundo os desenvolvedores do Sherlock, o parâmetro Z (zerobit), controla a granularidade da comparação. Quanto menor o valor, mais lenta é a comparação e o resultado mais próximo da precisão teórica. A experiência de utilização do Sherlock sugere que a variação do Z altera significativamente os resultados. Nesta análise é investigado o quanto o parâmetro Z interfere na qualidade dos resultados, ao mesmo tempo em que compara-se o desempenho das ferramentas de referência (JPlag, MOSS e SIM) e as versões propostas (Sherlock N-Overlap com normalizações 3 e 4).

O valor zerobit funciona como um critério para selecionar assinaturas para a fase de comparação. Quanto menor o zerobit, mais assinaturas serão selecionadas para essa etapa. Assim, para códigos pequenos, um zerobit alto pode significar a eventual incapacidade de efetuar a comparação por falta de assinaturas em quantidade suficiente. Isto é potencializado com a normalização 4, pois, devido à eliminação de grande quantidade de informação nos códigos comparados, o número de assinaturas geradas para comparação é bastante reduzido.

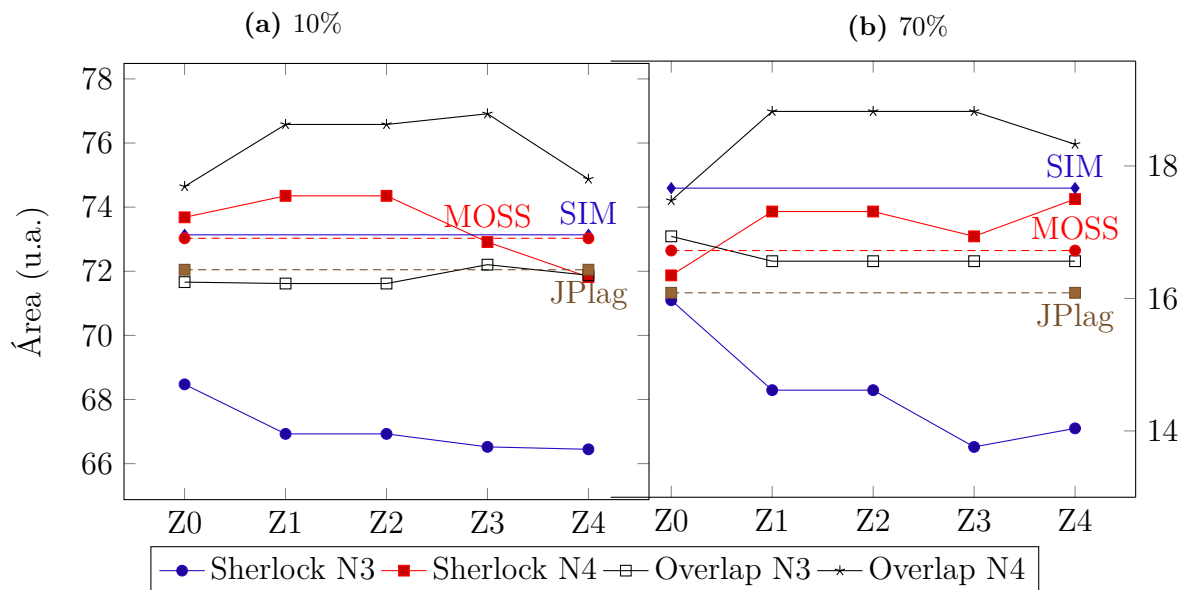
Códigos médios

Os gráficos apresentados nesta seção, a exemplo do que é encontrado na [Figura 5.6](#), contêm os resultados da área sob a curva da média harmônica entre a precisão e a revocação, conforme a variação do parâmetro Z do Sherlock, e registra os dados referentes ao conjunto de teste desenvolvido pelo programador 1. Cada figura é composta por dois gráficos, o primeiro contém o acumulado para todas as porcentagens, e o outro apenas o acumulado para o limiar de 70%, conforme discutido na seção anterior. Os resultados do SIM, MOSS

e JPlag estão representados no gráfico por uma linha horizontal, visto que não variam em função de Z . Nos Apêndices B e C estão listadas todas as tabelas utilizadas para obter os resultados a seguir.

Nos gráficos da Figuras 5.6a e da 5.6b, observa-se pouca variação dos resultados de cada ferramenta ao variar o valor de Z , ao mesmo tempo em que existe um distanciamento claro entre as linhas de cada algoritmo, o que sugere uma relação de melhoria progressiva entre as versões do Sherlock e as normalizações. Em consonância com a Figura 5.4, o desempenho ruim do Sherlock com normalização 3 é mostrado na Figura 5.6a não só para $Z=0$, mas também para todos os valores de Z analisados. Contudo, o Sherlock com normalização 4 apresenta uma melhora expressiva, sendo o bastante para se equiparar ao desempenho do JPlag. A normalização 3 usada com o Sherlock N-Overlap, configurado com Z igual 0, 1 e 2, também apresenta resultados bons, ultrapassando o desempenho do MOSS e do SIM. Por fim, Sherlock N-Overlap com normalização 4 tem desempenho nitidamente superior a todos os casos avaliados. Já com relação a Figura 5.6a, pode-se entender as informações aparentemente incongruentes encontradas na Figura 5.5. Considerando-se diferentes faixas de limiares, se, para $Z=0$, as informações dos gráficos 5.4 e 5.5 são divergentes, ao configurar o Z com outros valores, observam-se poucas mudanças nos resultados apresentados nas Figuras 5.6a e 5.6b. Dessa forma, para esse conjunto de teste, considerar apenas limiares altos não representa grandes diferenças de interpretação dos resultados.

Figura 5.6 – Áreas para código médio - programador 1

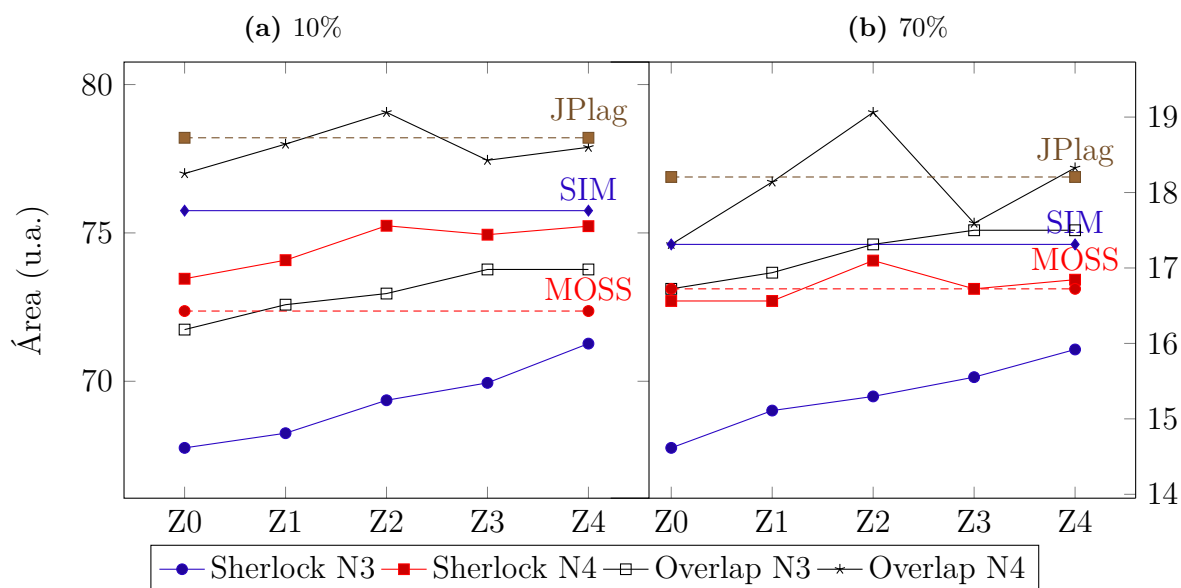
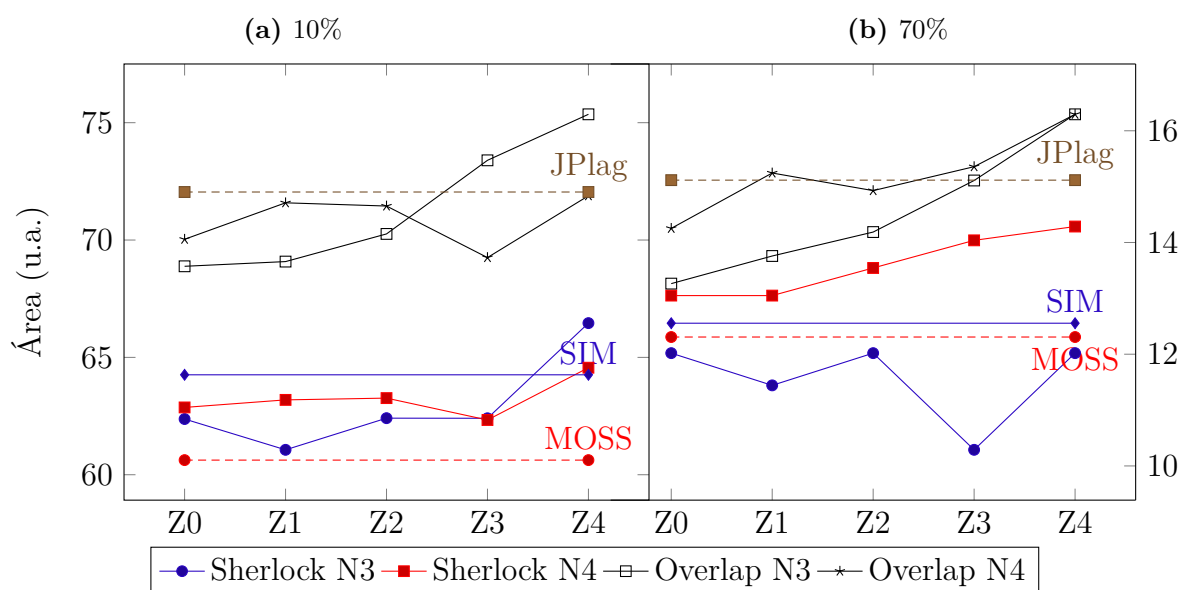


As relações de melhoria progressiva entre as versões do Sherlock também são observadas na [Figura 5.6b](#) e nos gráficos da [Figura 5.7](#), que apresenta os resultados extraídos para os cálculos feitos com os códigos plagiados pelo programador 2. Na [Figura 5.7a](#) observa-se melhora significativa da normalização 4 em relação a normalização 3, tanto para o Sherlock original quanto para o Sherlock N-Overlap. De modo geral, outra melhora positiva ocorre com o Sherlock N-Overlap em relação ao Sherlock original. Mantendo a coerência com resultados anteriores, o Sherlock com a normalização 3 apresenta o pior desempenho, sendo que a normalização 4 o melhora significativamente. O Sherlock N-Overlap com normalização 3 apresenta melhora em relação à versão original do algoritmo, e o Sherlock N-Overlap com normalização 4 é superior a todos os demais. Em relação à [Figura 5.6a](#), destaca-se a comparação do JPlag com o Sherlock N-Overlap com normalização 4, observando-se desempenho melhor para este último apenas quando configurado com $Z=2$, embora sendo os demais resultados ainda próximos aos do JPlag. Outra diferença entre os resultados encontrados para o plágio dos programadores 1 e 2 é um maior distanciamento entre os valores das ferramentas de referência (JPlag, MOSS e SIM). Enquanto na [Figura 5.6a](#) a variação era de 1 unidade de área (ua), na [Figura 5.7a](#) ([Figura 4.7a](#)) a variação é de 6 ua. De forma análoga, a [Figura 5.7b](#) conserva os extremos entre Sherlock com normalização 3 e Sherlock N-Overlap com normalização 4. Já no caso do Sherlock com normalização 4 e o Sherlock N-Overlap com normalização 3 ocorre uma inversão em relação à [Figura 5.7a](#).

Diferentemente das [Figuras 5.6](#) e [5.7](#), nas quais observa-se concentração em quatro níveis, a [Figura 5.8a](#) apresenta apenas duas níveis de concentração obtidos com o Sherlock e o Sherlock N-Overlap, como se a diferença de normalização não tivesse impactado em melhoria do resultado.

A análise das [Figuras 5.6](#) e [5.7](#) sugere uma relação de correspondência entre a variação do Z para os gráficos com os limiares de 10% e 70%. Especialmente na [Figura 5.8](#) essa tendência é rompida, principalmente para os algoritmos com a normalização 4, para os valores de Z iguais a 3 e 4. Enquanto na [Figura 5.8b](#), o Sherlock com normalização 4 e o Sherlock N-Overlap com normalização 4 apresentam crescimento, na [Figura 5.8a](#) para os valores de Z iguais a 3 e 4, essa tendência de crescimento não se mantém.

Antes de justificar a razão desse descompasso, é interessante analisar a [Tabela 5.2](#), que apresenta a quantidade de assinaturas que são efetivamente comparadas em função da

Figura 5.7 – Áreas para código médio - programador 2**Figura 5.8** – Áreas para código médio - programador 3**Tabela 5.2** – Contabilização de assinaturas processadas do código original médio de cada programador

	z=0			z=1			z=2			z=3			z=4		
	Original	N3	N4	Original	N3	N4	Original	N3	N4	Original	N3	N4	Original	N3	N4
Programador 1	241	256	112	129	124	54	75	64	23	35	33	13	17	15	5
Programador 2	385	182	88	221	92	46	96	47	22	59	27	14	36	20	8
Programador 3	105	66	31	56	34	16	37	15	7	15	6	4	10	3	2

normalização e do valor de Z . Cada um dos 10 códigos desenvolvidos por cada programador dá origem a um conjunto de assinaturas diferentes. A [Tabela 5.2](#), que é preenchida a partir de uma modificação no código do Sherlock, contabiliza apenas as assinaturas do código original por questões de simplificação. A quantidade de assinaturas dos demais códigos é proporcional ao tamanho do código, sendo todos eles comparados com o código original. Para o programador 1, com $Z=0$, o código sem normalização calculou similaridade por meio da comparação de 241 assinaturas. A normalização 3, em relação ao código sem normalização, apresenta um aumento na quantidade de assinaturas. Eventualmente a normalização 3 pode aumentar o número de “palavras” do código, o que aumenta também o número de assinaturas. De outro modo, a normalização 4 sempre apresenta menos assinaturas do que a normalização 3. Na [Tabela 5.2](#) verifica-se que o aumento do Z também diminui o número de assinaturas.

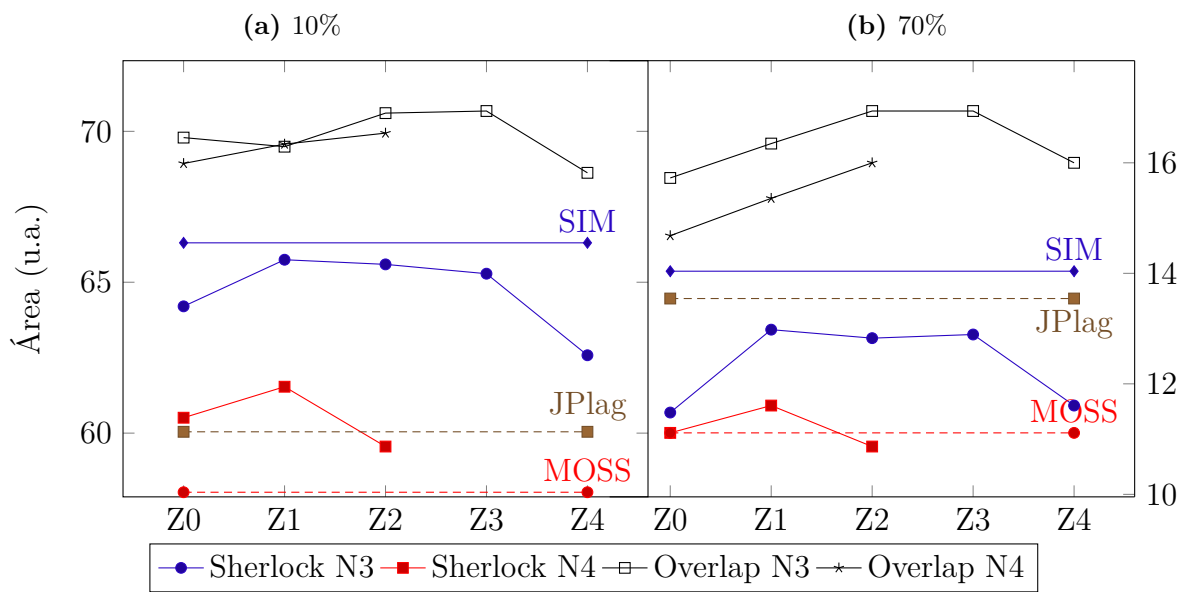
Assim, constata-se que, para o programador 3, a quantidade relativa de assinaturas decresce significativamente a ponto de o código original ser representado apenas por 6 e 4 assinaturas para $Z=3$, e 3 e 2 assinaturas, para $Z=4$, respectivamente, para as normalizações 3 e 4. Logo, com tão poucas assinaturas, não é possível garantir um bom desempenho na comparação dos códigos, o que justifica a queda obtida com a normalização 4 que é observada na [Figura 5.8a](#). Contudo, mesmo com pouquíssimas assinaturas, para limiares altos, ainda é possível obter resultados que conseguem alcançar ou superar todas as ferramentas de referência, conforme observado na [Figura 5.8](#).

Códigos pequenos

Nos gráficos da [Figura 5.9](#), que apresenta os resultados extraídos para os códigos plagiados pelo programador 1, percebe-se que, para o Sherlock original e Sherlock N-Overlap com a normalização 4, destaca-se a falta de informação quando o valor de Z é igual ou superior a 3. Os dados brutos referentes a esses valores de Z estão no Apêndice B, na [Figura B.19](#) e na [Figura B.20](#), nas quais se observa que as colunas referentes a esses algoritmos estão com valores -1% ou 0%. As células cinzas com valor -1%, representam a ausência de assinaturas, e as células verdes com 0%, representam que não foram geradas assinaturas em quantidade suficiente para obter um resultado mínimo. Constata-se que a progressiva diminuição do código gerado pelo aumento do parâmetro Z , associado com

a redução do código gerado pela normalização 4, provocou a insuficiência de informação para se gerar assinaturas em quantidade necessária para a comparação. A diminuição do resultado quando $Z=4$, observada para o Sherlock com normalização 3 e Sherlock N-Overlap com normalização 3, também é consequência dessa limitação na quantidade de assinaturas. Dessa forma, conclui-se que a [Figura 5.9](#) contém a representação do limite de informação que o Sherlock pode trabalhar com segurança. Logo, para códigos com até 20 linhas, valores de Z superiores a 2 não são recomendados.

Figura 5.9 – Áreas para código pequeno - programador 1



Contudo, em ambos os gráficos da [Figura 5.9](#), observa-se que o Sherlock N-Overlap com normalização 4 supera as ferramentas de referência quando o valor de Z é igual ou inferior a 2, sendo o Sherlock N-Overlap com normalização 3 superior para todos os valores de Z .

A [Tabela 5.3](#) é uma atualização da [Tabela 5.2](#) para contabilizar as assinaturas processadas a partir do código original pequeno. Para o programador 1, observa-se que a quantidade de assinaturas na normalização 4 é menos da metade da quantidade contabilizada na normalização 3, a ponto de zerar para valores de Z igual a 3 e 4, o que impossibilita a comparação, em consonância com a [Figura 5.9](#).

Uma outra situação indesejada pode ocorrer quando se tem código muito pequeno e a seleção de assinaturas pelo algoritmo de comparação capturar somente as partes do código que são diferentes ou iguais, potencializando resultados extremos para baixo ou

Tabela 5.3 – Contabilização de assinaturas processadas do código original pequeno de cada programador

	z=0			z=1			z=2			z=3			z=4		
	Original	N3	N4	Original	N3	N4	Original	N3	N4	Original	N3	N4	Original	N3	N4
Programador 1	66	50	20	23	29	12	11	12	3	6	5	0	3	3	0
Programador 2	240	66	31	112	34	16	55	15	7	27	6	4	12	3	2
Programador 3	36	47	16	12	18	9	2	12	4	2	6	1	2	2	1

Figura 5.10 – Áreas para código pequeno - programador 2

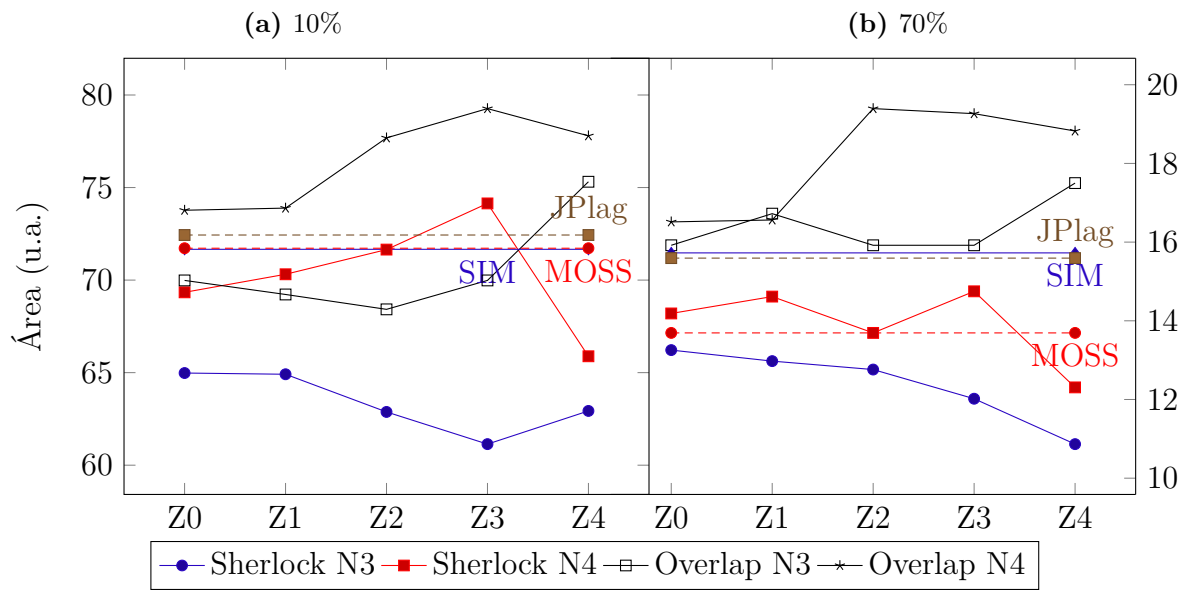
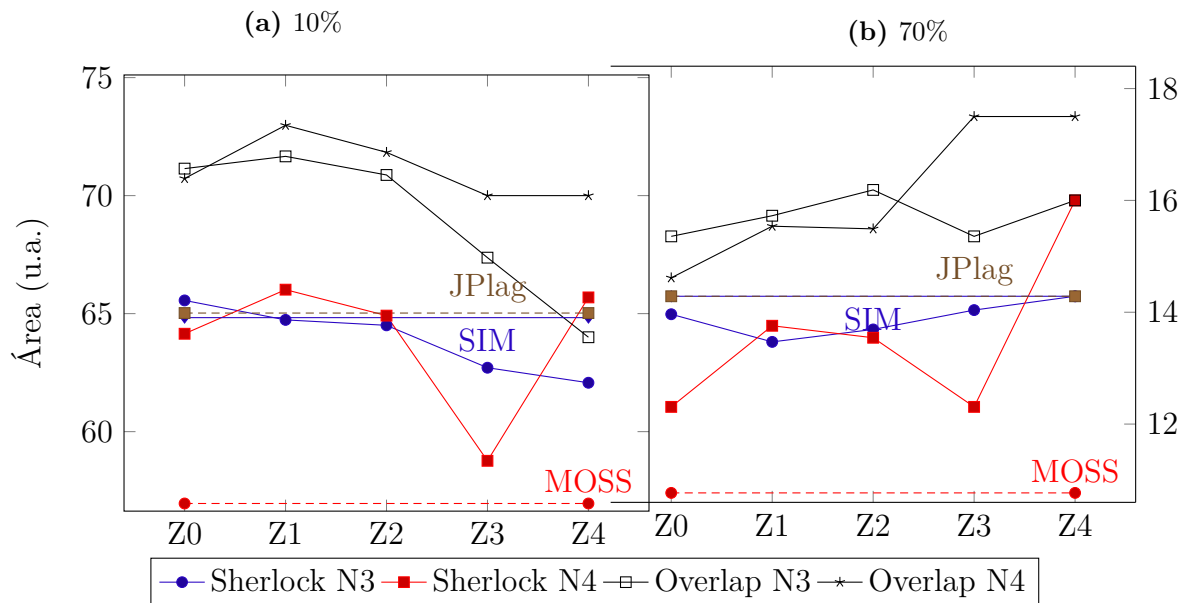


Figura 5.11 – Áreas para código pequeno - programador 3



para cima (falsos positivos ou falsos negativos).

Tendo em vista o limite de informação tratado anteriormente, para o Sherlock com normalização 4, nas Figuras 5.10a e 5.10b e nas Figuras 5.11a e 5.11b, na transição de $Z=3$ para $Z=4$, observa-se uma variação abrupta. Esse tipo de variação não ocorre, proporcionalmente, nos outros conjuntos de códigos estudados, o que evidencia ser uma limitação do Sherlock calcular similaridade com redução na quantidade de informação. De fato, de acordo com a Tabela 5.3, para o programador 2, com a normalização 4 e $Z=3$, tem-se 4 assinaturas e com $Z=4$ apenas 2, e para o programador 3, quando $Z=3$ e $Z=4$, tem-se apenas uma assinatura. Logo, tão poucas assinaturas não proporcionam base de comparação confiável, o que constitui um problema na utilização do Sherlock.

Apesar das limitações discutidas, observa-se que os resultados ainda apontam desempenho superior para o Sherlock N-Overlap quando comparado às ferramentas de referência, tanto na Figura 5.10 quanto na Figura 5.11.

Análise de falsos positivos e falsos negativos com códigos propositalmente plagiados

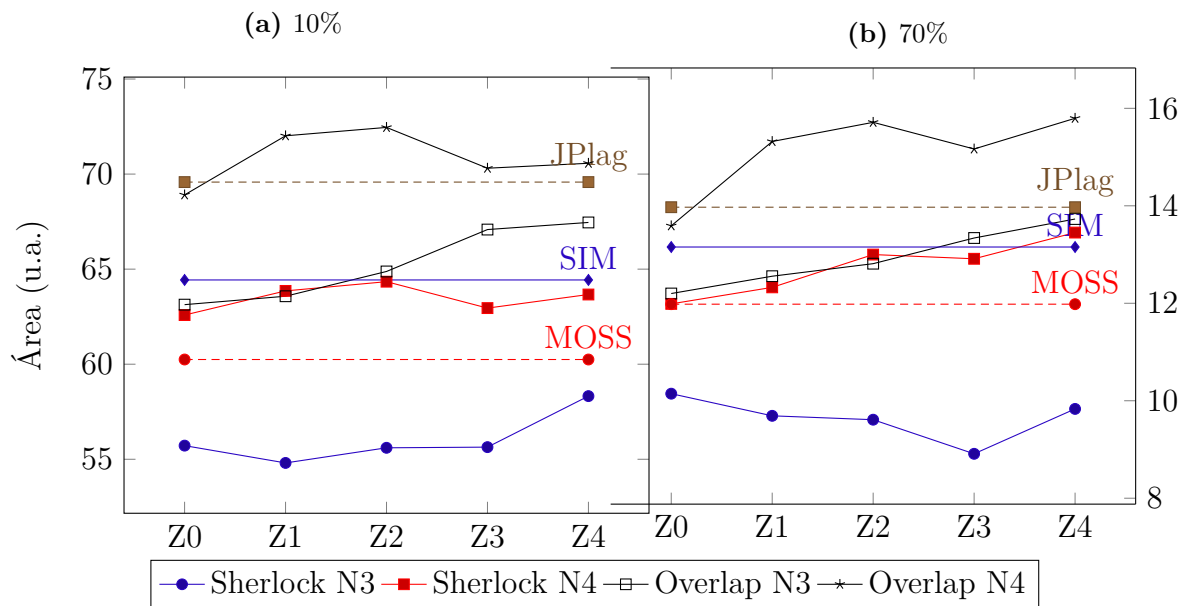
Os resultados apresentados até aqui consideram um pequeno universo de pares de controle para contabilização de resultados falsos. Para avaliar o desempenho frente a ocorrências de falsos positivos e falsos negativos, é necessário ampliar a quantidade de códigos com características semelhantes que são comparados simultaneamente. Assim, reuniu-se, para cada tipo de código (médio, pequeno), os 3 conjuntos de códigos produzidos e foram realizadas duas submissões com 33 arquivos em cada, dos quais coexistem 3 códigos originais, 24 códigos plagiados em relação ao respectivo código original e 3 códigos não plágio. Todos os 33 códigos foram comparados 2 a 2, assim os códigos originais, plagiados e os não-plágios desenvolvidos pelos programadores 1, 2 e 3 foram comparados em conjunto. Uma eventual indicação de similaridade, por exemplo, do código original do programador 1 com o código original do programador 2, é contabilizada como falso positivo.

É importante frisar que os dados deste experimento não podem ser considerados como um somatório ou uma média dos resultados dos testes anteriores, sendo esta, portanto uma nova avaliação. Com ela, existe a possibilidade de concluir a negação de todos os testes anteriores, pois é possível que os resultados de falsos positivos ou falsos negativos

superem os resultados de verdadeiros positivos para algum algoritmo, enquanto nos testes anteriores isso não era possível dada a pequena quantidade de códigos não-plágio. Dessa forma, cria-se um grande conjunto de códigos com características aproximadas em termos de complexidade e número de linhas.

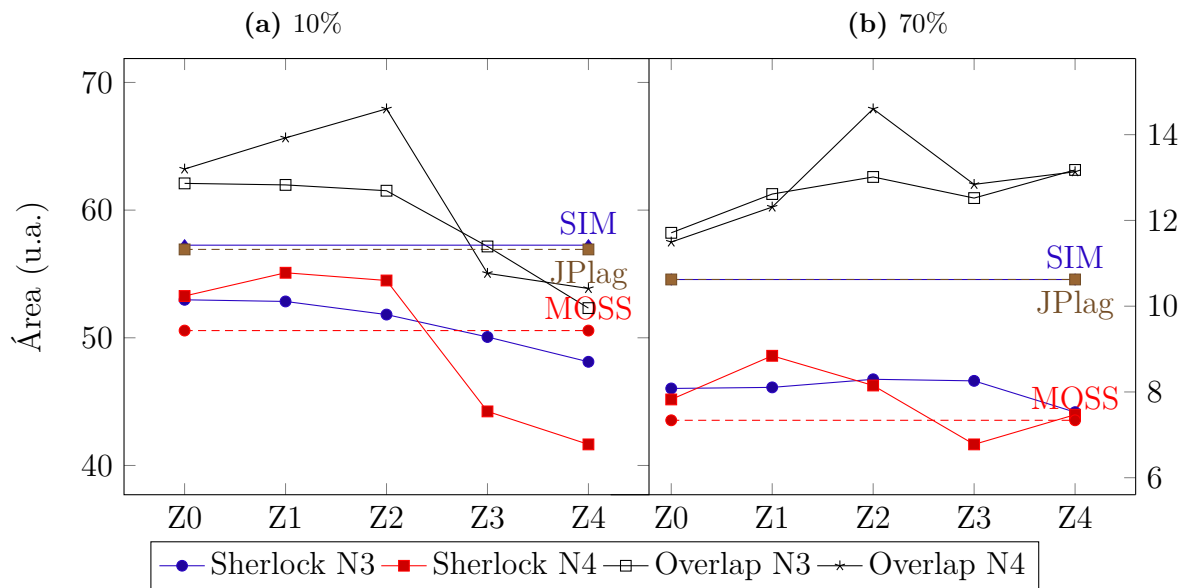
Uma contestação possível para os bons resultados obtidos com a normalização 4 é que, invariavelmente, os códigos resultantes dessa normalização são altamente similares aos olhos humanos, pois comparam um conjunto muito restrito de possibilidades léxicas. Logo, a alta similaridade nessa situação seria consequência do acaso. Com esse pensamento, é esperado que a normalização 4 apresente muitos casos de falsos positivos nesse novo teste com o conjuntos de códigos gerados manualmente.

Figura 5.12 – Áreas para códigos médios



A [Figura 5.12](#) contém os resultados obtidos com a comparação dos 33 códigos médios. Observa-se que as linhas de todos os algoritmos de ambos os gráficos dessa figura apresentam, de modo geral, uma tendência de pouca variação com a mudança de Z. Esse comportamento de tendência à estabilidade é esperado, pois uma quantidade maior de código é considerada, logo, estando menos sujeito a grandes variações ocasionadas por casos isolados.

Já na [Figura 5.13](#), que contém os resultados obtidos com a comparação dos 33 códigos pequenos, a tendência de estabilidade conserva-se apenas até Z igual a 2. A partir dessa configuração, observa-se uma queda acentuada e generalizada para as versões do

Figura 5.13 – Áreas para códigos pequenos

Sherlock na Figura 5.13a, enquanto que na Figura 5.13b a estabilidade prevalece. Diferentemente do que foi constatado anteriormente, por falta de assinaturas suficientes, essa queda não está limitada somente ao Sherlock com a normalização 4, embora nesses casos a queda tenha sido acentuada. Conclui-se, então, que o impacto da perda de desempenho nesses códigos para valores de Z igual a 3 e 4 é principalmente observado ao se considerar toda a faixa de limiares.

Quanto às eventuais ocorrências de resultados falsos, constata-se que a normalização 4, de fato, preserva as características dos códigos analisados e não apresenta quantidade de falsos positivos superior a das demais ferramentas, pois, conforme os gráficos apresentados, o comportamento dos algoritmos permanece constante quando comparados aos gráficos da seção anterior.

5.2 Avaliação conformativa: códigos gerados por alunos em práticas laboratoriais

5.2.1 Cenário de experimentação

Os códigos utilizados nos testes desta seção foram gerados ao longo do ano de 2012, em uma turma de Técnicas de Programação para Engenharia I, disciplina introdutória de

programação do curso de Engenharia de Teleinformática (esta disciplina é anual, assim como todas as demais disciplinas de 1º ano do referido curso). A turma era formada por 97 alunos, dividida em 4 grupos devido à restrição do número de máquinas nos laboratórios. A cada semana, eram propostos 2 problemas para cada grupo de laboratório, somando 8 problemas distintos e com dificuldade equivalente.

No total, foram considerados 22 problemas de cada turma de laboratório. Os códigos foram submetidos pelos alunos por meio da ferramenta integrada ao ambiente Moodle. Todos os códigos enviados sobre um determinado problema foram comparados entre si. Ao todo, foram considerados 6780 pares de códigos, conforme a [Tabela 5.4](#). Esse total é dado pelo somatório da combinação 2 a 2 do total de submissões para cada problema. Como cada par foi duas vezes comparado por 4 ferramentas, ao todo, foram realizadas 54240 comparações.

Tabela 5.4 – Relação quantitativa dos códigos analisados

	Total de Problemas propostos	Total de Exercícios Submetidos	Total de Pares Comparados
Turma A	22	288	2597
Turma B	22	250	1325
Turma C	22	202	902
Turma D	22	304	1956
Total:	88	1044	6780

Conforme exposto anteriormente, existem várias dificuldades para se avaliar ferramentas em um contexto com grande quantidade de pares. Assim, baseado no método de avaliação conformativa, duas análises são realizadas. A primeira baseada nos dados quantitativos obtidos com a contagem do total de ocorrências, ocorrências isoladas e ausências isoladas. Essa análise permite algumas conclusões, além de ajudar na compreensão da importância de calcular essas classes de dados. A segunda análise é baseada na aplicação direta da precisão e da revocação adaptadas, conforme apresentado na [subseção 4.2.2](#).

5.2.2 Análise preliminar: uma primeira visão sobre as quantidades de ocorrências, ocorrências isoladas e ausências isoladas

A [Tabela 5.5](#) apresenta a contabilização dos resultados dos 3 tipos de ocorrências, usando limiares de 60% a 90%, para os algoritmos JPlag, SIM e MOSS sem normalização, comparados com o Sherlock N-Overlap utilizando as normalizações 3 e 4, acumulados para os 22 problemas propostos por turma. Os valores estão distribuídos para cada uma das 4 turmas analisadas. No exemplo, todas as turmas consolidam informações referentes aos 88 problemas de programação distintos propostos ao longo do ano de 2012. Para cada limiar, todos os resultados estão comparados com Sherlock N-Overlap nas normalizações 3 e 4. Isso se deve à alteração dos quantitativos das ocorrências isoladas e das ausências isoladas ao se alternar a normalização do Sherlock N-Overlap.

Total de ocorrências

Na [Tabela 5.5](#), observa-se que o total de ocorrências não é alterado para o JPlag, SIM e MOSS em função da variação na normalização. Isso se deve ao fato de não serem usadas normalização nestes algoritmos. Percebe-se que, ao se alternar da normalização 3 para a 4 com o Sherlock N-Overlap, o total de ocorrências sempre aumenta significativamente para todos os limiares. Por exemplo, na turma A, ao se alternar da normalização 3 para a 4 no Sherlock N-Overlap, ocorre variação de 39 para 148 pares, considerando o limiar de 90%. Esse padrão demonstra a superioridade da normalização 4 em relação a normalização 3 em identificar casos similares. Em termos numéricos, observa-se que as quantidades de ocorrências indicadas pelo Overlap aproxima-se da quantidade indicada pelo JPlag. Isso pode ser verificado analisando-se a diferença do total de ocorrências entre duas ferramentas, considerando a normalização 4. Por exemplo, na turma A, a menor e a maior diferença entre o Overlap e JPlag é 6 e 77, enquanto entre o Overlap e o SIM tem-se 48 e 179, e entre o Overlap e o MOSS 89 e 299.

A quantidade do total de ocorrências para o MOSS merece destaque. Em todos os limiares e em todas as turmas, a quantidade de pares semelhantes indicados pelo MOSS é inferior em relação aos demais algoritmos. Esse resultado do MOSS confirma as conclusões obtidas na análise dos dados manuais.

Tabela 5.5 – Somatório dos pares, classificados dentre os 3 tipos de ocorrências, para os 22 problemas propostos e por turma

Turma A													
Limiar		Total de Ocorrências				Ocorrências Isoladas				Ausências Isoladas			
		Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS
90%	N3	39	154	100	59	4	68	18	1	15	1	0	11
	N4	148	154	100	59	41	57	4	1	1	2	3	28
80%	N3	62	217	133	86	8	115	16	2	29	1	0	10
	N4	259	217	133	86	89	77	5	4	1	7	2	33
70%	N3	90	382	202	126	11	208	22	3	33	4	0	18
	N4	341	382	202	126	105	161	10	8	3	11	10	59
60%	N3	146	558	302	182	18	282	37	1	46	5	1	25
	N4	481	558	302	182	114	207	17	21	5	16	14	105
Turma B													
Limiar		Total de Ocorrências				Ocorrências Isoladas				Ausências Isoladas			
		Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS
90%	N3	9	57	30	21	0	34	6	2	2	1	0	0
	N4	58	57	30	21	23	22	6	2	3	1	0	1
80%	N3	19	84	46	44	4	45	9	5	13	2	0	1
	N4	84	84	46	44	34	38	2	3	5	1	2	0
70%	N3	32	148	68	66	10	90	11	9	20	2	0	0
	N4	119	148	68	66	43	77	4	5	10	5	3	3
60%	N3	76	215	92	94	27	125	11	20	30	5	2	1
	N4	211	215	92	94	101	113	6	18	8	9	3	8
Turma C													
Limiar		Total de Ocorrências				Ocorrências Isoladas				Ausências Isoladas			
		Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS
90%	N3	13	34	24	12	3	21	11	0	3	1	0	0
	N4	41	34	24	12	18	23	1	2	0	3	0	1
80%	N3	30	65	35	22	10	35	9	2	4	1	1	1
	N4	65	65	35	22	22	32	1	1	1	2	0	5
70%	N3	55	124	50	51	15	66	8	8	12	0	2	2
	N4	87	124	50	51	25	67	6	2	2	0	0	2
60%	N3	78	187	79	64	11	98	7	5	18	0	0	6
	N4	137	187	79	64	45	97	6	3	6	1	2	10
Turma D													
Limiar		Total de Ocorrências				Ocorrências Isoladas				Ausências Isoladas			
		Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS	Overlap	JPlag	SIM	MOSS
90%	N3	15	65	60	39	0	24	18	1	15	1	0	1
	N4	104	65	60	39	41	19	2	1	1	1	0	3
80%	N3	45	122	88	53	8	58	16	2	24	1	0	1
	N4	162	122	88	53	67	47	1	2	5	3	0	8
70%	N3	80	184	132	80	15	76	22	3	25	2	0	3
	N4	208	184	132	80	75	69	12	1	3	4	0	25
60%	N3	135	312	200	118	35	148	37	1	33	4	1	6
	N4	295	312	200	118	103	136	22	1	6	9	3	31

Ocorrências isoladas

Em relação às ocorrências isoladas apontadas na [Tabela 5.5](#), novamente, observa-se que ao se alternar da normalização 3 para a 4, com o Sherlock N-Overlap, o total de ocorrências isoladas aumenta significativamente para todos os limiares. As ocorrências isoladas (bem como as ausências isoladas), diferentemente do total de ocorrências, variam de acordo com os resultados das demais ferramentas. Assim, com a alteração dos resultados em virtude da mudança de normalização para o Sherlock N-Overlap, os resultados das outras ferramentas sofreram alteração.

Com os dados da [Tabela 5.5](#), observa-se que o Sherlock N-Overlap é a ferramenta que mais se aproxima, em termos quantitativos, dos resultados do JPlag, enquanto o MOSS é a que mais se distancia.

Ausências isoladas

Na [Tabela 5.5](#), na Turma A, os resultados da normalização 3 em relação a normalização 4 para Sherlock N-Overlap, ao contrário das situações anteriores, apresentam redução das ausências isoladas, pois aumenta as ocorrências coincidentes. Isso é mais uma evidência do quanto a normalização 4 é superior a 3. A redução de ausências isoladas é altamente desejável, já que esse tipo dado é composto apenas por falsos negativos.

Enquanto para o Sherlock N-Overlap, da normalização 3 para a 4, reduz-se a quantidade de ausências isoladas, para os demais algoritmos ocorre aumento. Isso acontece, pois, para determinados pares, ao executar Sherlock N-Overlap com normalização 3, apenas dois (de quatro) algoritmos coincidiam com a indicação de similaridade. Com o aumento das indicações do Sherlock N-Overlap com normalização 4, muitos pares ganharam o 3º algoritmo coincidente, gerando assim mais ausências isoladas para esses algoritmos.

Para as demais turmas, de modo geral, principalmente para limiares altos (80% e 90%), os quantitativos de ausências isoladas para todas as ferramentas são bastante próximos, com exceção do MOSS que, especialmente na Turma A, apresenta quantidade de ausências isoladas bastante superior.

Desse modo, os dados da [Tabela 5.5](#) reafirmam algumas conclusões anteriores. Destaca-se que o Sherlock N-Overlap com normalização 4 apresenta resultados superiores

as obtidos com normalização 3. Nota-se também que, dentre as ferramentas analisadas, Sherlock N-Overlap e JPlag guardam os melhores resultados, enquanto o MOSS os piores.

5.2.3 Análise com aplicação do método de conformidade

Nesta subseção, apresentam-se os dados obtidos com o método definido na [subseção 4.2.2](#). Para obter esses dados, utilizou-se o sistema de Análise de Similaridade, que, após realizar uma comparação com múltiplos algoritmos, gera uma tabela contendo os dados da previsão e da revocação conformativa, bem como a respectiva média harmônica, conforme apresentado na [seção 4.1](#).

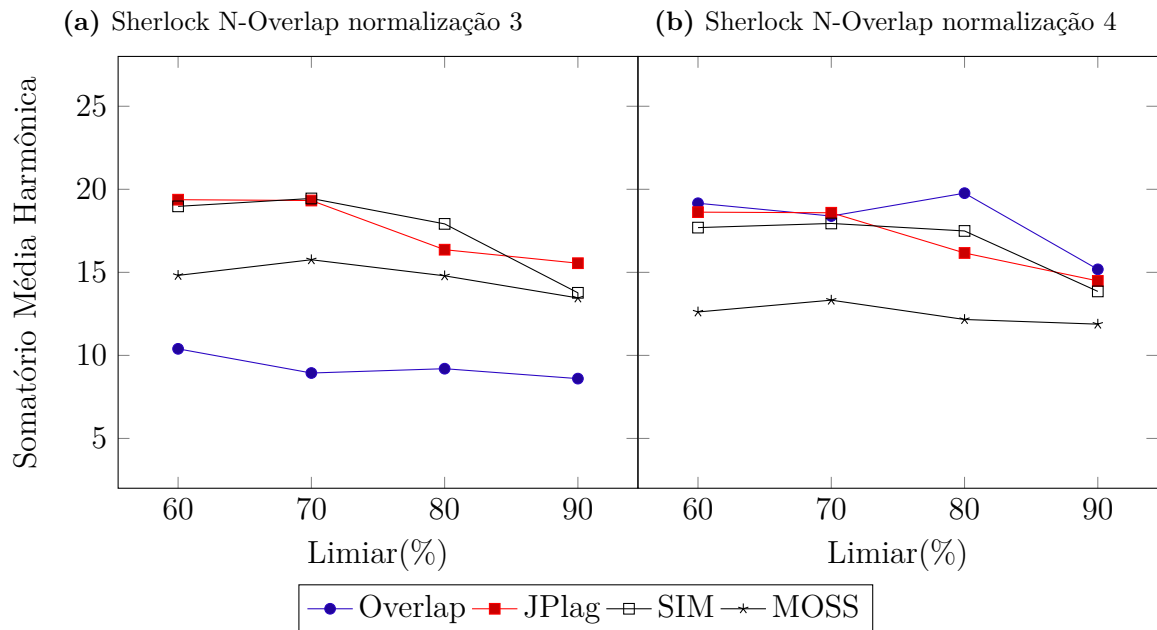
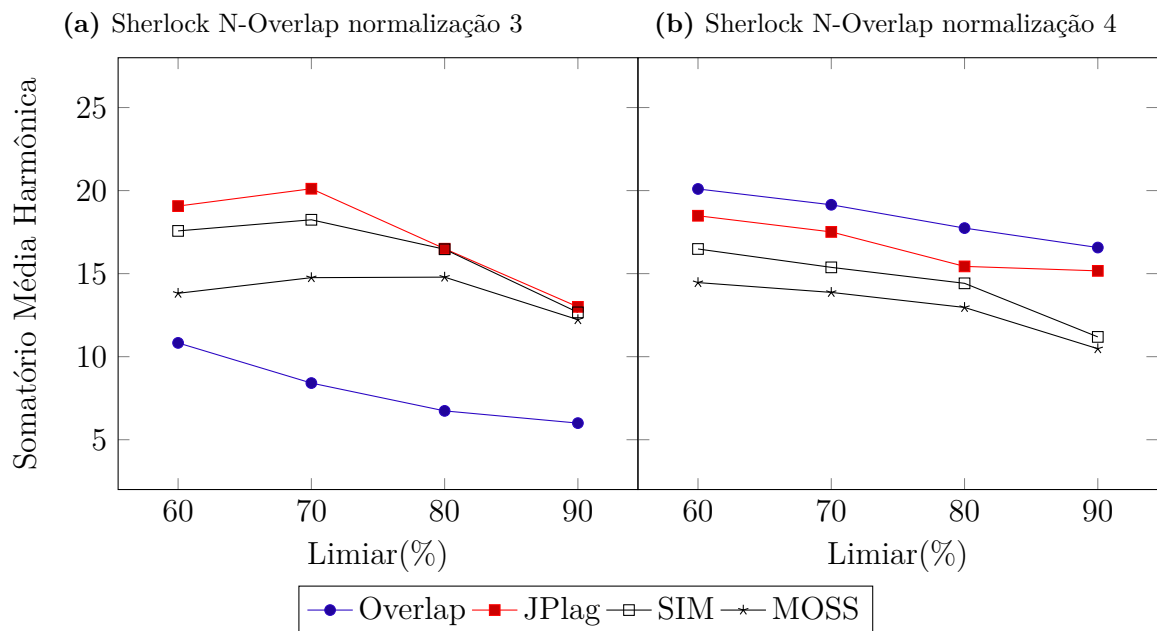
Assim, para cada um dos 88 problemas, somadas as 4 turmas (A, B, C e D), executou-se o JPlag, MOSS e SIM comparando com o Sherlock N-Overlap com normalizações 3 e 4. Os dados foram consolidados em planilhas contendo os quantitativos de ocorrências por limiares e por média harmônica. Mais de 5 mil linhas de dados foram processadas para obtenção desses dados, o que inviabiliza figurarem nos anexos. Entretanto, os dados contudo estão disponíveis na *Web*¹.

Para cada uma das quatro ferramentas e para cada uma das 4 turmas, foram efetuadas 22 execuções da verificação de similaridade, uma para cada problema proposto. Com os resultados de cada verificação de similaridade, foi calculada a média harmônica para quatro faixas de limiares (60%, 70%, 80% e 90%). Para cada limiar e para cada ferramenta, para efeito de comparação, os valores das médias harmônicas foram somados e apresentados nos gráficos das Figuras [5.14](#), [5.15](#), [5.16](#) e [5.17](#).

Na [subseção 5.1.2](#), os valores da média harmônica foram representados pela área sob a região do gráfico para limiares específicos. Essa representação, no presente contexto, não é apropriada, pois a contagem dos parâmetros analisados (principalmente para ocorrências e ausências isoladas) podem variar sem guardar inter-relação entre limiares vizinhos. Assim, decidiu-se somar as médias harmônicas, obtidas para cada limiar, como forma de obter o resultado acumulado com o desempenho geral.

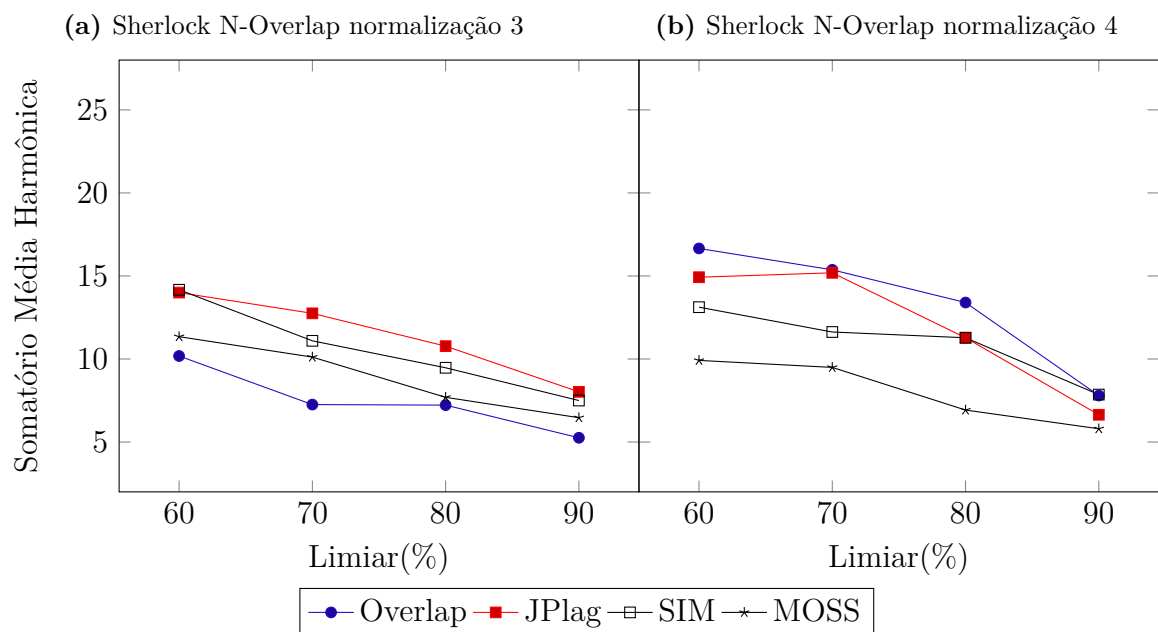
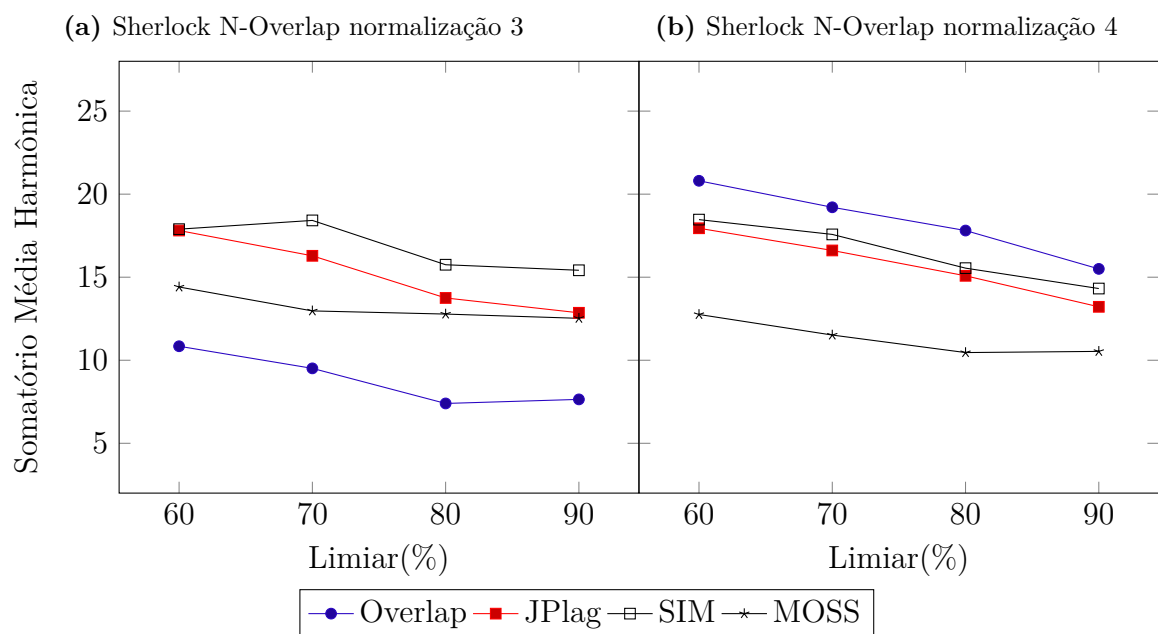
Comparam-se o Sherlock N-Overlap, o JPlag, o SIM e o MOSS em cada uma das figuras ([5.14](#), [5.15](#), [5.16](#) e [5.17](#)), que são divididas em dois gráficos: o da esquerda com o Sherlock N-Overlap com a normalização 3 e o da direita com o mesmo algoritmo usando

¹ <https://www.dropbox.com/s/ycuh8an9mg0iflz/Codigos%20Reais%20-%20Dados.zip>

Figura 5.14 – Turma A - Códigos Reais**Figura 5.15** – Turma B - Códigos Reais

a normalização 4. É interessante relembrar que a diferença de normalização somente é aplicada ao Sherlock N-Overlap, contudo o desempenho dos demais algoritmos também pode ser alterado, pois, com o método de conformidade, as indicações de similaridade dada por uma ferramenta é utilizado para avaliar o desempenho das demais em função da contabilização de ocorrências e ausências isoladas.

É possível observar várias características comuns às Figuras 5.14, 5.15, 5.16 e

Figura 5.16 – Turma C - Códigos Reais**Figura 5.17** – Turma D - Códigos Reais

5.17. Em todas elas o Sherlock N-Overlap com normalização 4 apresenta ganho bastante significativo em relação à execução com a normalização 3, ficando, para a maioria dos limiares, superior a todas as outras ferramentas. Nos gráficos com a normalização 4, o MOSS apresenta notória distância das demais ferramentas e sempre com o pior desempenho. Essas conclusões reafirmam os resultados obtidos ao empregar o método tradicional de cálculo.

Também é possível observar que a variação de limiar não altera de forma significativa o desempenho comparativo em relação aos algoritmos analisados.

Conclusão

Este trabalho se contextualiza na análise de similaridade entre códigos-fonte gerados em práticas laboratoriais de turmas de programação em cursos universitários. Objetiva-se instrumentalizar o professor para o acompanhamento e a avaliação de turmas numerosas, situação frequente em turmas de primeiro ano em cursos de Engenharia e de Ciência da Computação, devido não só ao ingresso de novos alunos, mas também à grande incidência de alunos repetentes.

Discutiu-se a importância em diferenciar plágio de similaridade, e as razões que justificam a ocorrência da similaridade no contexto de práticas de programação. Entretanto, por se tratar de uma investigação que se concentra especificamente na concepção de uma ferramenta capaz de identificar, de maneira eficiente, o grau de similaridade entre códigos-fonte, elencaram-se as principais modificações empregadas pelos alunos com a intenção de plagiar, o que serviu de base para análises de resultados em códigos controlados. As principais técnicas e algoritmos encontrados na literatura sobre detecção de plágio em código-fonte foram apresentadas e discutidas.

De maneira a preencher uma das diversas lacunas ainda existente no domínio da detecção de plágio e da análise de similaridade, foram estudadas e elaboradas técnicas de normalização que preparam os códigos antes da comparação, destacando suas propriedades relevantes e eliminando características irrelevantes.

O algoritmo de comparação Sherlock foi alvo de estudo aprofundado, devido as suas características intrínsecas e ao fato de ser um algoritmo de código aberto, disponível para modificações. De maneira especial, descobriu-se que a utilização de regras de

pré-processamento produz melhora significativa dos resultados do Sherlock. Além disso, a substituição da métrica de Jaccard pelo coeficiente de sobreposição (Overlap) alavancou os resultados do Sherlock, complementando a solução denominada Sherlock N-Overlap, principal contribuição deste trabalho.

Para avaliar essas duas propostas de modificação, foi necessário comparar os seus resultados com os das ferramentas mais bem recomendadas para esta finalidade e que, de alguma maneira, estão disponíveis, notoriamente o JPlag, o MOSS e o SIM. Dada a quantidade de códigos reunido para análise, seria muito dispendioso avaliar em separado o resultado de cada ferramenta com suas formatações específicas. Por isso, foi desenvolvido um ambiente computacional de análise que integra todas as ferramentas e técnicas de processamento e permite a inclusão de novas abordagens.

Mesmo tendo à disposição os resultados de todas as ferramentas de maneira simultânea em uma mesma interface, uma análise mais consistente de dados requer uma abordagem sistematizada. O método de avaliação, baseado no cálculo da precisão e da revocação, amplamente utilizado na literatura, exige o conhecimento *a priori* e a contabilização de falsos positivos e falsos negativos para avaliar a qualidade do resultado obtido. Para um conjunto de códigos que não foi construído de maneira controlada, a determinação do status plágio/não-plágio é feita com base na experiência de um ser humano. Muitas vezes, essa determinação é subjetiva e não garante reprodutibilidade, além de ser uma operação praticamente inviável para grandes quantidades de códigos. Neste sentido, de maneira a viabilizar a análise da qualidade do Sherlock N-Overlap, foi proposto neste trabalho um método baseado na contagem dos resultados de ferramentas de referência, ou método de conformidade, que pode ser generalizado para o estudo de novos algoritmos de detecção de plágio e análise de similaridade.

Assim, com os dois métodos disponíveis, foram avaliados, num primeiro momento, códigos produzidos e plagiados propositalmente e, num segundo momento, códigos produzido por alunos em uma disciplina anual de programação.

Os resultados mostram que tanto as técnicas de normalização quanto a substituição da métrica originalmente utilizada no Sherlock pelo coeficiente de sobreposição produz melhores resultados no domínio de práticas laboratoriais de programação.

Das quatro técnicas de normalização desenvolvidas, numeradas de 1 a 4, as

normalizações 3 e 4 foram aquelas que mostraram valor especial. A normalização 4, técnica bastante invasiva, em conjunto com o coeficiente de sobreposição no algoritmo Sherlock, apresenta resultados bastante satisfatórios e, na maioria das situações, superior a todas as ferramentas de referência que foram utilizadas, o que representa um avanço no estado da arte.

O sistema de análise de similaridade mostrou-se útil tanto para utilização por professores em sala de aula, como também para a avaliação da qualidade de novas ferramentas e algoritmos. O método de análise proposto, apesar de ser dependente de outros algoritmos, representa um importante recurso para a identificação de similaridade em grande quantidade de códigos-fonte, viabilizando a comparação que seria de difícil execução por agentes humanos.

Alguns dos resultados apresentados foram publicados em revistas de alcance nacional, elencados na próxima seção e presentes no Apêndice [D](#).

6.1 Publicações/Resultados

MACIEL, Danilo Leal; SOARES, J. M; FRANÇA, Allyson Bonetti; GOMES, D. G. Análise de Similaridade de Códigos-Fonte como Estratégia para o Acompanhamento de Atividades de Laboratório de Programação. Revista Novas Tecnologias na Educação (RENOTE), v. 10, p. 1-10, 2012.

FRANÇA, Allyson Bonetti; MACIEL, Danilo Leal; SOARES, José Marques. Sistema de apoio a atividades de laboratório de programação via Moodle com suporte ao balanceamento de carga e análise de similaridade de código. Revista Brasileira de Informática na Educação (RBIE), v. 21, p. 91-105, 2013.

Software: Sistema de Análise de similaridade com integração para JPlag, MOSS, SIM, Sherlock, Sherlock N-Overlap e normalizações.

6.2 Perspectivas futuras

Tendo em vista o escopo aberto em que este trabalho se insere, muitas ideias desenvolvidas ao longo do processo não puderam ser amadurecidas. Assim, tem-se uma ampla perspectiva de trabalhos futuros, sendo alguns deles enumerados a seguir:

- ▶ Avaliar as técnicas de normalização desenvolvidas e aprimorá-las, em especial, no que concerne a possíveis ambiguidades léxicas, como estruturas de seleção e de repetição baseada em instruções diferentes;
- ▶ Avaliar a dispersão das assinaturas e o critério de seleção por zerobit, permitindo a escolha automatizada com base em características que possam ser identificadas nos códigos a serem comparados;
- ▶ Modificar o Sherlock para armazenar as posições das estruturas comparadas a fim de permitir a visualização dos trechos léxicos em comum;
- ▶ Avaliar desempenho do Sherlock com tokenização e outras métricas além do Overlap e do Dice, já estudadas;
- ▶ Atribuir peso ao valor de similaridade com base na quantidade de assinaturas consideradas;
- ▶ Estudar formas de melhorar o método de avaliação proposto, implementando o processamento e a geração dos gráficos na própria ferramenta;
- ▶ Definir uma versão do Sistema de Análise de Similaridade com configurações otimizadas para o uso do professor em sala de aula/lab. de programação.

Referências

- APIRATIKUL, P. *Document Fingerprinting Using Graph Grammar Induction*. Dissertação (Mestrado) — Oklahoma State University, 2004.
- BIN-HABTOOR, A. S.; ZAHER, M. A. A survey on plagiarism detection systems. *International Journal of Computer Theory and Engineering*, v. 4, n. 2, p. 185–188, 2012.
- BOWYER, K.; HALL, L. Experience using “MOSS” to detect cheating on programming assignments. In: *Frontiers in Education Conference, 1999. FIE '99. 29th Annual*. [S.l.: s.n.], 1999. v. 3, p. 13B3/18–13B3/22 vol.3. ISSN 0190-5848.
- BURROWS, S.; TAHAGHOGHI, S. M. M.; ZOBEL, J. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, John Wiley & Sons, Ltd., v. 37, n. 2, p. 151–175, 2007. ISSN 1097-024X. Disponível em: <http://dx.doi.org/10.1002/spe.750>.
- CAMARGOS, N. dos S. *Análise de Similaridade entre Códigos-Fonte em Disciplinas de Programação: Adaptações do Algoritmo Sherlock para Uso dos Coeficientes de Sorensen-Dice e de Sobreposição*. Monografia (Trabalho de Conclusão de Curso) — Universidade Federal do Ceará, 2013.
- CEDENÑO, L. A. B. *On the Mono- and Cross-Language Detection of Text Re-Use and Plagiarism*. Tese (Doutorado) — Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, June 2012.
- CEGLAREK, D. Evaluation of the SHAPD2 algorithm efficiency in plagiarism detection tasks. In: *Technological Advances in Electrical, Electronics and Computer Engineering (TAECE), 2013 International Conference on*. [S.l.: s.n.], 2013. p. 465–470.
- CESARE, S.; XIANG, Y. *Software Similarity and Classification*. [S.l.]: Springer, 2012. I-XIV, 1-88 p. (Springer Briefs in Computer Science). ISBN 978-1-4471-2908-0.
- CHARIKAR, M. S. Similarity estimation techniques from rounding algorithms. In: *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 2002. (STOC '02), p. 380–388. ISBN 1-58113-495-9. Disponível em: <http://doi.acm.org/10.1145/509907.509965>.
- CLOUGH, P.; STEVENSON, M. Developing a corpus of plagiarised short answers. *Language Resources and Evaluation*, Springer Netherlands, v. 45, n. 1, p. 5–24, 2011. ISSN 1574-020X. Disponível em: <http://dx.doi.org/10.1007/s10579-009-9112-1>.

COSMA, G.; JOY, M. *Source-code Plagiarism: a UK Academic Perspective*. Technical Report. Coventry, 2006. Disponível em: <<http://eprints.dcs.warwick.ac.uk/52/>>.

COSMA, G.; JOY, M. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, DC, USA, v. 61, n. 3, p. 379–394, mar. 2012. ISSN 0018-9340. Disponível em: <<http://dx.doi.org/10.1109/TC.2011.223>>.

DICE, L. R. Measures of the amount of ecologic association between species. *Ecology*, v. 26, n. 3, p. 297–302, jul 1945.

DOUGIAMAS, M.; TAYLOR, P. C. Moodle: Using learning communities to create an open source course management system. In: *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications*. [S.l.: s.n.], 2003.

DURIC, Z.; GASEVIC, D. A source code similarity system for plagiarism detection. *The Computer Journal*, Oxford University Press, Oxford, UK, v. 56, n. 1, p. 70–86, 2012. ISSN 0010-4620. Disponível em: <<http://dx.doi.org/10.1093/comjnl/bxs018>>.

FAIDHI, J. A. W.; ROBINSON, S. K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, Elsevier Science Ltd., Oxford, UK, UK, v. 11, n. 1, p. 11–19, jan. 1987. ISSN 0360-1315. Disponível em: <[http://dx.doi.org/10.1016/0360-1315\(87\)90042-X](http://dx.doi.org/10.1016/0360-1315(87)90042-X)>.

FRANÇA, A. B.; SOARES, J. M. Sistema de apoio a atividades de laboratório de programação via moodle com suporte ao balanceamento de carga. In: *XXII Simpósio Brasileiro de Informática na Educação*. Anais do XXII SBIE, 2011. p. 710–719. ISBN 2176-4301. Disponível em: <http://www.br-ie.org/sbie-wie2011/SBIE-Trilha5/92971_1.pdf>.

GITCHELL, D.; TRAN, N. Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 31, n. 1, p. 266–270, mar. 1999. ISSN 0097-8418. Disponível em: <<http://doi.acm.org/10.1145/384266.299783>>.

GREEN, P.; LANE, P. C. R.; RAINER, A.; SCHOLZ, S.-B.; BENNETT, S. Same difference: Detecting collusion by finding unusual shared elements. In: *Proceedings of the 5th International Plagiarism Conference*. Newcastle-upon-Tyne, UK: [s.n.], 2012. [Http://archive.plagiarismadvice.org/conference-programme](http://archive.plagiarismadvice.org/conference-programme).

GRUNE, D. *Concise Report on the Algorithms in SIM*. 2013. Disponível em: <ftp://ftp.cs.vu.nl/pub/dick/similarity_tester/TechnReport>.

GRUNE, D. *The software and text similarity tester SIM*. 2013. Disponível em: <http://dickgrune.com/Programs/similarity_tester/>.

GRUNE, D.; VAKGROEP, M. *Detecting copied submissions in computer science workshops*. Technical Report. [S.l.], 1989.

HAGE, J.; RADEMAKER, P.; VUGT, N. van. *A comparison of plagiarism detection tools*. Technical Report. [S.l.], 2010. Disponível em: <<http://www.cs.uu.nl/research/techreps/repo/CS-2010/2010-015.pdf>>.

HECKEL, P. A technique for isolating differences between files. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 4, p. 264–268, abr. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359460.359467>.

JACCARD, P. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, v. 37, p. 547–579, 1901.

JOY, M.; LUCK, M. Plagiarism in programming assignments. *IEEE Transactions on Education*, Institute of Electrical and Electronics Engineers, v. 42, n. 2, p. 129–133, 1999. Disponível em: <http://eprints.dcs.warwick.ac.uk/81/>.

KARP, R. M.; RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 31, n. 2, p. 249–260, mar. 1987. ISSN 0018-8646. Disponível em: <http://dx.doi.org/10.1147/rd.312.0249>.

KLEIMAN, A. B. *Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação*. Dissertação (Mestrado) — UNICAMP, 2007.

KROKOSCH, M. Abordagem do plágio nas três melhores universidades de cada um dos cinco continentes e do brasil. *Revista Brasileira de Educação*, v. 16, n. 48, 2011. Disponível em: <http://www.scielo.br/pdf/rbedu/v16n48/v16n48a11>.

MACIEL, D. L.; SOARES, J. M.; BONETTI, A.; GOMES, D. Análise de similaridade de códigos-fonte como estratégia para o acompanhamento de atividades de laboratório de programação. *Revista Novas Tecnologias na Educação (RENTE)*, v. 10, p. 1–10, 2012. ISSN 1679-1916.

MATSUO, Y.; MORI, J.; HAMASAKI, M.; NISHIMURA, T.; TAKEDA, H.; HASIDA, K.; ISHIZUKA, M. Polyphonet: An advanced social network extraction system from the web. *Web Semantics: Science, Services and Agents on the World Wide Web*, v. 5, n. 4, p. 262–278, 2007. ISSN 1570-8268. World Wide Web Conference 2006 Semantic Web Track. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1570826807000340>.

MOZGOVOY, M. Desktop tools for offline plagiarism detection in computer programs. *Informatics in Education*, v. 5, n. 1, p. 97–112, 2006.

MOZGOVOY, M.; KARAKOVSKIY, S.; KLYUEV, V. Fast and reliable plagiarism detection system. In: *Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE '07. 37th Annual*. [S.l.: s.n.], 2007. p. S4H–11–S4H–14. ISSN 0190-5848.

OHMANN, A. *Efficient Clustering-based Plagiarism Detection using IPPDC*. Dissertação (Mestrado) — Saint John's University, April 2013.

ONLINEJUDGE. 2013. Disponível em: <https://github.com/hit-moodle/onlinejudge>.

PCPP. *Programming Code Plagiarism Plugin*. 2012. Disponível em: http://docs.moodle.org/22/en/Programming_Code_Plagiarism_Plugin.

PIKE, R.; LOKI. *The Sherlock Plagiarism Detector*. 2013. Disponível em: <http://sydney.edu.au/engineering/it/~scilect/sherlock/>.

- POON, J. Y.; SUGIYAMA, K.; TAN, Y. F.; KAN, M.-Y. Instructor-centric source code plagiarism detection and plagiarism corpus. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. New York, NY, USA: ACM, 2012. (ITiCSE '12), p. 122–127. ISBN 978-1-4503-1246-2. Disponível em: <<http://doi.acm.org/10.1145/2325296.2325328>>.
- POTTHAST, M.; STEIN, B.; BARRÓN-CEDENO, A.; ROSSO, P. An Evaluation Framework for Plagiarism Detection. In: *Proceedings of the 23rd International Conference on Computational Linguistics (COLING 2010)*. Beijing, China: Association for Computational Linguistics, 2010.
- PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, v. 8, n. 11, p. 1016–1038, nov 2002.
- SALES, G. L.; BARROSO, G. C.; SOARES, J. M. Learning vectors (lv): Um modelo de avaliação processual com mensuração não-linear da aprendizagem em ead online. *Revista Brasileira de Informática na Educação (RBIE)*, v. 20, n. 1, p. 60–74, 2012. ISSN 1414-5685.
- SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2003. (SIGMOD '03), p. 76–85. ISBN 1-58113-634-X. Disponível em: <<http://doi.acm.org/10.1145/872757.872770>>.
- SILVA, C. H. de C.; SAMPAIO, R. F.; LEÃO, R. P. S.; BARROSO, G. C.; SOARES, J. M. Desenvolvimento de um laboratório virtual para capacitação tecnológica à distância em proteção de sistemas elétricos. *Revista Novas Tecnologias na Educação (RENOTE)*, v. 9, n. 1, 2011. ISSN 1679-1916.
- SORENSEN, T. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *Biologiske Skrifter*, p. 1–34, 1948.
- STAMATATOS, E. Plagiarism detection based on structural information. In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. ACM, 2011. (CIKM '11), p. 1221–1230. ISBN 978-1-4503-0717-8. Disponível em: <<http://doi.acm.org/10.1145/2063576.2063754>>.
- TAVARES, D. A. B.; FRANÇA, A. B.; SOARES, J. M.; BARROSO, N. M. C.; MOTA, J. C. M. Integração do ambiente wims ao moodle usando arquitetura orientada a serviços e compilação automática de médias. *Revista Novas Tecnologias na Educação (RENOTE)*, v. 8, n. 3, 2010. ISSN 1679-1916.
- TVERSKY, A. Features of similarity. *Psychological Review*, v. 84, n. 4, p. 327–351, July 1977.
- VPL. 2013. Disponível em: <<http://vpl.dis.ulpgc.es/>>.
- WHALE, G. Identification of program similarity in large populations. *The Computer Journal*, p. 140–146, 1990.

WISE, M. J. *String similarity via greedy string tiling and running Karp-Rabin matching*. Basser Department of Computer Science, University of Sydney, 1993. Disponível em: <http://www.pam1.bcs.uwa.edu.au/~michaelw/ftp/doc/RKR_GST.ps>.

WISE, M. J. Yap3: Improved detection of similarities in computer program and other texts. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 28, n. 1, p. 130–134, mar. 1996. ISSN 0097-8418. Disponível em: <<http://doi.acm.org/10.1145/236462.236525>>.

WOLDA, H. Similarity indices, sample size and diversity. *Oecologia*, v. 50, n. 3, p. 296–302, 1981. Disponível em: <<http://dx.doi.org/10.1007/bf00344966>>.

ZAKOVA, K.; PISTEJ, J.; BISTAK, P. Online tool for student's source code plagiarism detection. In: *Emerging eLearning Technologies and Applications (ICETA), 2013 IEEE 11th International Conference on*. [S.l.: s.n.], 2013. p. 415–419.

ZOBEL, J.; MOFFAT, A. Exploring the similarity space. *SIGIR FORUM*, v. 32, p. 18–34, 1998.

Enunciados dos problemas plagiados propositalmente

Conforme os critérios explicitados na [subseção 5.1.1](#), solicitou-se a 3 programadores que sugerissem e desenvolvessem códigos plagiados e não plagiados. A seguir, os enunciados desses problemas.

Enunciados dos códigos sugeridos e plagiados pelo Programador

1. Implementação do código não plágio pelo Programador 2

Código pequeno: Crie um jogo simples de adivinhe um número. O número deve estar entre 1 e 10 e o programa deve dar dicas sobre cada tentativa que o usuário fizer, por exemplo: o número tentado é menor, maior... até que o usuário ganhe.

Código grande: Desenvolva um programa que executa as seguintes operações sobre uma estrutura que representa pontos 2D: redefina os pontos, some dois pontos, subtraia dois pontos, calcule a distância entre dois pontos. Seu programa também deve conter um menu que exponha as funcionalidades citadas.

Enunciados dos códigos sugeridos e plagiados pelo Programador

2. Implementação do código não plágio pelo Programador 3

Código pequeno: Criar um programa que leia dez números inteiros e imprima o maior e o segundo maior número da lista.

Código grande: Construa um programa para calcular a média de valores PARES e ÍMPARES, de 10 números que serão digitados pelo usuário. Ao final o programa deve mostrar estas duas médias. O programa deve mostrar também o maior número PAR digitado e o menor número ÍMPAR digitado. Esses dados devem ser armazenados em um vetor. Além disso, devem ser impressos os valores PARES maiores que a média PAR, bem como os valores ÍMPARES menor que a média ÍMPAR.

Enunciados dos códigos sugeridos e plagiados pelo Programador

3. Implementação do código não plágio pelo Programador 1

Código pequeno: Crie um programa que leia uma string via console e imprima na saída essa mesma string toda em caixa alta.

Código grande: Elabore um programa que exiba e atualize um relógio de software que marque horas, minutos e segundos. Nesse programa, o delay para mudança de estados deve ser definido a priori, e não necessita manter uma relação direta com o tempo real (relógio de software!).

Apêndice B

Tabelas: dados brutos

Figura B.1 – Código médio - Programador 1 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	0%	94%	34%	97%	97%	95%	100.0%
mod3.c	88%	78%	94%	89%	94%	83%	70.6%
mod4.c	95%	89%	96%	92%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	90%	100%	98%	100%	99%	98%	99.1%
mod7.c	72%	65%	80%	74%	71%	67%	78.3%
mod8.c	0%	27%	22%	39%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	27%	39%	0%	21.4%

Figura B.2 – Código médio - Programador 1 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	0%	92%	37%	97%	97%	95%	100.0%
mod3.c	84%	71%	94%	90%	94%	83%	70.6%
mod4.c	95%	92%	96%	97%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	85%	100%	97%	100%	99%	98%	99.1%
mod7.c	70%	80%	81%	97%	71%	67%	78.3%
mod8.c	0%	30%	0%	54%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	24%	39%	0%	21.4%

Figura B.3 – Código médio - Programador 1 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	0%	95%	38%	100%	97%	95%	100.0%
mod3.c	81%	86%	95%	100%	94%	83%	70.6%
mod4.c	95%	91%	98%	100%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	80%	100%	100%	100%	99%	98%	99.1%
mod7.c	69%	68%	79%	94%	71%	67%	78.3%
mod8.c	0%	23%	22%	47%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	28%	39%	0%	21.4%

Figura B.4 – Código médio - Programador 1 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	24%	100%	55%	100%	97%	95%	100.0%
mod3.c	84%	84%	100%	100%	94%	83%	70.6%
mod4.c	96%	92%	100%	100%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	71%	100%	100%	100%	99%	98%	99.1%
mod7.c	61%	63%	76%	90%	71%	67%	78.3%
mod8.c	0%	0%	0%	36%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	0%	39%	0%	21.4%

Figura B.5 – Código médio -
Programador 1 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	98%	100.0%
mod1.c	100%	100%	100%	100%	100%	98%	100.0%
mod2.c	31%	100%	50%	100%	97%	95%	100.0%
mod3.c	86%	100%	100%	100%	94%	83%	70.6%
mod4.c	100%	100%	100%	100%	98%	87%	84.7%
mod5.c	100%	100%	100%	100%	100%	98%	100.0%
mod6.c	60%	100%	100%	100%	99%	98%	99.1%
mod7.c	57%	36%	76%	80%	71%	67%	78.3%
mod8.c	0%	0%	0%	20%	16%	8%	14.4%
nao-plagio.c	0%	0%	0%	20%	39%	0%	21.4%

Figura B.6 – Código médio -
Programador 2 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	22%	98%	34%	100%	100%	88%	100.0%
mod3.c	83%	79%	88%	85%	88%	82%	82.6%
mod4.c	94%	91%	96%	93%	97%	94%	84.3%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	84%	92%	97%	98%	100%	96%	93.5%
mod7.c	66%	44%	81%	69%	56%	35%	100.0%
mod8.c	0%	48%	28%	68%	59%	32%	76.7%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.7 – Código médio -
Programador 2 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	23%	97%	38%	100%	100%	88%	100.0%
mod3.c	86%	79%	93%	89%	88%	82%	82.6%
mod4.c	95%	91%	97%	93%	97%	94%	84.3%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	83%	93%	97%	100%	100%	96%	93.5%
mod7.c	68%	54%	81%	80%	56%	35%	100.0%
mod8.c	0%	48%	30%	67%	59%	32%	76.7%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.8 – Código médio -
Programador 2 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	23%	95%	35%	100%	100%	88%	100.0%
mod3.c	84%	86%	91%	92%	88%	82%	82.6%
mod4.c	97%	87%	100%	93%	97%	94%	84.3%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	85%	95%	97%	100%	100%	96%	93.5%
mod7.c	71%	51%	89%	80%	56%	35%	100.0%
mod8.c	21%	58%	32%	80%	59%	32%	76.7%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.9 – Código médio -
Programador 2 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	23%	100%	35%	100%	100%	88%	100.0%
mod3.c	82%	80%	94%	87%	88%	82%	82.6%
mod4.c	96%	86%	100%	88%	97%	94%	84.3%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	84%	100%	100%	100%	100%	96%	93.5%
mod7.c	83%	42%	90%	66%	56%	35%	100.0%
mod8.c	27%	66%	40%	77%	59%	32%	76.7%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.10 – Código médio -
Programador 2 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	29%	100%	35%	100%	100%	88%	100.0%
mod3.c	76%	87%	91%	100%	88%	82%	82.6%
mod4.c	100%	77%	100%	80%	97%	94%	84.3%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	83%	100%	100%	100%	100%	96%	93.5%
mod7.c	90%	40%	93%	60%	56%	35%	100.0%
mod8.c	35%	75%	42%	100%	59%	32%	76.7%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.11 – Código médio -
Programador 3 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	42%	100%	53%	100%	100%	97%	100.0%
mod3.c	59%	56%	70%	74%	51%	36%	71.6%
mod4.c	86%	78%	91%	86%	74%	66%	75.4%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	34%	23%	52%	50%	28%	0%	52.4%
mod7.c	57%	39%	76%	64%	64%	52%	100.0%
mod8.c	21%	0%	36%	27%	0%	0%	35.2%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.12 – Código médio -
Programador 3 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	41%	100%	54%	100%	100%	97%	100.0%
mod3.c	58%	54%	71%	75%	51%	36%	71.6%
mod4.c	83%	79%	92%	89%	74%	66%	75.4%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	33%	23%	50%	53%	28%	0%	52.4%
mod7.c	57%	47%	81%	75%	64%	52%	100.0%
mod8.c	0%	0%	34%	28%	0%	0%	35.2%
nao-plagio.c	0%	0%	0%	0%	0%	0%	0%

Figura B.13 – Código médio -
Programador 3 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	45%	100%	64%	100%	100%	97%	100.0%
mod3.c	63%	52%	76%	75%	51%	36%	71.6%
mod4.c	89%	81%	96%	91%	74%	66%	75.4%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	35%	27%	53%	66%	28%	0%	52.4%
mod7.c	53%	39%	81%	66%	64%	52%	100.0%
mod8.c	0%	0%	34%	33%	0%	0%	35.2%
nao-plagio.c	0%	0%	0%	25%	0%	0%	0%

Figura B.14 – Código médio -
Programador 3 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	50%	100%	66%	100%	100%	97%	100.0%
mod3.c	64%	40%	80%	66%	51%	36%	71.6%
mod4.c	66%	85%	87%	100%	74%	66%	75.4%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	36%	0%	60%	66%	28%	0%	52.4%
mod7.c	73%	50%	100%	83%	64%	52%	100.0%
mod8.c	0%	0%	50%	0%	0%	0%	35.2%
nao-plagio.c	0%	21%	0%	50%	0%	0%	0%

Figura B.15 – Código médio -
Programador 3 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	97%	100.0%
mod1.c	100%	100%	100%	100%	100%	97%	100.0%
mod2.c	62%	100%	80%	100%	100%	97%	100.0%
mod3.c	44%	57%	60%	80%	51%	36%	71.6%
mod4.c	66%	100%	100%	100%	74%	66%	75.4%
mod5.c	100%	100%	100%	100%	100%	97%	100.0%
mod6.c	55%	27%	80%	60%	28%	0%	52.4%
mod7.c	85%	44%	100%	80%	64%	52%	100.0%
mod8.c	36%	0%	60%	33%	0%	0%	35.2%
nao-plagio.c	0%	0%	0%	40%	0%	0%	0%

Figura B.16 – Código pequeno -
Programador 1 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	93%	100.0%
mod1.c	100%	100%	100%	100%	100%	93%	100.0%
mod2.c	28%	100%	46%	100%	100%	58%	100.0%
mod3.c	85%	73%	91%	85%	89%	76%	82.7%
mod4.c	56%	33%	72%	50%	43%	0%	0%
mod5.c	73%	51%	87%	75%	56%	93%	50.0%
mod6.c	78%	75%	95%	90%	100%	0%	0%
mod7.c	78%	37%	89%	55%	51%	74%	100.0%
mod8.c	0%	0%	0%	30%	0%	0%	0%
nao-plagio.c	0%	20%	27%	35%	0%	0%	0%

Figura B.17 – Código pequeno -
Programador 1 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	93%	100.0%
mod1.c	100%	100%	100%	100%	100%	93%	100.0%
mod2.c	21%	100%	38%	100%	100%	58%	100.0%
mod3.c	81%	75%	92%	100%	89%	76%	82.7%
mod4.c	63%	29%	78%	50%	43%	0%	0%
mod5.c	87%	50%	92%	83%	56%	93%	50.0%
mod6.c	82%	78%	100%	91%	100%	0%	0%
mod7.c	75%	50%	85%	66%	51%	74%	100.0%
mod8.c	0%	0%	0%	33%	0%	0%	0%
nao-plagio.c	0%	28%	30%	50%	0%	0%	0%

Figura B.18 – Código pequeno -
Programador 1 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	93%	100.0%
mod1.c	100%	100%	100%	100%	100%	93%	100.0%
mod2.c	20%	100%	36%	100%	100%	58%	100.0%
mod3.c	92%	66%	100%	100%	89%	76%	82.7%
mod4.c	64%	0%	81%	33%	43%	0%	0%
mod5.c	84%	50%	91%	100%	56%	93%	50.0%
mod6.c	75%	75%	100%	100%	100%	0%	0%
mod7.c	75%	40%	100%	66%	51%	74%	100.0%
mod8.c	0%	0%	0%	33%	0%	0%	0%
nao-plagio.c	0%	0%	20%	33%	0%	0%	0%

Figura B.19 – Código pequeno -
Programador 1 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	-1%	100%	-1%	100%	93%	100.0%
mod1.c	100%	-1%	100%	-1%	100%	93%	100.0%
mod2.c	22%	-1%	40%	-1%	100%	58%	100.0%
mod3.c	83%	-1%	100%	-1%	89%	76%	82.7%
mod4.c	57%	0%	80%	0%	43%	0%	0%
mod5.c	100%	0%	100%	0%	56%	93%	50.0%
mod6.c	71%	0%	100%	0%	100%	0%	0%
mod7.c	80%	0%	100%	0%	51%	74%	100.0%
mod8.c	0%	0%	0%	0%	0%	0%	0%
nao-plagio.c	0%	0%	25%	0%	0%	0%	0%

Figura B.20 – Código pequeno -
Programador 1 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	-1%	100%	-1%	100%	93%	100.0%
mod1.c	100%	-1%	100%	-1%	100%	93%	100.0%
mod2.c	0%	-1%	33%	-1%	100%	58%	100.0%
mod3.c	75%	-1%	100%	-1%	89%	76%	82.7%
mod4.c	50%	-1%	66%	-1%	43%	0%	0%
mod5.c	100%	0%	100%	0%	56%	93%	50.0%
mod6.c	75%	0%	100%	0%	100%	0%	0%
mod7.c	66%	-1%	100%	-1%	51%	74%	100.0%
mod8.c	0%	0%	0%	0%	0%	0%	0%
nao-plagio.c	20%	0%	33%	0%	0%	0%	0%

Figura B.21 – Código pequeno -
Programador 2 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	92%	100.0%
mod1.c	100%	100%	100%	100%	100%	92%	100.0%
mod2.c	0%	100%	23%	100%	100%	92%	100.0%
mod3.c	64%	60%	76%	78%	74%	67%	75.0%
mod4.c	88%	77%	94%	89%	86%	79%	84.2%
mod5.c	100%	100%	100%	100%	100%	92%	100.0%
mod6.c	84%	80%	96%	93%	86%	77%	74.4%
mod7.c	69%	53%	81%	70%	35%	69%	100.0%
mod8.c	0%	32%	23%	50%	35%	41%	71.1%
nao-plagio.c	0%	25%	0%	42%	0%	0%	80.0%

Figura B.22 – Código pequeno -
Programador 2 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	92%	100.0%
mod1.c	100%	100%	100%	100%	100%	92%	100.0%
mod2.c	0%	100%	0%	100%	100%	92%	100.0%
mod3.c	62%	57%	80%	76%	74%	67%	75.0%
mod4.c	86%	83%	93%	93%	86%	79%	84.2%
mod5.c	100%	100%	100%	100%	100%	92%	100.0%
mod6.c	78%	80%	96%	100%	86%	77%	74.4%
mod7.c	71%	52%	85%	66%	35%	69%	100.0%
mod8.c	0%	33%	0%	46%	35%	41%	71.1%
nao-plagio.c	0%	0%	0%	33%	0%	0%	80.0%

Figura B.23 – Código pequeno -
Programador 2 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	92%	100.0%
mod1.c	100%	100%	100%	100%	100%	92%	100.0%
mod2.c	0%	100%	0%	100%	100%	92%	100.0%
mod3.c	50%	57%	76%	100%	74%	67%	75.0%
mod4.c	88%	77%	100%	100%	86%	79%	84.2%
mod5.c	100%	100%	100%	100%	100%	92%	100.0%
mod6.c	78%	77%	100%	100%	86%	77%	74.4%
mod7.c	58%	66%	80%	85%	35%	69%	100.0%
mod8.c	0%	50%	0%	80%	35%	41%	71.1%
nao-plagio.c	0%	0%	0%	40%	0%	0%	80.0%

Figura B.24 – Código pequeno -
Programador 2 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	92%	100.0%
mod1.c	100%	100%	100%	100%	100%	92%	100.0%
mod2.c	0%	100%	20%	100%	100%	92%	100.0%
mod3.c	41%	75%	80%	100%	74%	67%	75.0%
mod4.c	85%	66%	100%	100%	86%	79%	84.2%
mod5.c	100%	100%	100%	100%	100%	92%	100.0%
mod6.c	66%	80%	100%	100%	86%	77%	74.4%
mod7.c	57%	75%	75%	100%	35%	69%	100.0%
mod8.c	0%	60%	20%	75%	35%	41%	71.1%
nao-plagio.c	0%	0%	0%	0%	0%	0%	80.0%

Figura B.25 – Código pequeno -
Programador 2 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	92%	100.0%
mod1.c	100%	100%	100%	100%	100%	92%	100.0%
mod2.c	33%	100%	50%	100%	100%	92%	100.0%
mod3.c	42%	50%	100%	100%	74%	67%	75.0%
mod4.c	75%	50%	100%	100%	86%	79%	84.2%
mod5.c	100%	100%	100%	100%	100%	92%	100.0%
mod6.c	60%	66%	100%	100%	86%	77%	74.4%
mod7.c	66%	50%	100%	100%	35%	69%	100.0%
mod8.c	28%	33%	50%	50%	35%	41%	71.1%
nao-plagio.c	0%	0%	0%	0%	0%	0%	80.0%

Figura B.26 – Código pequeno -
Programador 3 (Z=0)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	89%	100.0%
mod1.c	100%	100%	100%	100%	100%	89%	100.0%
mod2.c	27%	100%	41%	100%	100%	89%	100.0%
mod3.c	88%	68%	93%	81%	64%	57%	61.1%
mod4.c	70%	45%	81%	62%	51%	0%	0%
mod5.c	100%	100%	100%	100%	100%	89%	100.0%
mod6.c	42%	32%	67%	56%	0%	0%	50.0%
mod7.c	86%	68%	93%	81%	98%	63%	100.0%
mod8.c	0%	0%	34%	25%	0%	0%	42.8%
nao-plagio.c	0%	0%	0%	0%	0%	0%	48.4%

Figura B.27 – Código pequeno -
Programador 3 (Z=1)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	89%	100.0%
mod1.c	100%	100%	100%	100%	100%	89%	100.0%
mod2.c	23%	100%	35%	100%	100%	89%	100.0%
mod3.c	85%	70%	100%	87%	64%	57%	61.1%
mod4.c	72%	42%	88%	66%	51%	0%	0%
mod5.c	100%	100%	100%	100%	100%	89%	100.0%
mod6.c	40%	37%	70%	66%	0%	0%	50.0%
mod7.c	80%	80%	88%	88%	98%	63%	100.0%
mod8.c	0%	0%	35%	33%	0%	0%	42.8%
nao-plagio.c	0%	0%	0%	0%	0%	0%	48.4%

Figura B.28 – Código pequeno -
Programador 3 (Z=2)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	89%	100.0%
mod1.c	100%	100%	100%	100%	100%	89%	100.0%
mod2.c	0%	100%	27%	100%	100%	89%	100.0%
mod3.c	92%	60%	100%	75%	64%	57%	61.1%
mod4.c	78%	50%	90%	75%	51%	0%	0%
mod5.c	100%	100%	100%	100%	100%	89%	100.0%
mod6.c	47%	20%	72%	50%	0%	0%	50.0%
mod7.c	78%	80%	90%	100%	98%	63%	100.0%
mod8.c	0%	0%	27%	25%	0%	0%	42.8%
nao-plagio.c	0%	0%	0%	0%	0%	0%	48.4%

Figura B.29 – Código pequeno -
Programador 3 (Z=3)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	89%	100.0%
mod1.c	100%	100%	100%	100%	100%	89%	100.0%
mod2.c	0%	100%	20%	100%	100%	89%	100.0%
mod3.c	100%	50%	100%	100%	64%	57%	61.1%
mod4.c	62%	33%	83%	100%	51%	0%	0%
mod5.c	100%	100%	100%	100%	100%	89%	100.0%
mod6.c	25%	0%	50%	0%	0%	0%	50.0%
mod7.c	85%	50%	100%	100%	98%	63%	100.0%
mod8.c	0%	0%	0%	0%	0%	0%	42.8%
nao-plagio.c	0%	0%	0%	0%	0%	0%	48.4%

Figura B.30 – Código pequeno -
Programador 3 (Z=4)

	original.c						
Algoritmo:	sherlock	sherlock	sherlock_overlap	sherlock_overlap	sim	moss	jplag
Normalização:	3	4	3	4	1	1	1
mod0.c	100%	100%	100%	100%	100%	89%	100.0%
mod1.c	100%	100%	100%	100%	100%	89%	100.0%
mod2.c	0%	100%	0%	100%	100%	89%	100.0%
mod3.c	100%	100%	100%	100%	64%	57%	61.1%
mod4.c	66%	33%	100%	100%	51%	0%	0%
mod5.c	100%	100%	100%	100%	100%	89%	100.0%
mod6.c	0%	0%	0%	0%	0%	0%	50.0%
mod7.c	100%	100%	100%	100%	98%	63%	100.0%
mod8.c	0%	0%	0%	0%	0%	0%	42.8%
nao-plagio.c	0%	0%	0%	0%	0%	0%	48.4%

Tabelas: precisão, revocação e média harmônica

Tabela C.1 – Dados contabilizados - código médio - programador 1 (Z=0)

Limiar (%)	Sherlock Norm3			Sherlock Norm4			Overlap Norm3			Overlap Norm4			SIM			MOSS			JPlag		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.5556	0.714286	1	0.5556	0.7143	1	0.6667	0.8	1	0.6667	0.8	1	0.7778	0.875	1	0.5556	0.7143	1	0.5556	0.7143
85	1	0.6667	0.8	1	0.6667	0.8000	1	0.6667	0.8	1	0.7778	0.875	1	0.7778	0.875	1	0.6667	0.8000	1	0.5556	0.7143
80	1	0.6667	0.8	1	0.6667	0.8000	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.8750	1	0.6667	0.8000
75	1	0.6667	0.8	1	0.7778	0.8750	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.875	1	0.7778	0.8750	1	0.7778	0.8750
70	1	0.7778	0.875	1	0.7778	0.8750	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.7778	0.8750	1	0.8889	0.9412
65	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
60	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
55	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
50	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
45	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
40	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	0.8889	0.94118	1	0.8889	0.9412	1	0.8889	0.9412	1	0.8889	0.9412
35	1	0.7778	0.875	1	0.8889	0.9412	1	0.7778	0.875	1	1	1	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
30	1	0.7778	0.875	1	0.8889	0.9412	1	0.8889	0.9412	1	1	1	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
25	1	0.7778	0.875	1	1	1	1	0.8889	0.9412	0.9	1	0.94737	0.8889	0.8889	0.8889	1	0.8889	0.9412	1	0.8889	0.9412
20	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.8889	0.8889	0.8889	1	0.8889	0.9412	0.8889	0.8889	0.8889
15	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.9	1	0.9474	1	0.8889	0.9412	0.8889	0.8889	0.8889
10	1	0.7778	0.875	1	1	1	1	1	1	0.9	1	0.94737	0.9	1	0.9474	1	0.8889	0.9412	0.9000	1	0.9474

Tabela C.2 – Dados contabilizados - código médio - programador 1 (Z=1)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1.0000	0.4444	0.6154	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.5556	0.7143	1.0000	0.5556	0.7143
85	1.0000	0.5556	0.7143	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.5556	0.7143
80	1.0000	0.6667	0.8000	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000
75	1.0000	0.6667	0.8000	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750
70	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412
65	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
60	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
55	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
50	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
45	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
40	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
35	1.0000	0.7778	0.8750	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	0.8889	0.8889	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
30	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	0.8889	0.8889	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
25	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	0.8889	0.8889	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412
20	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	0.9000	1.0000	0.9474	0.8889	0.8889	0.8889	1.0000	0.8889	0.9412	0.8889	0.8889	0.8889
15	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	0.9000	1.0000	0.9474	0.9000	1.0000	0.9474	1.0000	0.8889	0.9412	0.8889	0.8889	0.8889
10	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	0.9000	1.0000	0.9474	0.9000	1.0000	0.9474	1.0000	0.8889	0.9412	0.9000	1.0000	0.9474

[illegible][illegible][illegible]

Tabela C.9 – Dados contabilizados - código médio - programador 2 (Z=3)

[illegible]

Tabela C.10 – Dados contabilizados - código médio - programador 2 (Z=4)

[illegible]

Tabela C.11 – Dados contabilizados - código médio - programador 3 (Z=0)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1.0000	0.3333	0.5000	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143
85	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143
80	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143
75	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.5556	0.7143	1.0000	0.5556	0.7143	1.0000	0.4444	0.6154	1.0000	0.4444	0.6154	1.0000	0.6667	0.8000
70	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.5556	0.7143	1.0000	0.4444	0.6154	1.0000	0.7778	0.8750
65	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.5556	0.7143	1.0000	0.5556	0.7143	1.0000	0.7778	0.8750
60	1.0000	0.4444	0.6154	1.0000	0.5556	0.7143	1.0000	0.6667	0.8000	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.5556	0.7143	1.0000	0.7778	0.8750
55	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.5556	0.7143	1.0000	0.7778	0.8750
50	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412
45	1.0000	0.6667	0.8000	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412
40	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.6667	0.8000	1.0000	0.8889	0.9412
35	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000
30	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000
25	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000	1.0000
20	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000
15	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000
10	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.8889	0.9412	1.0000	0.7778	0.8750	1.0000	1.0000

Tabela C.30 – Dados contabilizados - código pequeno - programador 3 (Z=4)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.555556	0.714286	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	0	0	0	1	0.555556	0.714286
85	1	0.555556	0.714286	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	1	0.444444	0.615385	1	0.555556	0.714286
80	1	0.555556	0.714286	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	1	0.444444	0.615385	1	0.555556	0.714286
75	1	0.555556	0.714286	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	1	0.444444	0.615385	1	0.555556	0.714286
70	1	0.555556	0.714286	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	1	0.444444	0.615385	1	0.555556	0.714286
65	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.555556	0.714286	1	0.444444	0.615385	1	0.555556	0.714286
60	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.555556	0.714286	1	0.666667	0.8
55	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8
50	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875
45	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.875	0.777778	0.823529
40	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
35	1	0.666667	0.8	1	0.666667	0.8	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
30	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
25	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
20	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
15	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889
10	1	0.666667	0.8	1	0.777778	0.875	1	0.666667	0.8	1	0.777778	0.875	1	0.777778	0.875	1	0.666667	0.8	0.888889	0.888889	0.888889

Tabela C.31 – Dados para análise de falsos positivos e negativos - código médio para os 3 programadores (Z=0)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.222222	0.363636	1	0.333333	0.5	1	0.340741	0.5082873	1	0.392593	0.5638298	1	0.42963	0.601036	1	0.296296	0.457143	1	0.377778	0.548387
85	1	0.296296	0.457143	1	0.385185	0.5561497	1	0.385185	0.5561497	1	0.511111	0.6764706	1	0.488889	0.656716	1	0.362963	0.532609	1	0.385185	0.55615
80	1	0.377778	0.548387	1	0.392593	0.5638298	1	0.444444	0.6153846	1	0.525926	0.6893204	1	0.488889	0.656716	1	0.481481	0.65	1	0.533333	0.695652
75	1	0.377778	0.548387	1	0.518519	0.6829268	1	0.496296	0.6633663	1	0.525926	0.6893204	1	0.488889	0.656716	1	0.488889	0.656716	1	0.703704	0.826087
70	1	0.414815	0.586387	1	0.525926	0.6893204	1	0.540741	0.7019231	1	0.614815	0.7614679	1	0.562963	0.720379	1	0.488889	0.656716	1	0.792593	0.884298
65	1	0.459259	0.629442	1	0.562963	0.7203791	1	0.562963	0.7203791	1	0.703704	0.826087	1	0.585185	0.738318	1	0.562963	0.720379	1	0.792593	0.884298
60	1	0.474074	0.643216	1	0.57037	0.7264151	1	0.57037	0.7264151	1	0.777778	0.875	1	0.62963	0.772727	1	0.57037	0.726415	1	0.8	0.888889
55	1	0.555556	0.714286	1	0.614815	0.7614679	1	0.57037	0.7264151	1	0.792593	0.8842975	1	0.718519	0.836207	1	0.577778	0.732394	1	0.8	0.888889
50	1	0.562963	0.720379	1	0.622222	0.7671233	1	0.637037	0.7782805	1	0.844444	0.9156027	1	0.77037	0.870293	1	0.62963	0.772727	1	0.844444	0.915663
45	1	0.562963	0.720379	1	0.681481	0.8105727	1	0.666667	0.8	1	0.859259	0.9243028	1	0.785185	0.879668	1	0.62963	0.772727	1	0.844444	0.915663
40	1	0.6	0.75	1	0.748148	0.8559322	1	0.703704	0.826087	1	0.881481	0.9370079	1	0.8	0.888889	1	0.62963	0.772727	1	0.859259	0.924303
35	1	0.614815	0.761468	1	0.792593	0.8842975	1	0.755556	0.8607595	1	0.948148	0.973384	0.931624	0.807407	0.865079	1	0.703704	0.826087	1	0.918519	0.957529
30	1	0.666667	0.8	1	0.814815	0.8979592	1	0.881481	0.9370079	0.992308	0.955556	0.9735849	0.932203	0.814815	0.869565	1	0.77037	0.870293	1	0.940741	0.969466
25	1	0.703704	0.826087	1	0.881481	0.9370079	1	0.940741	0.9694656	0.944056	1	0.971223	0.920635	0.859259	0.888889	1	0.785185	0.879668	1	0.940741	0.969466
20	1	0.822222	0.902439	1	0.933333	0.9655172	1	0.992593	0.9962825	0.924658	1	0.9608541	0.920635	0.859259	0.888889	1	0.8	0.888889	0.940741	0.940741	0.940741
15	1	0.822222	0.902439	1	0.933333	0.9655172	1	0.992593	0.9962825	0.924658	1	0.9608541	0.925926	0.925926	0.925926	1	0.82963	0.906883	0.940741	0.940741	0.940741
10	1	0.822222	0.902439	1	0.933333	0.9655172	1	0.992593	0.9962825	0.924658	1	0.9608541	0.928058	0.955556	0.941606	1	0.837037	0.91129	0.943662	0.992593	0.967509

Tabela C.32 – Dados para análise de falsos positivos e negativos - código médio para os 3 programadores (Z=1)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.192593	0.322981	1	0.362963	0.5326087	1	0.385185	0.5561497	1	0.481481	0.65	1	0.42963	0.601036	1	0.296296	0.457143	1	0.377778	0.548387
85	1	0.251852	0.402367	1	0.385185	0.5561497	1	0.385185	0.5561497	1	0.585185	0.7383178	1	0.488889	0.656716	1	0.362963	0.532609	1	0.385185	0.55615
80	1	0.37037	0.540541	1	0.422222	0.59375	1	0.474074	0.6432161	1	0.614815	0.7614679	1	0.488889	0.656716	1	0.481481	0.65	1	0.533333	0.695652
75	1	0.37037	0.540541	1	0.518519	0.6829268	1	0.511111	0.6764706	1	0.696296	0.8209607	1	0.488889	0.656716	1	0.488889	0.656716	1	0.703704	0.826087
70	1	0.414815	0.586387	1	0.577778	0.7323944	1	0.555556	0.7142857	1	0.718519	0.8362069	1	0.562963	0.720379	1	0.488889	0.656716	1	0.792593	0.884298
65	1	0.459259	0.629442	1	0.577778	0.7323944	1	0.57037	0.7264151	1	0.777778	0.875	1	0.585185	0.738318	1	0.562963	0.720379	1	0.792593	0.884298
60	1	0.466667	0.636364	1	0.577778	0.7323944	1	0.57037	0.7264151	1	0.8	0.888889	1	0.62963	0.772727	1	0.57037	0.726415	1	0.8	0.888889
55	1	0.555556	0.714286	1	0.585185	0.7383178	1	0.57037	0.7264151	1	0.822222	0.902439	1	0.718519	0.836207	1	0.577778	0.732394	1	0.8	0.888889
50	1	0.555556	0.714286	1	0.681481	0.8105727	1	0.637037	0.7782805	1	0.925926	0.9615385	1	0.77037	0.870293	1	0.62963	0.772727	1	0.844444	0.915663
45	1	0.562963	0.720379	1	0.762963	0.8655462	1	0.659259	0.7946429	1	0.933333	0.9655172	1	0.785185	0.879668	1	0.62963	0.772727	1	0.844444	0.915663
40	1	0.6	0.75	1	0.792593	0.8842975	1	0.703704	0.826087	1	0.933333	0.9655172	1	0.8	0.888889	1	0.62963	0.772727	1	0.859259	0.924303
35	1	0.607407	0.75576	1	0.822222	0.902439	1	0.822222	0.902439	1	0.955556	0.9772727	0.931624	0.807407	0.865079	1	0.703704	0.826087	1	0.918519	0.957529
30	1	0.659259	0.794643	1	0.874074	0.9328063	1	0.911111	0.9534884	1	0.962963	0.9811321	0.932203	0.814815	0.869565	1	0.77037	0.870293	1	0.940741	0.969466
25	1	0.688889	0.815789	1	0.881481	0.9370079	1	0.940741	0.9694656	0.985401	1	0.992647	0.920635	0.859259	0.888889	1	0.785185	0.879668	1	0.940741	0.969466
20	1	0.785185	0.879668	1	0.925926	0.9615385	1	0.955556	0.9772727	0.931034	1	0.9642857	0.920635	0.959259	0.888889	1	0.8	0.888889	0.940741	0.940741	0.940741
15	1	0.785185	0.879668	1	0.925926	0.9615385	1	0.955556	0.9772727	0.931034	1	0.9642857	0.925926	0.925926	0.925926	1	0.82963	0.906883	0.940741	0.940741	0.940741
10	1	0.785185	0.879668	1	0.925926	0.9615385	1	0.955556	0.9772727	0.931034	1	0.9642857	0.928058	0.955556	0.941606	1	0.837037	0.91129	0.943662	0.992593	0.967509

Tabela C.33 – Dados para análise de falsos positivos e negativos - código médio para os 3 programadores (Z=2)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.192593	0.322981	1	0.340741	0.5082873	1	0.377778	0.5483871	1	0.57037	0.7264151	1	0.42963	0.601036	1	0.296296	0.457143	1	0.377778	0.548387
85	1	0.251852	0.402367	1	0.451852	0.622449	1	0.422222	0.59375	1	0.585185	0.7383178	1	0.488889	0.656716	1	0.362963	0.532609	1	0.385185	0.55615
80	1	0.355556	0.52459	1	0.525926	0.6893204	1	0.459259	0.6294416	1	0.651852	0.7892377	1	0.488889	0.656716	1	0.481481	0.65	1	0.533333	0.695652
75	1	0.37037	0.540541	1	0.525926	0.6893204	1	0.540741	0.7019231	1	0.711111	0.8311688	1	0.488889	0.656716	1	0.488889	0.656716	1	0.703704	0.826087
70	1	0.414815	0.586387	1	0.525926	0.6893204	1	0.57037	0.7264151	1	0.725926	0.8412017	1	0.562963	0.720379	1	0.488889	0.656716	1	0.792593	0.884298
65	1	0.459259	0.629442	1	0.57037	0.7264151	1	0.57037	0.7264151	1	0.837037	0.9112903	1	0.585185	0.738318	1	0.562963	0.720379	1	0.792593	0.884298
60	1	0.496296	0.663366	1	0.585185	0.7383178	1	0.607407	0.7557604	1	0.859259	0.9243028	1	0.62963	0.772727	1	0.57037	0.726415	1	0.8	0.888889
55	1	0.518519	0.682927	1	0.62963	0.7727273	1	0.622222	0.7671233	1	0.859259	0.9243028	1	0.718519	0.836207	1	0.577778	0.732394	1	0.8	0.888889
50	1	0.555556	0.714286	1	0.725926	0.8412017	1	0.666667	0.8	1	0.896296	0.9453125	1	0.77037	0.870293	1	0.62963	0.772727	1	0.844444	0.915663
45	1	0.592593	0.744186	1	0.733333	0.8461538	1	0.688889	0.8157895	1	0.940741	0.9694656	1	0.785185	0.879668	1	0.62963	0.772727	1	0.844444	0.915663
40	1	0.607407	0.75576	1	0.77037	0.8702929	1	0.718519	0.8362069	1	0.940741	0.9694656	1	0.8	0.888889	1	0.62963	0.772727	1	0.859259	0.924303
35	1	0.659259	0.794643	1	0.814815	0.8979592	1	0.814815	0.8979592	1	0.940741	0.9694656	0.931624	0.807407	0.865079	1	0.703704	0.826087	1	0.918519	0.957529
30	1	0.681481	0.810573	1	0.814815	0.8979592	1	0.940741	0.9694656	1	1	1	0.932203	0.814815	0.869565	1	0.77037	0.870293	1	0.940741	0.969466
25	1	0.696296	0.820961	1	0.874074	0.9328063	1	0.962963	0.9811321	0.912162	1	0.9540636	0.920635	0.859259	0.888889	1	0.785185	0.879668	1	0.940741	0.969466
20	1	0.844444	0.915663	1	0.925926	0.9615385	1	1	1	0.89404	1	0.9440559	0.920635	0.859259	0.888889	1	0.8	0.888889	0.940741	0.940741	0.940741
15	1	0.844444	0.915663	1	0.925926	0.9615385	1	1	1	0.89404	1	0.9440559	0.925926	0.925926	0.925926	1	0.82963	0.906883	0.940741	0.940741	0.940741
10	1	0.844444	0.915663	1	0.925926	0.9615385	1	1	1	0.89404	1	0.9440559	0.928058	0.955556	0.941606	1	0.837037	0.91129	0.943662	0.992593	0.967509

Tabela C.34 – Dados para análise de falsos positivos e negativos - código médio para os 3 programadores (Z=3)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.192593	0.322981	1	0.348148	0.5164835	1	0.414815	0.5863874	1	0.481481	0.65	1	0.42963	0.601036	1	0.296296	0.457143	1	0.377778	0.548387
85	1	0.192593	0.322981	1	0.42963	0.6010363	1	0.466667	0.6363636	1	0.585185	0.7383178	1	0.488889	0.656716	1	0.362963	0.532609	1	0.385185	0.55615
80	1	0.333333	0.5	1	0.525926	0.6893204	1	0.503704	0.6699507	1	0.622222	0.7671233	1	0.488889	0.656716	1	0.481481	0.65	1	0.533333	0.695652
75	1	0.340741	0.508287	1	0.525926	0.6893204	1	0.548148	0.708134	1	0.666667	0.8	1	0.488889	0.656716	1	0.488889	0.656716	1	0.703704	0.826087
70	1	0.407407	0.578947	1	0.525926	0.6893204	1	0.562963	0.7203791	1	0.674074	0.8053097	1	0.562963	0.720379	1	0.488889	0.656716	1	0.792593	0.884298
65	1	0.459259	0.629442	1	0.57037	0.7264151	1	0.614815	0.7614679	1	0.822222	0.902439	1	0.585185	0.738318	1	0.562963	0.720379	1	0.792593	0.884298
60	1	0.525926	0.68932	1	0.614815	0.7614679	1	0.644444	0.7837838	1	0.82963	0.9068826	1	0.62963	0.772727	1	0.57037	0.726415	1	0.8	0.888889
55	1	0.533333	0.695652	1	0.62963	0.7727273	1	0.703704	0.826087	1	0.851852	0.92	1	0.718519	0.836207	1	0.577778	0.732394	1	0.8	0.888889
50	1	0.585185	0.738318	1	0.681481	0.8105727	1	0.755556	0.8607595	0.959016	0.866667	0.9105058	1	0.77037	0.870293	1	0.62963	0.772727	1	0.844444	0.915663
45	1	0.6	0.75	1	0.688889	0.8157895	1	0.785185	0.879668	0.959016	0.866667	0.9105058	1	0.785185	0.879668	1	0.62963	0.772727	1	0.844444	0.915663
40	1	0.66	0.75	1	0.792593	0.8842975	1	0.844444	0.9156627	0.951613	0.874074	0.9111969	1	0.8	0.888889	1	0.62963	0.772727	1	0.859259	0.924303
35	1	0.659259	0.794643	1	0.807407	0.8934426	1	0.918519	0.957529	0.954198	0.925926	0.9398496	0.931624	0.807407	0.865079	1	0.703704	0.826087	1	0.918519	0.957529
30	1	0.674074	0.80531	1	0.814815	0.8979592	1	0.948148	0.973384	0.947761	0.940741	0.94442379	0.932203	0.814815	0.869565	1	0.77037	0.870293	1	0.940741	0.969466
25	1	0.755556	0.860759	1	0.814815	0.8979592	1	0.955556	0.9772727	0.933824	0.940741	0.9372694	0.920635	0.859259	0.888889	1	0.785185	0.879668	1	0.940741	0.969466
20	1	0.881481	0.937008	0.948718	0.822222	0.8809524	1	0.962963	0.9811321	0.933824	0.940741	0.9372694	0.920635	0.859259	0.888889	1	0.8	0.888889	0.940741	0.940741	0.940741
15	1	0.881481	0.937008	0.948718	0.822222	0.8809524	1	0.962963	0.9811321	0.933824	0.940741	0.9372694	0.925926	0.925926	0.925926	1	0.82963	0.906883	0.940741	0.940741	0.940741
10	1	0.881481	0.937008	0.948718	0.822222	0.8809524	1	0.962963	0.9811321	0.933824	0.940741	0.9372694	0.928058	0.955556	0.941606	1	0.837037	0.91129	0.943662	0.992593	0.967509

Tabela C.35 – Dados para análise de falsos positivos e negativos - código médio para os 3 programadores (Z=4)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.22963	0.373494	1	0.42963	0.6010363	1	0.466667	0.6363636	1	0.525926	0.6893204	1	0.42963	0.601036	1	0.296296	0.457143	1	0.377778	0.548387
85	1	0.296296	0.457143	1	0.481481	0.65	1	0.466667	0.6363636	1	0.525926	0.6893204	1	0.488889	0.656716	1	0.362963	0.532609	1	0.385185	0.55615
80	1	0.333333	0.5	1	0.481481	0.65	1	0.533333	0.6956522	1	0.725926	0.8412017	1	0.488889	0.656716	1	0.481481	0.65	1	0.533333	0.695652
75	1	0.377778	0.548387	1	0.57037	0.7264151	1	0.57037	0.7264151	1	0.748148	0.8559322	1	0.488889	0.656716	1	0.488889	0.656716	1	0.703704	0.826087
70	1	0.377778	0.548387	1	0.57037	0.7264151	1	0.585185	0.7383178	1	0.748148	0.8559322	1	0.562963	0.720379	1	0.488889	0.656716	1	0.792593	0.884298
65	1	0.414815	0.586387	1	0.57037	0.7264151	1	0.637037	0.7782805	1	0.755556	0.8607595	1	0.585185	0.738318	1	0.562963	0.720379	1	0.792593	0.884298
60	1	0.488889	0.656716	1	0.585185	0.7383178	1	0.703704	0.826087	1	0.851852	0.92	1	0.62963	0.772727	1	0.57037	0.726415	1	0.8	0.888889
55	1	0.57037	0.726415	1	0.637037	0.7782805	1	0.703704	0.826087	1	0.851852	0.92	1	0.718519	0.836207	1	0.577778	0.732394	1	0.8	0.888889
50	1	0.592593	0.744186	1	0.644444	0.7837838	1	0.792593	0.8842975	1	0.859259	0.9243028	1	0.77037	0.870293	1	0.62963	0.772727	1	0.844444	0.915663
45	1	0.6	0.75	1	0.644444	0.7837838	1	0.8	0.8888889	1	0.859259	0.9243028	1	0.785185	0.879668	1	0.62963	0.772727	1	0.844444	0.915663
40	1	0.651852	0.789238	1	0.733333	0.8461538	1	0.881481	0.9370079	0.944	0.874074	0.9076923	1	0.8	0.888889	1	0.62963	0.772727	1	0.859259	0.924303
35	1	0.702963	0.865546	1	0.792593	0.8842975	1	0.933333	0.9655172	0.944	0.874074	0.9076923	0.931624	0.807407	0.865079	1	0.703704	0.826087	1	0.918519	0.957529
30	1	0.851852	0.92	1	0.8	0.8888889	1	0.940741	0.9694656	0.940741	0.940741	0.9407407	0.932203	0.814815	0.809565	1	0.77037	0.870293	1	0.940741	0.969466
25	1	0.925926	0.961538	1	0.866667	0.9285714	1	0.948148	0.973384	0.940741	0.940741	0.9407407	0.920635	0.859259	0.888889	1	0.785185	0.879668	1	0.940741	0.969466
20	1	0.940741	0.969466	1	0.866667	0.9285714	0.914286	0.948148	0.9309091	0.838509	1	0.9121622	0.920635	0.859259	0.888889	1	0.8	0.888889	0.940741	0.940741	0.940741
15	1	0.940741	0.969466	1	0.866667	0.9285714	0.914286	0.948148	0.9309091	0.838509	1	0.9121622	0.925926	0.925926	0.925926	1	0.82963	0.906883	0.940741	0.940741	0.940741
10	1	0.940741	0.969466	1	0.866667	0.9285714	0.914286	0.948148	0.9309091	0.838509	1	0.9121622	0.928058	0.925926	0.941606	1	0.837037	0.91129	0.943602	0.925923	0.967509

Tabela C.36 – Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores (Z=0)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.111111	0.2	1	0.192593	0.3229814	1	0.281481	0.4393064	1	0.259259	0.4117647	1	0.259259	0.411765	1	0.118519	0.211921	1	0.296296	0.457143
85	1	0.222222	0.363636	1	0.192593	0.3229814	1	0.340741	0.5082873	1	0.333333	0.5	1	0.37037	0.540541	1	0.192593	0.322981	1	0.296296	0.457143
80	1	0.259259	0.411765	1	0.22963	0.373494	1	0.422222	0.59375	1	0.407407	0.5789474	1	0.37037	0.540541	1	0.192593	0.322981	0.87931	0.377778	0.528497
75	1	0.311111	0.474576	1	0.296296	0.4571429	1	0.496296	0.6633663	1	0.496296	0.6633663	1	0.37037	0.540541	1	0.303704	0.465909	0.878788	0.42963	0.577114
70	1	0.362963	0.532609	1	0.333333	0.5	1	0.555556	0.7142857	1	0.540741	0.7019231	1	0.422222	0.59375	1	0.333333	0.5	0.888889	0.533333	0.666667
65	1	0.407407	0.578947	1	0.407407	0.5789474	1	0.585185	0.7383178	1	0.540741	0.7019231	1	0.42963	0.601036	1	0.422222	0.59375	0.879518	0.540741	0.669725
60	1	0.481481	0.65	1	0.444444	0.6153846	1	0.614815	0.7614679	1	0.607407	0.7557604	1	0.466667	0.636364	1	0.459259	0.629442	0.891304	0.607407	0.722467
55	1	0.540741	0.701923	1	0.459259	0.6294416	1	0.637037	0.7782805	1	0.703704	0.826087	1	0.488889	0.656716	1	0.548148	0.708134	0.891304	0.607407	0.722467
50	1	0.555556	0.714286	1	0.540741	0.7019231	1	0.644444	0.7837838	1	0.814815	0.8979592	1	0.577778	0.732394	1	0.562963	0.720379	0.903846	0.696296	0.786611
45	1	0.555556	0.714286	1	0.585185	0.7383178	1	0.681481	0.8105727	0.991228	0.837037	0.9076305	1	0.614815	0.761468	1	0.57037	0.726415	0.849558	0.711111	0.774194
40	1	0.607407	0.75576	1	0.607407	0.7557604	1	0.748148	0.8559322	0.944	0.874074	0.9076923	1	0.666667	0.8	1	0.622222	0.767123	0.858333	0.762963	0.807843
35	1	0.637037	0.778281	1	0.674074	0.8053097	1	0.77037	0.8702929	0.903704	0.903704	0.9037037	1	0.777778	0.875	1	0.62963	0.772727	0.858333	0.762963	0.807843
30	1	0.644444	0.783784	1	0.807407	0.8934426	1	0.82963	0.9068826	0.894366	0.940741	0.9169675	1	0.792593	0.884298	1	0.62963	0.772727	0.858333	0.762963	0.807843
25	1	0.718519	0.836207	0.95082	0.859259	0.9027237	0.92623	0.837037	0.8793774	0.888158	1	0.9407666	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
20	1	0.785185	0.879668	0.900763	0.874074	0.887218	0.927007	0.940741	0.9338235	0.808383	1	0.8940397	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
15	1	0.785185	0.879668	0.900763	0.874074	0.887218	0.927007	0.940741	0.9338235	0.808383	1	0.8940397	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
10	1	0.785185	0.879668	0.900763	0.874074	0.887218	0.927007	0.940741	0.9338235	0.808383	1	0.8940397	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843

Tabela C.37 – Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores (Z=1)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.111111	0.2	1	0.192593	0.3229814	1	0.281481	0.4393064	1	0.325926	0.4916201	1	0.259259	0.411765	1	0.118519	0.211921	1	0.296296	0.457143
85	1	0.192593	0.322981	1	0.192593	0.3229814	1	0.422222	0.59375	1	0.414815	0.5863874	1	0.37037	0.540541	1	0.192593	0.322981	1	0.296296	0.457143
80	1	0.266667	0.421053	1	0.303704	0.4659091	1	0.459259	0.6294416	1	0.444444	0.6153846	1	0.37037	0.540541	1	0.192593	0.322981	0.87931	0.377778	0.528497
75	1	0.325926	0.49162	1	0.362963	0.5326087	1	0.548148	0.708134	1	0.511111	0.6764706	1	0.37037	0.540541	1	0.303704	0.465909	0.878788	0.42963	0.577114
70	1	0.4	0.571429	1	0.4	0.5714286	1	0.592593	0.744186	1	0.511111	0.6764706	1	0.422222	0.59375	1	0.333333	0.5	0.888889	0.533333	0.666667
65	1	0.422222	0.59375	1	0.407407	0.5789474	1	0.592593	0.744186	1	0.688889	0.8157895	1	0.42963	0.601036	1	0.422222	0.59375	0.879518	0.540741	0.669725
60	1	0.518519	0.682927	1	0.407407	0.5789474	1	0.614815	0.7614679	1	0.703704	0.826087	1	0.466667	0.636364	1	0.459259	0.629442	0.891304	0.607407	0.722467
55	1	0.548148	0.708134	1	0.459259	0.6294416	1	0.637037	0.7782805	1	0.740741	0.8510638	1	0.488889	0.656716	1	0.548148	0.708134	0.891304	0.607407	0.722467
50	1	0.562963	0.720379	1	0.57037	0.7264151	1	0.644444	0.7837838	0.956522	0.814815	0.88	1	0.577778	0.732394	1	0.562963	0.720379	0.903846	0.696296	0.786611
45	1	0.562963	0.720379	1	0.577778	0.7323944	1	0.659259	0.7946429	0.95122	0.866667	0.9069767	1	0.614815	0.761468	1	0.57037	0.726415	0.849558	0.711111	0.774194
40	1	0.607407	0.75576	1	0.651852	0.7892377	1	0.659259	0.7946429	0.938462	0.903704	0.9207547	1	0.666667	0.8	1	0.622222	0.767123	0.858333	0.762963	0.807843
35	1	0.614815	0.761468	1	0.718519	0.8362069	1	0.807407	0.8934426	0.938931	0.911111	0.924812	1	0.777778	0.875	1	0.62963	0.772727	0.858333	0.762963	0.807843
30	1	0.637037	0.778281	1	0.785185	0.879668	0.92437	0.814815	0.8661417	0.893333	0.992593	0.9403509	1	0.792593	0.884298	1	0.62963	0.772727	0.858333	0.762963	0.807843
25	1	0.651852	0.789238	0.951613	0.874074	0.9111969	0.918033	0.82963	0.8715953	0.882353	1	0.9375	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
20	1	0.755556	0.800759	0.938462	0.903704	0.9207547	0.92	0.851852	0.8846154	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
15	1	0.755556	0.800759	0.938462	0.903704	0.9207547	0.92	0.851852	0.8846154	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
10	1	0.755556	0.800759	0.938462	0.903704	0.9207547	0.92	0.851852	0.8846154	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843

Tabela C.38 – Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores (Z=2)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.162963	0.280255	1	0.192593	0.3229814	1	0.4	0.5714286	1	0.474074	0.6432161	1	0.259259	0.411765	1	0.118519	0.211921	1	0.296296	0.457143
85	1	0.192593	0.322981	1	0.192593	0.3229814	1	0.407407	0.5789474	1	0.511111	0.6764706	1	0.37037	0.540541	1	0.192593	0.322981	1	0.296296	0.457143
80	1	0.22963	0.373494	1	0.237037	0.3832335	1	0.503704	0.6699507	1	0.555556	0.7142857	1	0.37037	0.540541	1	0.192593	0.322981	0.87931	0.377778	0.528497
75	1	0.37037	0.540541	1	0.340741	0.5082873	1	0.540741	0.7019231	1	0.674074	0.8053097	1	0.37037	0.540541	1	0.303704	0.465909	0.878788	0.42963	0.577114
70	1	0.392593	0.56383	1	0.340741	0.5082873	1	0.577778	0.7323944	1	0.674074	0.8053097	1	0.422222	0.59375	1	0.333333	0.5	0.888889	0.533333	0.666667
65	1	0.4	0.571429	1	0.407407	0.5789474	1	0.592593	0.744186	1	0.703704	0.826087	1	0.42963	0.601036	1	0.422222	0.59375	0.879518	0.540741	0.669725
60	1	0.451852	0.622449	1	0.451852	0.622449	1	0.6	0.75	0.989796	0.718519	0.832618	1	0.466667	0.636364	1	0.459259	0.629442	0.891304	0.607407	0.722467
55	1	0.488889	0.656716	1	0.496296	0.6633663	1	0.622222	0.7671233	0.989796	0.718519	0.832618	1	0.488889	0.656716	1	0.548148	0.708134	0.891304	0.607407	0.722467
50	1	0.540741	0.701923	1	0.62963	0.7727273	1	0.637037	0.7782805	0.99115	0.82963	0.9032258	1	0.577778	0.732394	1	0.562963	0.720379	0.903846	0.696296	0.786611
45	1	0.592593	0.744186	1	0.62963	0.7727273	1	0.644444	0.7837838	0.99115	0.82963	0.9032258	1	0.614815	0.761468	1	0.57037	0.726415	0.849558	0.711111	0.774194
40	1	0.6	0.75	1	0.696296	0.8209607	1	0.651852	0.789377	0.934959	0.851852	0.8914729	1	0.666667	0.8	1	0.622222	0.767123	0.858333	0.762963	0.807843
35	1	0.622222	0.767123	1	0.696296	0.8209607	1	0.711111	0.831688	0.934959	0.851852	0.8914729	1	0.777778	0.875	1	0.62963	0.772727	0.858333	0.762963	0.807843
30	1	0.62963	0.772727	1	0.725926	0.8412047	1	0.718519	0.8362609	0.911111	0.911111	0.911111	1	0.792593	0.884298	1	0.62963	0.772727	0.858333	0.762963	0.807843
25	1	0.644444	0.783784	0.990385	0.762963	0.8619217	1	0.814815	0.897592	0.899329	0.992593	0.943662	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
20	1	0.696296	0.820961	0.99115	0.82963	0.9032258	0.916667	0.814815	0.8627451	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
15	1	0.696296	0.820961	0.99115	0.82963	0.9032258	0.916667	0.814815	0.8627451	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
10	1	0.696296	0.820961	0.99115	0.82963	0.9032258	0.916667	0.814815	0.8627451	0.870968	1	0.9310345	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843

Tabela C.39 – Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores (Z=3)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.162963	0.280255	1	0.148148	0.2580645	1	0.37037	0.5405405	1	0.451852	0.622449	1	0.259259	0.411765	1	0.118519	0.211921	1	0.296296	0.457143
85	1	0.22963	0.373494	1	0.148148	0.2580645	1	0.37037	0.5405405	1	0.451852	0.622449	1	0.37037	0.540541	1	0.192593	0.322981	1	0.296296	0.457143
80	1	0.288889	0.448276	1	0.192593	0.3229814	1	0.481481	0.65	1	0.459259	0.6294416	1	0.37037	0.540541	1	0.192593	0.322981	0.87931	0.377778	0.528497
75	1	0.288889	0.448276	1	0.274074	0.4302326	1	0.525926	0.6893204	1	0.503704	0.6699507	1	0.37037	0.540541	1	0.303704	0.465909	0.878788	0.42963	0.577114
70	1	0.318519	0.483146	1	0.274074	0.4302326	1	0.548148	0.708134	1	0.503704	0.6699507	1	0.422222	0.59375	1	0.333333	0.5	0.888889	0.533333	0.666667
65	1	0.362963	0.532609	1	0.311111	0.4745763	1	0.57037	0.7264151	1	0.518519	0.6829268	1	0.42963	0.601036	1	0.422222	0.59375	0.879518	0.540741	0.669725
60	1	0.414815	0.586387	1	0.362963	0.5326087	1	0.57037	0.7264151	1	0.518519	0.6829268	1	0.466667	0.636364	1	0.459259	0.629442	0.891304	0.607407	0.722467
55	1	0.488889	0.656716	1	0.37037	0.5405405	1	0.57037	0.7264151	1	0.518519	0.6829268	1	0.488889	0.656716	1	0.548148	0.708134	0.891304	0.607407	0.722467
50	1	0.503704	0.669951	1	0.466667	0.6363636	1	0.637037	0.7782805	1	0.562963	0.7203791	1	0.577778	0.732394	1	0.562963	0.720379	0.903846	0.696296	0.786611
45	1	0.511111	0.676471	1	0.466667	0.6363636	1	0.637037	0.7782805	1	0.562963	0.7203791	1	0.614815	0.761468	1	0.57037	0.726415	0.849558	0.711111	0.774194
40	1	0.562963	0.720379	1	0.481481	0.65	1	0.674074	0.8053097	1	0.57037	0.7264151	1	0.666667	0.8	1	0.622222	0.767123	0.858333	0.762963	0.807843
35	1	0.57037	0.726415	1	0.481481	0.65	1	0.674074	0.8053097	1	0.57037	0.7264151	1	0.777778	0.875	1	0.62963	0.772727	0.858333	0.762963	0.807843
30	1	0.585185	0.738318	1	0.525926	0.6893204	1	0.696296	0.8209607	0.916667	0.57037	0.7031963	1	0.792593	0.884298	1	0.62963	0.772727	0.858333	0.762963	0.807843
25	1	0.62963	0.772727	1	0.555556	0.7142857	0.718519	0.718519	0.7185185	0.916667	0.57037	0.7031963	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
20	1	0.688889	0.815789	0.949367	0.555556	0.7009346	0.561576	0.844444	0.6745562	0.916667	0.57037	0.7031963	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
15	1	0.688889	0.815789	0.949367	0.555556	0.7009346	0.561576	0.844444	0.6745562	0.916667	0.57037	0.7031963	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
10	1	0.688889	0.815789	0.949367	0.555556	0.7009346	0.561576	0.844444	0.6745562	0.916667	0.57037	0.7031963	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843

Tabela C.40 – Dados para análise de falsos positivos e negativos - código pequeno para os 3 programadores (Z=4)

Limiar(%)	Sherlock Norm3			Sherlock Norm4			Sherlock Overlap Norm3			Sherlock Overlap Norm4			SIM			MOSS			JPLAG		
	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f	p	r	f
90	1	0.2	0.333333	1	0.22963	0.373494	0.886076	0.518519	0.6542056	1	0.488889	0.6567164	1	0.259259	0.411765	1	0.118519	0.211921	1	0.296296	0.457143
85	1	0.2	0.333333	1	0.22963	0.373494	0.886076	0.518519	0.6542056	1	0.488889	0.6567164	1	0.37037	0.540541	1	0.192593	0.322981	1	0.296296	0.457143
80	1	0.2	0.333333	1	0.22963	0.373494	0.886076	0.518519	0.6542056	1	0.488889	0.6567164	1	0.37037	0.540541	1	0.192593	0.322981	0.87931	0.377778	0.528497
75	1	0.288889	0.448276	1	0.22963	0.373494	0.888889	0.533333	0.6666667	1	0.488889	0.6567164	1	0.37037	0.540541	1	0.303704	0.465909	0.878788	0.42963	0.577114
70	1	0.288889	0.448276	1	0.22963	0.373494	0.888889	0.533333	0.6666667	1	0.488889	0.6567164	1	0.422222	0.59375	1	0.333333	0.5	0.888889	0.533333	0.666667
65	1	0.4	0.571429	1	0.274074	0.4302326	0.9	0.6	0.72	1	0.496296	0.6633663	1	0.42963	0.601036	1	0.422222	0.59375	0.879518	0.540741	0.669725
60	1	0.437037	0.608247	1	0.274074	0.4302326	0.9	0.6	0.72	1	0.496296	0.6633663	1	0.466667	0.636364	1	0.459259	0.629442	0.891304	0.607407	0.722467
55	1	0.444444	0.615385	1	0.274074	0.4302326	0.9	0.6	0.72	1	0.496296	0.6633663	1	0.488889	0.656716	1	0.548148	0.708134	0.891304	0.607407	0.722467
50	1	0.511111	0.676471	1	0.4	0.5714286	0.62585	0.681481	0.6524823	0.924051	0.540741	0.682243	1	0.577778	0.732394	1	0.562963	0.720379	0.903846	0.696296	0.786611
45	1	0.511111	0.676471	1	0.4	0.5714286	0.62585	0.681481	0.6524823	0.924051	0.540741	0.682243	1	0.614815	0.761468	1	0.57037	0.726415	0.849558	0.711111	0.774194
40	1	0.57037	0.726415	1	0.407407	0.5789474	0.62585	0.681481	0.6524823	0.924051	0.540741	0.682243	1	0.666667	0.8	1	0.622222	0.767123	0.858333	0.762963	0.807843
35	1	0.585185	0.738318	1	0.407407	0.5789474	0.62585	0.681481	0.6524823	0.924051	0.540741	0.682243	1	0.777778	0.875	1	0.62963	0.772727	0.858333	0.762963	0.807843
30	0.988372	0.62963	0.769231	0.971831	0.511111	0.6699029	0.510204	0.740741	0.6042296	0.925	0.548148	0.6883721	1	0.792593	0.884298	1	0.62963	0.772727	0.858333	0.762963	0.807843
25	0.87619	0.681481	0.766667	0.947368	0.533333	0.6824645	0.492958	0.777778	0.6034483	0.925	0.548148	0.6883721	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
20	0.685714	0.711111	0.698182	0.924051	0.540741	0.682243	0.495327	0.785185	0.6074499	0.925	0.548148	0.6883721	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
15	0.685714	0.711111	0.698182	0.924051	0.540741	0.682243	0.495327	0.785185	0.6074499	0.925	0.548148	0.6883721	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843
10	0.685714	0.711111	0.698182	0.924051	0.540741	0.682243	0.495327	0.785185	0.6074499	0.925	0.548148	0.6883721	0.990741	0.792593	0.880658	1	0.62963	0.772727	0.858333	0.762963	0.807843

Apêndice D

Artigos Publicados

ANÁLISE DE SIMILARIDADE DE CÓDIGOS-FONTE COMO ESTRATÉGIA PARA O ACOMPANHAMENTO DE ATIVIDADES DE LABORATÓRIO DE PROGRAMAÇÃO

Danilo Leal Maciel¹ - daniloleal@alu.ufc.br

José Marques Soares¹ - marques@ufc.br

Allyson Bonetti França¹ - allysonbonetti@gmail.com

Danielo Gonçalves Gomes¹ - danielo@ufc.br

¹Programa de Pós-Graduação em Engenharia de Teleinformática (PPGETI/UFC)

Resumo. *Este trabalho apresenta a implementação e a avaliação de uma ferramenta de apoio ao acompanhamento de atividades laboratoriais de programação de computadores. Um suporte automatizado instrumentaliza o professor para a comparação e a análise de similaridade entre as soluções apresentadas pelos alunos, para os problemas de programação propostos. A fim de desconsiderar diferenças irrelevantes entre códigos-fonte, foram desenvolvidas técnicas para normalizar os códigos antes do processo de comparação realizado pelo algoritmo Sherlock, frequentemente empregado para detecção de plágio em documentos de caráter geral. As técnicas desenvolvidas apresentam resultados equivalentes aos das melhores ferramentas usadas para a detecção de plágio. A proposição do problema pelo professor, o envio das soluções dos alunos e a análise de similaridade são feitos através de um módulo implantado no Moodle, facilitando a gestão desse tipo de atividade em turmas numerosas. O sistema foi experimentado em uma turma de programação ao longo de um semestre e os resultados são apresentados nesse artigo.*

Palavras-chave: *análise de similaridade, laboratório de programação, avaliação.*

SIMILARITY ANALYSIS OF SOURCE CODE AS A STRATEGY FOR MONITORING OF PROGRAMMING LABORATORY ACTIVITIES

Abstract. *This paper presents the implementation and evaluation of a tool to support the monitoring of computer programming laboratory activities. This tool assists the professor in the analysis of similarity between the students' solutions of the proposed activities. In order to ignore irrelevant differences between source-codes, techniques have been developed to normalize the source-codes before the comparison performed by the algorithm Sherlock, often used for detecting plagiarism in documents of a general nature. The techniques developed have equivalent results to those of the best tools used to detect plagiarism. The proposition of the problem by the teacher, the upload of the solution by the student, and the similarity analysis is done through a module deployed in Moodle, facilitating the management of this type of activity in large classes. The system was tested on a programming class over a semester and the results are presented in this article.*

Keywords: *similarity analysis, laboratory programming, assessment.*

1. Introdução

Para dar suporte ao trabalho com turmas numerosas e minimizar as dificuldades associadas aos requisitos de acompanhamento de alunos, professores buscam usar a tecnologia para facilitar o gerenciamento de turmas grandes. Alguns ambientes virtuais, como, por exemplo, o Moodle (Dougiamas e Taylor, 2003), podem ser usados para disponibilização de notas de aula, proposta e submissão de trabalhos e registro de notas. Outras ferramentas são mais especificamente adequadas para auxiliar nas tarefas de compilação e execução dos programas desenvolvidos pelos alunos, como o VPL (2012) e o Onlinejudge (2012). No entanto, embora o uso de tais ferramentas possa mitigar os problemas de natureza organizacional e de conferência de resultados em práticas laboratoriais, não são suficientes para contornar todas as dificuldades inerentes ao acompanhamento individualizado.

Um problema não raramente encontrado em laboratórios de programação é a cópia total ou parcial de soluções entre colegas, frequentemente com a mudança de nomes de variáveis ou com a inserção de comentários, para dificultar a percepção da ação. Em cenários de turmas numerosas, a detecção deste tipo de conduta se torna mais problemática.

Embora a detecção de plágio seja um aspecto importante na condução de uma turma de programação, os estudos desenvolvidos neste trabalho revelam que os resultados da comparação entre códigos podem indicar outros aspectos que são de natureza irrepreensível. Dependendo da perspectiva e do contexto, a semelhança pode ser indicativa de parceria, de trabalho colaborativo, de referência ou apoio sobre solução encontrada em livros ou exemplos fornecidos pelo professor, entre outros motivos. Por essa razão, ao invés de “detecção de plágio”, adota-se neste trabalho a expressão “análise de similaridade”.

A detecção de plágio (e a análise de similaridade) em código-fonte vem sendo estudada em diversos trabalhos (Hage *et al.*, 2010) (Bin-Habtoor, 2012) (Djuric e Gasevic, 2012) e algumas ferramentas foram disponibilizadas para este fim (Prechelt, 2002) (MOSS, 2012). Neste trabalho, avalia-se especialmente a sensibilidade do algoritmo Sherlock (Pike, 2012) quando os códigos são submetidos a técnicas de pré-processamento que visam a normalização antes da comparação. O objetivo é reduzir a diferenciação dos códigos por aspectos irrelevantes, como comentários, valores literais ou nomes de variáveis, entre outros aspectos. Os resultados são comparados com aqueles obtidos com o uso do JPlag (Prechelt, 2002) e MOSS (2012). Os testes foram realizados em duas etapas, na primeira, a partir de dois códigos base, foram geradas variações baseadas nas principais modificações que comumente são realizadas ao se plagiar um código, no segundo utilizou-se de 32 trabalhos de programação propostos e desenvolvidos para uma turma iniciante com 97 alunos.

A função de análise de similaridade corresponde a um dos módulos do BOCA-LAB (França e Soares, 2011), ferramenta que foi integrada ao Moodle e permite a submissão *online* das atividades de laboratório desenvolvidas nas disciplinas de programação. O BOCA-LAB compila e executa remotamente as soluções enviadas pelos alunos, efetuando, ainda, a verificação das saídas geradas pelos programas. As técnicas de pré-processamento para o uso do Sherlock e a análise comparativa dos resultados alcançados representam a principal contribuição deste trabalho, o que permitiu a implementação de melhorias no módulo de análise de similaridade do BOCA-LAB.

Este artigo está organizado da seguinte maneira: após a introdução, discutem-se,

na seção 2, as principais ferramentas para análise de similaridade; na seção 3, enumeram-se os detalhes das técnicas de normalização desenvolvidas neste trabalho. Posteriormente, na seção 4, exploram-se os resultados obtidos com essas técnicas analisando códigos em duas abordagens: manual e com códigos produzidos por alunos de graduação, seguindo, na seção 5, apresenta-se como a ferramenta alvo deste trabalho está inserida contexto do ambiente Moodle e a importância para o professor. Por fim, são apresentadas as conclusões e as perspectivas para o trabalho.

2. Ferramentas para Análise de Similaridade

Diversas ferramentas podem ser utilizadas para realizar a análise de similaridade entre códigos-fonte, tais como SIM (Grune e Vakgroep, 1989), YAP (Lancaster e Culwin, 2001), JPlag (Prechelt, 2002), SID (Chen *et al.*, 2004), Plaggie (Ahtiainen, 2006) e MOSS (2012). Estudos comparativos sobre algumas destas ferramentas foram realizados por Hage *et al.* (2010) e Kleiman (2007). De modo geral, ferramentas especificamente construídas para esta finalidade permitem encontrar semelhanças no caso de alterações de nome de variáveis, nome de funções, comentários ou, ainda, alterando-se a ordem de partes do código. Segundo Burrows *et al.* (2007), JPlag e MOSS são algumas das mais importantes e mais citadas para detecção de plágio, que são sucintamente descritos nos próximos parágrafos, seguidos pelo Sherlock, sobre o qual se apóiam as contribuições deste trabalho.

O JPlag (Prechelt, 2002) é uma ferramenta desenvolvida em Java para análise de similaridade entre códigos-fonte. É disponibilizada exclusivamente por meio de um *WebService* e possui código fechado. Ela requer o cadastramento e a requisição de autorização para o acesso ao serviço. O envio de arquivos para comparação é realizado através de um *applet* que retorna, em formato HTML, o resultado do processamento. A semelhança é apresentada em percentual apenas para aqueles com maior similaridade. Não é disponibilizada, portanto, a visualização de pares com porcentagens pequenas.

O MOSS (*Measure of Software Similarity*) (MOSS, 2012) também é acessado por meio de um *WebService*, disponibilizado na Universidade da Califórnia. Para usá-lo, é necessário realizar um cadastramento por *email*. Os arquivos são enviados para o servidor por um *script* fornecido pelo seu desenvolvedor. Ao final do envio, é gerada uma URL que fornece o resultado da comparação. Segundo informações obtidas na página Web da aplicação, o desenvolvimento do MOSS iniciou-se em 1994, apresentando uma melhora significativa em relação a outros algoritmos (que não são mencionados) para detecção de plágio. O MOSS é um sistema de código fechado baseado no algoritmo Winnowing (Schleimer, 2003).

O Sherlock (Pike, 2012) é uma alternativa às ferramentas apresentadas precedentemente por ser de código aberto, permitindo modificações e melhorias. Além disso, foi possível implantá-la no BOCA-LAB, sem exigir conectividade com *WebServices* de terceiros, o que por si só já representa perda performática expressiva, ao se comparar com o tempo em que o Sherlock retorna os resultados. Desenvolvido em linguagem C, apresenta bom desempenho e realiza a análise de semelhança léxica entre documentos textuais, incluindo códigos-fonte, enquanto que as outras ferramentas foram concebidas para comparar documentos em linguagens de programação específicas. Para encontrar trechos duplicados de cada documento, é gerada uma assinatura digital calculando valores *hash* para palavras e sequência de palavras. Ao final, comparam-se as assinaturas geradas e identifica-se o percentual de semelhança.

O Sherlock funciona por meio da comparação de palavras separadas por

espaços. Decorrente dessa característica, por exemplo, a expressão “for(” é avaliada diferentemente da expressão “for (”, com espaço, apesar de essa diferença não ser significativa para a compilação. Logo, foi necessário desenvolver técnicas para manipular os códigos com a ideia inicial de uniformizá-los, buscando-se garantir que expressões equivalente sejam consideradas de maneira coerente.

3. Critérios e Técnicas para normalização de código

A análise de similaridade acontece em duas etapas: o pré-processamento (normalização) dos códigos e a comparação pelo Sherlock (Pike, 2012). A normalização atua principalmente na identificação e remoção de trechos de código sem relevância, além de padronizar os códigos para facilitar a comparação. Além disso, visa desconsiderar modificações inseridas para encobrir plágios. Essas modificações, em geral, não são complexas, sendo limitadas a mudanças que não alteram o funcionamento do programa.

Com a finalidade de investigar as melhores técnicas para análise de similaridade, adotaram-se quatro abordagens de normalização, com graus de intervenção crescentes. Assim, o código gerado com o uso da primeira técnica é o que mais se assemelha ao da versão original, enquanto aquele gerado pela última técnica realiza alterações mais invasivas. As quatro técnicas são descritas na Tabela 1.

Tabela 1 - Descrição das técnicas de normalização

Código Original	// Imprime resposta for (i=1;i<= n;i++){ if (i %3== 1){ printf("Resp %d", v[i]/x+z); } }
<Sem modificações>	
Normalização 1	for(i=1; i<=n; i++) { if(i % 3 ==1) { printf (" Resp %d ", v[i] / x +z); } }
Remoção de linhas e espaços vazios; Remoção de todos os comentários; Remoção das referências aos arquivos externos (bibliotecas); Inclusão (ou remoção) de espaços em branco entre expressões, declaração de variáveis e outras estruturas. Regras específicas para aproximar e afastar caracteres para normalização 1.	
Normalização 2	for(i=1; i<=n; i++) { if(i % 3 ==1) { printf(, v[i] / x +z); } }
Aplicação da normalização 1; Remoção de todos os caracteres situados entre aspas. Regras específicas para aproximar e afastar caracteres para normalização 2.	
Normalização 3	for(= i <= i ++) { if(% ==) { printf(, [] / +); } }
Aplicação da normalização 2; Remoção de todos os valores literais e variáveis. Regras específicas para aproximar e afastar caracteres para normalização 3.	
Normalização 4	(=i <=i ++) { (%==) { (, [] / +); } }
Aplicação da normalização 3; Remoção de todas as palavras reservadas. Regras específicas para aproximar e afastar caracteres para normalização 4.	

As técnicas propostas são formadas por algumas dezenas de regras especificadas a partir de análise lógica e empírica. Observando a técnica de normalização 1 aplicada ao código exemplo da Tabela 1, verificou-se, por exemplo, que palavras reservadas indicativas de funções parametrizadas e estruturas condicionais, como: “printf (“, e “if (“, são melhor comparadas quando se eliminam os espaços antes do parêntese, resultando em “printf(” e “if(”. As consequências ocorridas na comparação feita pelo Sherlock em relação a separação pelo espaço em branco de elementos de estruturas logicamente associadas requer que, na normalização 4, os caracteres especiais sejam unidos, como em “(,) ;”, ao invés de “(,) ;”, para se obter o melhor resultado.

Em alguns experimentos, observou-se empiricamente que, para expressões como “i%3==1”, obtêm-se melhores resultados quando o valor numérico está junto ao operador do lado direito, e os elementos do lado esquerdo estão separados, com a seguinte configuração: “i % 3 ==1”. Contudo, para declaração de variáveis, por exemplo, “int i=1;”, a organização que resulta em melhores resultados é “int i = 1;”.

Cada normalização representa em si uma estratégia única, que herda da anterior algumas propriedades, que podem ser diferenciadas nas regras que levam a aproximar ou afastar determinados caracteres. Por exemplo, para a normalização 3 os resultados são melhores quando “% ==” ficam separados, contudo na normalização 4 convém deixar “%==” juntos. O conjunto dessas pequenas regras produz resultados eficazes, conforme será demonstrado na próxima seção.

4. Testes e Resultados

Utilizamos duas abordagens para validar a proposta apresentada. A primeira baseia-se em códigos gerados de modo manual, e, na segunda, códigos desenvolvidos por alunos.

4.1. Análise com códigos-fonte gerados manualmente

Para verificar a análise de similaridade através de um procedimento de simulação, foram realizadas modificações manualmente em 2 códigos de teste. Eles foram escolhidos por reunirem a maioria dos conceitos presentes em um primeiro curso de graduação para a linguagem C. A partir de cada código base, foi criado um conjunto de 8 variações, seguindo os padrões de plágio mais frequentemente encontrados, adaptando-se aqueles que são elencados por Ahmadzadeh *et al.* (2011). As alterações inseridas no código foram realizadas da seguinte maneira: (I) Código base; (II) Cópia do código base; (III) Inclusão/edição de comentários; (IV) Mudança de nomes de variáveis; (V) Troca de posição de variáveis e funções; (VI) Mudança de escopo de variável; (VII) Alteração na indentação; (VIII) Inclusão de informação inútil: incluir bibliotecas, variáveis, comentários; (IX) Rearranjo de expressões; (X) Todas as alterações combinadas. As alterações realizadas não ocorrem de forma incremental, ao contrário, são originadas do mesmo código base, com exceção da última, que reúne alterações de todos os itens anteriores.

As Tabelas 2a e 2b listam todos os resultados obtido pelas ferramentas JPlag (Prechelt, 2002), MOSS (2012) e o Sherlock (Pike, 2012) após a aplicação das técnicas de normalização 1 (T1), 2 (T2), 3 (T3) e 4 (T4) propostas. A primeira representa alterações em um código base com 30 linhas e a segunda em outro com 70 linhas. Supondo que as ferramentas sejam ideais, o resultado esperado em cada célula da tabela é 100% de similaridade, já que os códigos preservam em sua totalidade, a lógica contida no programa de base, contudo verifica-se que as ferramentas apresentam limitações.

Conforme concluído por Kleiman (2007), também verificamos que a normalização é mais importante do que a própria ferramenta de comparação.

Tabela 2 - Comparação de similaridade a partir de 2 códigos base

(a)							(b)						
	JPlag	MOSS	T1	T2	T3	T4		JPlag	MOSS	T1	T2	T3	T4
I	-	-	-	-	-	-	I	-	-	-	-	-	-
II	100%	92%	100%	100%	100%	100%	II	100%	97%	100%	100%	100%	100%
III	100%	92%	100%	100%	100%	100%	III	100%	97%	100%	100%	100%	100%
IV	100%	92%	100%	50%	42%	100%	IV	100%	97%	19%	11%	36%	100%
V	50%	0%	50%	50%	50%	33%	V	62%	55%	75%	70%	83%	75%
VI	70.8%	69%	100%	100%	100%	50%	VI	70.9%	65%	100%	100%	100%	100%
VII	100%	92%	100%	100%	100%	66%	VII	100%	97%	100%	100%	81%	66%
VIII	31.1%	17%	50%	50%	11%	40%	VIII	46%	50%	75%	70%	50%	50%
IX	100%	92%	50%	50%	100%	66%	IX	100%	82%	75%	70%	76%	75%
X	34.7%	0%	33%	0%	7%	16%	X	50.3%	53%	19%	11%	30%	100%
							X(2)	33.3%	35%	20%	4%	18%	40%

Pela análise da Tabela 2a, das técnicas propostas, a T1 e T3 destacam-se com os melhores resultados. Das 9 comparações T1 apresenta resultados melhores que o MOSS em 8 casos (II, III, IV, V, VI, VII, VIII e X), e com relação ao JPlag, 7 resultados são equivalentes ou melhor (II, III, IV, V, VI, VII, VIII e X), perdendo apenas em IX. Enquanto T3 apresenta resultados mais favoráveis que o MOSS em 7 casos (II, III, V, VI, VII, IX e X) e com resultados piores em 2 (IV e VIII).

Para a Tabela 2b, na linha X, obteve-se um resultado inesperado, em que a técnica de normalização 4, aplicada a uma variação do código base que acumulava todas as alterações, apresentou um resultado muito diferente dos demais. Para realizar uma análise mais aprofundada, foi gerada nova versão para o código X, conforme a linha X(2), na qual alterações mais profundas foram realizadas e o teste repetido. Ainda dessa forma, a técnica de normalização 4 apresentou resultados ainda melhores do que do JPlag e MOSS. Comparando os desempenhos, a T4 se destaca, apresentando resultados inferiores ao MOSS e JPlag apenas em duas situações (V e IX), enquanto as demais, apresenta resultados inferiores em média em 4 dos 9 casos.

Dos dados apresentados, conclui-se que as técnicas propostas possuem comportamentos complementares, o que leva a sugerir que uma solução combinada pode apresentar resultados melhores que as duas principais ferramentas e com as vantagens citadas na seção 2.

4.2. Análise com códigos-fonte submetidos por alunos

A análise de similaridade também foi avaliada em uma turma de introdução à programação, em trabalhos desenvolvidos com a linguagem C, durante o 1º semestre de 2012. A turma é formada por 97 alunos, divididos em 4 grupos. Cada grupo realizou 4 atividades contendo 2 problemas diferentes, perfazendo, ao todo, 32 problemas.

Os resultados da análise de similaridade usando as quatro técnicas de normalização foram comparados entre si e também aos resultados das ferramentas JPlag e MOSS. A tabela 3 relaciona a quantidade de códigos que apresentam índice de similaridade maior do que 20% e 70% para cada tipo de normalização. A análise da tabela mostra que, em geral, a sensibilidade à identificação de similaridade apresenta um comportamento crescente da técnica 1 para a técnica 4. Entretanto, pode-se observar que, para alguns problemas, como o de número 8, essa característica não se confirma em relação às técnicas 2, 3 e 4, sendo, na verdade, invertidas. Isso mostra que as

técnicas podem revelar semelhanças que vão além da organização estrutural do código, fornecendo elementos de análise diferenciados para o professor.

Tabela 3 - Grupo A: comparação das 4 técnicas com JPlag e MOSS

	Norm 1		Norm 2		Norm 3		Norm 4		JPlag		MOSS	
	> 20%	> 70%	>20%	> 70%	> 20%	> 70%	> 20%	> 70%	> 20%	>70%	> 20%	>70%
Problema 1	13	2	33	1	52	0	10	1	7	4	15	3
Problema 2	5	0	22	0	19	1	4	0	2	2	2	1
Problema 3	5	0	23	1	32	4	12	10	45	38	14	4
Problema 4	4	0	12	0	30	0	30	3	47	9	15	3
Problema 5	15	0	45	4	38	3	72	10	106	34	65	6
Problema 6	13	0	59	5	57	3	101	22	115	37	62	5
Problema 7	12	3	42	4	65	4	15	6	15	6	12	3
Problema 8	16	0	120	55	71	1	10	5	9	5	6	2

Sobre as diferenças observadas entre os resultados da análise de similaridade usando-se normalizações distintas, pode-se interpretar que: (i) a normalização é necessária para que se possa obter resultados significativos com o uso do Sherlock em comparação de códigos-fonte; (ii) normalizações diferentes podem revelar características de similaridade diferentes.

As justificativas para as diferentes ocorrências entre as aplicações da normalização poderiam também levar a duas hipóteses: (i) existe pouca ocorrência de plágio e, na verdade, a normalização 4 é constituída por falsos positivos; (ii) a normalização 1 não é eficiente e a 4 é a que apresenta melhores resultados. Entretanto, para aprofundar um pouco mais essa análise, foram gerados os resultados para as mesmas questões com o uso do JPlag e do MOSS, ilustrada na tabela 3 e 4.

A tabela 4 apresenta os comparativos do número de ocorrência de similaridades para o grupo D da turma de programação, levando-se em consideração apenas as técnicas de normalização 1 e 4, além do JPlag e do MOSS. Pela baixa incidência de similaridade apontada pela técnica de normalização 1 em relação às demais, infere-se que a técnica 1 produz os resultados menos significativos. Além disso, o JPlag, praticamente em todos os problemas, indica o maior número de casos de similaridade, e, por outro lado, o MOSS sempre aponta o menor número de similaridades registradas. Isso demonstra que a sensibilidade utilizada pelo MOSS é a mais conservadora para indicação de similaridade. A técnica de normalização 4, com o uso do Sherlock, apresenta resultados intermediários entre os do MOSS e do JPlag em 62% dos problemas para os registros de similaridades acima de 70%, demonstrando, assim, constituir-se em uma promissora ferramenta de análise. Um aspecto importante da técnica 4 associada ao Sherlock é a redução expressiva do tamanho do código a ser comparado devido à eliminação de grande parte do código, o que, na maioria das vezes, permitirá o processamento mais rápido da comparação.

O uso do Sherlock com a técnica de normalização 4, em algumas situações, pode apresentar maior ocorrência de similaridade do que o JPlag, como pode ser observado pelo problema 31, que propõe a impressão por extenso do valor de um algarismo inteiro. A análise visual dos códigos dos alunos mostrou que quase todos desenvolveram uma solução com o uso do *swicth*, com código bastante semelhante, o que foi feito por orientação do professor (e poderia também ter sido consequência do uso de algum exemplo encontrado em fontes de referência). Na perspectiva da similaridade, os resultados não são, portanto, considerados falsos positivos para essa

questão. Assim, verifica-se que os resultados do Sherlock com a normalização 4 foram mais significativos do que os apresentados pelo JPlag e MOSS, característica também observada na Tabela 2.

Tabela 4 - Grupo D: relação da técnica 1 e 4 com JPlag e MOSS

	N01		N04		JPLAG		MOSS	
	>20%	>70%	> 20%	>70%	> 20%	>70%	> 20%	>70%
Probl 25	26	1	30	7	42	12	33	3
Probl 26	7	0	34	3	12	4	12	7
Probl 27	11	1	16	11	29	15	12	11
Probl 28	6	1	27	15	45	21	26	7
Probl 29	8	1	16	3	41	4	29	2
Probl 30	6	1	50	4	70	14	44	1
Probl 31	9	0	28	28	29	10	22	4
Probl 32	9	1	1	0	4	1	1	0

5. Ferramenta para análise de similaridade no Moodle

A ferramenta foi utilizada ao longo do semestre, tendo sido recolhidos os dados apresentados na seção anterior, permitindo ao professor melhor conhecer o comportamento dos alunos e fazer a avaliação da metodologia empregada. Ao final do semestre, o comportamento das duplas de alunos que apresentaram índices recorrentemente altos de similaridade foi estudado pelo professor e pelos monitores que acompanharam as práticas laboratoriais. Alguns alunos foram convidados a esclarecer os motivos da alta similaridade. Da análise, pôde-se inferir que os altos índices nem sempre indicaram o plágio, embora a facilidade de comunicação com o uso da internet tenha sido um elemento facilitador desse comportamento. Com base na análise e no acompanhamento presencial do professor e dos monitores em laboratório, pode-se verificar que algumas ocorrências de similaridade se deram entre duplas de alunos que estudam regularmente em conjunto e que, por isso, recorre usualmente ao mesmo tipo de suporte, além de compartilharem ideias e soluções, atitude que não pode ser condenada. Em relação ao plágio, alguns elementos metodológicos podem contribuir com a redução da prática, como a proposição de exercícios distintos para diferentes alunos em uma mesma aula de laboratório e apresentação de enunciados apenas no seu início, pois os problemas, em sua maioria, eram propostos com bastante antecedência. Entretanto, faz-se necessário a realização de novas experimentações para alcançar o modelo mais conveniente.

Ainda sob a perspectiva metodológica, a experiência permitiu ao professor conhecer alguns dos recursos utilizados pelo aluno para o aprendizado. Verificou-se que grande parte da turma utiliza redes sociais para compartilhar informações e, muitas vezes, soluções para as atividades propostas. Se, por um lado, a utilização de recursos virtuais pode auxiliar no processo de aprendizagem, fomentando discussões coletivas sobre os problemas e suas soluções, em alguns casos, acabou servindo de suporte aos alunos que apresentam postura mais passiva, e que buscam respostas prontas.

Todos os cenários descritos e as observações realizadas estão sendo continuamente reavaliadas, visto que a ferramenta continua sendo utilizada e gerando novos dados. Espera-se que, com a contribuição da ferramenta para análise do comportamento dos alunos seja possível ao professor melhor conhecer os seus alunos e interferir de maneira mais eficiente para um melhor aprendizado e, consequentemente, para a redução da evasão e da reprovação.

6. Considerações finais e trabalhos futuros

Neste trabalho, identificamos que a ferramenta Sherlock não é eficaz para análise de similaridade de código-fonte sem que se faça, *a priori*, uma normalização criteriosa do código a ser processado. Para lidar com essa limitação, foram propostas e desenvolvidas técnicas de normalização que, de acordo com os resultados apresentados, são comparáveis as duas principais ferramentas de detecção de plágio referenciadas na literatura. As técnicas de normalização desenvolvidas, somadas à característica *open source* do Sherlock, que permite a sua inserção como um componente de um ambiente virtual preparado para práticas laboratoriais de programação, viabilizam a sua utilização de maneira eficaz na ferramenta BOCA-LAB.

Constituindo-se em um instrumento de análise para o professor de programação, a verificação de similaridade para todas as quatro técnicas de normalização propostas pode ser de grande utilidade, visto que elas apresentam variações que afetam a sensibilidade da comparação. Apesar de a técnica 4 ser a mais sensível em termos estruturais, é possível que seja importante para o professor, por exemplo, considerar os nomes de funções ou de palavras reservadas na comparação entre códigos, o que o remete para as outras técnicas. Além disso, é importante frisar que a ferramenta pode ser utilizada para finalidades diferentes da detecção de plágio. É possível, em especial nas turmas de iniciantes em programação, que a similaridade seja até mesmo desejável. Para exemplificar, o professor pode usar a comparação para verificar se as soluções apresentadas foram baseadas em um exemplo fornecido ou em uma orientação dada aos alunos ou se foram usadas lógicas e estruturas diferentes daquelas que foram propostas.

Como trabalhos futuros, pretende-se aprimorar as regras de normalização, investigando outras abordagens e formas de interferência no algoritmo Sherlock. Além disso, novos testes serão realizados para problemas mais avançados e que utilizem mais de um código fonte. Os dados obtidos neste trabalho serão usados para cruzar se existe relação entre a alta similaridade encontrada em alguns alunos e os índices de evasão e reprovação.

Referências Bibliográficas

- Ahmadzadeh, M.; Mahmoudabadi, E.; Khodadadi F. (2011) "Pattern of Plagiarism in Novice Students' Generated Programs: An Experimental Approach" In Journal of Information Technology Education, v10.
- Ahtiainen, Aleksi; Surakka, Sami; Rahikainen, Mikko. (2006). "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises". In Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research, pages 141–142, New York, NY, USA, 2006. ACM.
- Bin-Habtoor, A. S.; Zaher, M. A. (2012) "A Survey on Plagiarism Detection Systems" International Journal of Computer Theory and Engineering vol. 4, no. 2, pp. 185-188.
- Burrows, S.; Tahaghoghi, S.M.M.; Zobel, J. (2007). "Efficient and effective plagiarism detection for large code repositories". Software-Practice & Experience, 37(2), 151-175.
- Chen, Xin; Francia, Brent; Li, Ming; Mckinnon, Brian; Seker, Amit (2004). "Shared information and program plagiarism detection," IEEE Transactions on Information Theory, vol. 50, no. 7, pp. 1545–1551, 2004.

- Dougiamas, M. & Taylor, P. (2003). "Moodle: Using Learning Communities to Create an Open Source Course Management System". In D. Lassner & C. McNaught (Eds.), *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2003* (pp. 171-178).
- Djuric, Z., Gasevic, D., "A Source Code Similarity System for Plagiarism Detection" *The Computer Journal*, 2012.
- Grune, D. and Vakgroep, M. 1989. Detecting copied submissions in computer science workshops. Tech. rep., Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit.
- França, A. B.; Soares, J. M. (2011). "Sistema de apoio a atividades de laboratório de programação via Moodle com suporte ao balanceamento de carga". In: XXII Simpósio Brasileiro de Informática na Educação, Aracaju - SE. Anais do XXII SBIE, 2011. p. 710-719.
- Hage, J.; Rademaker, P.; Vugt, N. V. "A comparison of plagiarism detection tools." Utrecht Uni-versity. Utrecht, The Netherlands, p. 28. 2010.
- Kleiman, Alan B. (2007) "Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação." Universidade Estadual de Campinas. Dissertação de Mestrado.
- Lancaster, Thomas; Culwin, Fintan. (2001). "Towards an error free plagiarism detection process". In *Proceedings of the 6th annual conference on Innovation and technology in computer science education (ITiCSE '01)*. ACM, New York, NY, USA, 57-60.
- MOSS (2012). "MOSS (Measure Of Software Similarity) plagiarism detection system". Univ. California, Berkeley. Disponível em "<http://theory.stanford.edu/~aiken/moss/>". Acesso em: 01 de agosto de 2012.
- Mota, M. P., Brito, S. R., Moreira, M. P., Favero, E. L. (2009) "Ambiente Integrado à Plataforma Moodle para Apoio ao Desenvolvimento das Habilidades Iniciais de Programação." In: XX Simpósio Brasileiro de Informática na Educação. Florianópolis: SBC.
- Onlinejudge (2012). Disponível em: <https://github.com/hit-moodle/onlinejudge>. Acesso em 01 de agosto de 2012.
- Prechelt, L.; Malpohl, G. & Philippsen, M. (2002). "Finding plagiarisms among a set of programs with JPlag". *J. UCS – Journal of Universal Computer Science*.
- Pike, R. (2012). The Sherlock Plagiarism Detector. Disponível em: "<http://sydney.edu.au/engineering/it/~scilect/sherlock/>" Acesso em: 01 de agosto de 2012.
- Schleimer, S., Wiljerson, D. S., & Aiken, A. (2003) "Winnowing: local algorithms for document fingerprinting". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA.
- VPL (2012). "Virtual Programming Lab" Disponível em: "<http://vpl.dis.ulpgc.es/>". Acesso em: 01 de agosto de 2012.



Sistema de apoio a atividades de laboratório de programação via Moodle com suporte ao balanceamento de carga e análise de similaridade de código

Title: System to support laboratory activities of programming via Moodle with support for load balancing and similarity analysis code

Allyson Bonetti França

Universidade Federal do Ceará
Departamento de Engenharia de
Teleinformática (DETI)
allysonbonetti@gmail.com

Danilo Leal Maciel

Universidade Federal do Ceará
Departamento de Engenharia de
Teleinformática (DETI)
daniloleal@alu.ufc.br

José Marques Soares

Universidade Federal do Ceará
Departamento de Engenharia de
Teleinformática (DETI)
marques@ufc.br

Resumo

Visando instrumentalizar o professor para o melhor acompanhamento de turmas numerosas de disciplinas de programação, este trabalho apresenta um ambiente que integra e adapta ferramentas Web para avaliar os programas implementados por alunos, bem como fazer inferências sobre a similaridade entre os códigos desenvolvidos. Com este sistema, alunos submetem e validam suas soluções e, adicionalmente, professores podem executar algoritmos de pré-processamento e comparação para dar suporte à análise de similaridade entre essas soluções. Todas as operações são realizadas por meio da interface do ambiente Moodle. Considerando que a compilação e execução concorrente de grande quantidade de programas podem exigir alta disponibilidade de recursos computacionais, o ambiente desenvolvido oferece suporte ao balanceamento de carga.

Palavras-Chave: *Análise de similaridade, detecção de plágio, laboratório virtual, avaliação automática de código-fonte*

Abstract

Aiming to instrumentalize the teacher to make a better monitoring of large classrooms of programming disciplines, this work presents an environment that integrates and adapts Web tools to evaluate programs implemented by students, as well as making inferences about the similarity between the codes developed. With this system, students can submit and validate their solutions and, in addition, teachers can execute preprocessing and matching algorithms to support analysis of similarity between these solutions. All operations are performed through the interface of the Moodle environment. Whereas the compilation and concurrent execution of large number of programs may require high availability of computational resources, the developed environment supports load balancing.

Keywords: *Similarity analysis, plagiarism detection, virtual laboratory, automatic assessment of source code*

1 Introdução

Disciplinas de técnicas de programação em cursos de computação e engenharia são, em geral, muito numerosas, exigindo bastante do professor e dos monitores que, muitas vezes, não conseguem realizar um acompanhamento individualizado de maneira eficiente. Isto pode provocar desestímulo, impelindo a turma, por vezes, à dispersão em aulas de laboratório, situação dificilmente controlável pelo professor. Conforme constata-se por Essi [1] e Tan [2], ainda é um desafio obter um bom rendimento em tais turmas, que, de modo geral, possuem altos índices de evasão e de reprovação.

Professores podem fazer uso de ferramentas de suporte para o seu trabalho. Aplicações Web, frequentemente, são usadas para disponibilização de notas de aula, proposição e submissão de trabalhos e registro de notas. Entretanto, embora o uso de tais ferramentas possa mitigar os problemas de natureza organizacional em práticas laboratoriais, não são suficientes para solucionar a dificuldade de acompanhamento e orientação aos alunos.

Para ilustrar situações comuns em laboratórios de programação, cita-se uma questão frequentemente colocada por alunos:

Professor, o meu programa está correto?

Para responder a esta questão, é necessário que o professor se desloque até o aluno no laboratório, observe a execução do programa, verifique o seu resultado e, eventualmente, analise o código desenvolvido pelo aluno. Em caso de erro, muitos alunos assumem posturas passivas e aguardam que o professor o descubra. Em uma turma de 60 alunos, por exemplo, essa atividade de simples verificação pode tornar o tempo de aula insuficiente.

Uma maneira de reduzir significativamente esse trabalho é permitir que o próprio aluno valide o resultado de seu programa em um procedimento semelhante ao realizado em maratonas de programação. Nesta perspectiva, visando contribuir com as condições de ensino e aprendizagem de cursos de programação, é apresentado neste trabalho um ambiente que permite a automatização de avaliações de programas propostos pelo professor para desenvolvimento nas linguagens de programação C, C++ e Java. O objetivo é, por um lado, fornecer ao professor uma ferramenta que permita o gerenciamento de seus recursos didáticos e que lhe dê apoio ao acompanhamento das práticas laboratoriais. Por outro lado, objetiva-se permitir ao aluno um *feedback* mais rápido, que o incentive a um comportamento mais autônomo.

Adicionalmente, é definido um modelo de integração de ferramentas, que é voltado especificamente para a

avaliação de programas, aos chamados ambientes virtuais de aprendizagem (AVA). Os recursos e funcionalidades das ferramentas integradas são oferecidos aos usuários de forma complementar, através de uma interface única e coesa. Para a composição e avaliação do modelo de integração, adotou-se uma metodologia que se baseia no conceito de arquitetura orientada a serviços (*Service Oriented Architecture* – SOA) [3].

O ambiente desenvolvido para apoio a laboratórios de programação foi concebido como extensão do sistema BOCA [4]. Desenvolvido na Universidade de São Paulo (USP) para dar suporte ao julgamento de trabalhos desenvolvidos em maratonas de programação, permite submissão e avaliação automática de soluções para os problemas apresentados aos concorrentes, o BOCA precisou ser adaptado para atender a necessidades específicas ao trabalho em laboratórios de programação. Para viabilizar a sua integração aos demais recursos do ambiente de integração, foi desenvolvida uma camada de software para a exposição de suas funcionalidades em forma de serviços. Além disso, foi desenvolvido uma infraestrutura para dar suporte ao balanceamento de carga, visto que a solução para alguns problemas propostos podem apresentar carga computacional considerável para execução em um único servidor, levando-se em conta a complexidade da solução, o número de alunos e a quantidade de turmas com trabalhos concorrentes. O sistema estendido é denominado neste trabalho BOCA-LAB.

Com a finalidade de monitorar as submissões semelhantes, o sistema integrado oferece também ao professor uma ferramenta específica para oferecer suporte à análise de similaridade entre códigos de alunos, pois um problema não raramente encontrado em laboratórios de programação é a cópia total ou parcial de soluções entre colegas, frequentemente com a mudança de nomes de variáveis ou com a inserção de comentários para tornar mais difícil a percepção da ação. Em cenários de turmas numerosas, a detecção deste tipo de conduta se torna bastante difícil.

Embora a detecção de plágio seja um aspecto importante na condução de uma turma de programação, a experiência trazida com a utilização da ferramenta de análise de similaridade revelou que os resultados da comparação entre códigos podem indicar outros aspectos que são de natureza irrepreensível. Dependendo da perspectiva e do contexto, a semelhança pode ser indicativa de parceria, de trabalho colaborativo, de referência a uma solução encontrada em livros ou exemplos fornecidos pelo professor, entre outros motivos. Por essa razão, ao invés de “detecção de plágio”, adota-se neste trabalho a expressão “análise de similaridade”.

A detecção de plágio (e a análise de similaridade) em código-fonte vem sendo estudada em diversos trabalhos [5] [6] [7] e algumas ferramentas foram disponibilizadas para este fim [8] [9]. No ambiente de integração proposto neste trabalho, avalia-se especialmente a sensibilidade do algoritmo Sherlock [10] quando os códigos são submetidos a técnicas de pré-processamento (denominadas “normalização”) que ressaltam características específicas dos códigos antes da comparação. O objetivo é remover aspectos irrelevantes, como comentários, valores literais ou nomes de variáveis, destacando os aspectos estruturais do código. Os resultados apresentados com a aplicação das técnicas de normalização e uso do algoritmo Sherlock no ambiente integrado são confrontados com aqueles obtidos com o uso do JPlag [8] e Moss [9], ferramentas destinadas ao controle de plágio e amplamente discutidas [26].

O BOCA-LAB, nome dado ao ambiente integrado, foi implantado no Moodle [11], que forneceu a interface e o conjunto de funcionalidades necessárias à gestão e ao acompanhamento das atividades associadas ao laboratório de programação. A integração se apóia no uso de *Web Services* (WS), que se destacam como tecnologia para a implementação de SOA e vêm sendo utilizados em sistemas educacionais como o Sakai [12].

O texto está disposto da seguinte forma: a seção 2 aborda os trabalhos relacionados, apresentando soluções que buscam essa automatização na correção e avaliação de códigos fontes em sistemas de cunho educacional; a seção 3 mostra as características do Moodle, do BOCA [4] e do Sherlock [10], que são as ferramentas de base que compõem o ambiente de integração; na seção 4 é mostrada a arquitetura de integração. A interação entre os usuários e a arquitetura é explicada na seção 5. Na seção 6 discute-se a experiência de utilização do ambiente e apresenta-se os testes realizados, por último, são apresentadas as conclusões e perspectivas do trabalho.

2 Trabalhos Relacionados

2.1 Ambientes Virtuais

O uso de ambientes virtuais para dar suporte a atividades de programação vem sendo estudado há alguns anos.

Ng [13] investiga como a nova tecnologia pode auxiliar no processo de ensino e aprendizagem em disciplinas de programação, propondo um ambiente Web interativo para o ensino de linguagens de programação Java. O ambiente apresenta funcionalidades que permitem a compilação e o retorno de erros dos programas submetidos.

Wang [14], propõe um sistema Web para o ensino da linguagem de programação C. Esse sistema é desenvolvi-

do em .NET e oferece funcionalidades que permitem a compilação e checagem de erros dos programas submetidos.

O Sistema Susy [15] é utilizado pelos alunos do Instituto de Computação, na Unicamp, para a submissão e o teste automático das atividades submetidas. Não apresenta integração com AVAs e possui código fechado.

Pode-se citar ainda, as ferramentas BOSS [16], *Programming Assignment aSessment System* (PASS) [17] e Pratomat [18], que também são ferramentas para submissão e avaliação remota de código. As características e limitações específicas de cada ferramenta referenciada são detalhadas na Tabela 1.

Embora os trabalhos correlatos apresentem plataformas interativas para o ensino de linguagem de programação, eles não oferecem, em um ambiente integrado, outros recursos de apoio ao processo de ensino e aprendizagem, como ferramentas de discussão síncronas e assíncronas, suporte a gestão de conteúdo, entre outros recursos importantes, principalmente para disciplinas que possuam algum suporte a distância. Outro limite de algumas das plataformas citadas é restringir o suporte a apenas um tipo de linguagem de programação.

Em um contexto mais aproximado ao trabalho aqui apresentado, algumas iniciativas foram realizadas no sentido de integrar recursos de apoio a disciplinas de programação ao ambiente Moodle, como o VPL [19] e o Onlinejudge [20]. O VPL (*Virtual Programming Lab*) é uma ferramenta de código aberto que permite o desenvolvimento remoto de programas através de um módulo acoplado ao Moodle. A edição do código é feita através de um *applet* e a compilação e execução do código é realizada com segurança em um servidor Linux remoto. É possível efetuar a compilação em várias linguagens de programação, dentre elas C, C++, PHP, Java e Fortran. Para a correção e compilação de códigos fonte, este módulo necessita, a cada atividade cadastrada pelo professor, da configuração de como serão os processos de compilação de códigos fonte e de correção automática. A arquitetura utilizada pelo VPL não permite a adição de novas ferramentas ou o balanceamento de carga, visto que o servidor responsável pela compilação e execução do código submetido é único. A centralização deste aspecto pode se tornar um gargalo uma vez que podemos ter, em um mesmo servidor Moodle, várias turmas contendo dezenas de alunos submetendo soluções simultaneamente.

O Onlinejudge [20], também desenvolvido para gerenciar a submissão de códigos fontes adicionado ao Moodle, pode ser integrado com o uso de WS a uma aplicação denominada Ideone [21]. Essa aplicação permite escrever códigos fonte em aproximadamente 40 lin-

guagens de programação diferentes, sendo os mesmos executados diretamente a partir do navegador. O Onlinejudge também pode ser executado sem a integração com o Ideone, dando suporte, nesse caso, apenas às linguagens C e C++. Entretanto, como se trata de uma aplicação comercial e de código fechado, o modelo de integração permite a submissão de apenas 1000 códigos fonte por mês em uma conta gratuita e não aceita a submissão de vários códigos fonte por vez.

Apesar de todos os trabalhos citados conterem importantes contribuições para o apoio a práticas laboratoriais em turmas de programação, neste trabalho propõe-se um

ambiente de auxílio a compilação e execução remota de programas que seja capaz de reunir as seguintes características: (i) ser integrado a um ambiente virtual de aprendizagem, permitindo o seu uso e o acompanhamento de resultados através da mesma interface de outras ferramentas disponíveis no ambiente virtual; (ii) dar suporte ao uso de diversas linguagens de programação; (iii) permitir a gestão de múltiplos servidores e executar o balanceamento de carga entre os servidores disponíveis; (iv) permitir análise de similaridade entre os códigos enviados, permitindo a relação entre as soluções de alunos ou, dependendo da situação, auxiliando na identificação de plágios.

Funcionalidade	BOCA-LAB	Sistema Susy	BOSS	PASS	Praktomat	VPL	Onlinejudge + Ideone
Submissão de Atividades	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Suporte para integração com AVAs	Sim	Não é possível determinar	Não	Não	Não	Sim	Sim
Deteção de Similaridade	Sim	Não é possível determinar	Sim	Sim	Não	Sim	Não
Escalabilidade (Balanceamento de Carga)	Sim	Não é possível determinar	Não	Não	Não	Não	Não
Código aberto	Sim	Não	Sim	Não	Sim	Sim	Não
Suporta ao menos 2 linguagens	Sim	Sim	Sim	Sim	Não	Sim	Sim
Análise estatística	Não	Não é possível determinar	Não	Sim	Não	Não	Não

Tabela 1: Comparação dos trabalhos relacionados

2.2 Deteção de Plágio

Diversas ferramentas podem ser utilizadas para realizar a análise de similaridade entre códigos-fonte, tais como SIM [22], YAP [23], JPlag [8], SID [24], Plaggie [25] e Moss [9]. Estudos comparativos sobre algumas destas ferramentas podem ser encontrados em [5]. De modo geral, ferramentas especificamente construídas para esta finalidade permitem encontrar semelhanças no caso de alterações de nome de variáveis, nome de funções, comentários ou, ainda, alterando-se a ordem de partes do código. Segundo Burrows [26], JPlag e MOSS são algumas das mais importantes e mais citadas para deteção de plágio, que são sucintamente descritos nos próximos parágrafos, seguidos pelo Sherlock [10], sobre o qual se apoiam as contribuições deste trabalho para a análise de similaridade.

O JPlag [8] é uma ferramenta desenvolvida em Java disponibilizada exclusivamente por meio de um Webservice, possuindo código fechado. Ela requer o cadastra-

mento e a requisição de autorização para o acesso ao serviço. O envio de arquivos para comparação é realizado através de um *applet* que retorna, em formato HTML, o resultado do processamento. A semelhança entre códigos é apresentada apenas para os com maior percentual de similaridade, não sendo, portanto, disponibilizada a visualização de pares com porcentagens pequenas.

O MOSS (*Measure of Software Similarity*) [9] também é acessado por meio de um Webservice, disponibilizado na Universidade de Califórnia. Para usá-lo, é necessário realizar um cadastramento por *e-mail*. Os arquivos são enviados para o servidor por meio de um *script* fornecido pelo seu desenvolvedor. Ao final do envio, é gerada uma URL que identifica a página que apresenta o resultado da comparação. Segundo informações obtidas na página da aplicação, o desenvolvimento do MOSS iniciou-se em 1994, apresentando uma melhora significativa de outros algoritmos, que não são mencionados, para deteção de plágio. O MOSS, que também é de código fechado, é baseado no algoritmo Winnowing [27].

O Sherlock [10] representa uma alternativa às ferramentas apresentadas precedentemente por ser de código aberto, permitindo modificações, melhorias e integração a outros ambientes, permitindo que o BOCA-LAB tenha o controle total de seus recursos sem exigir conectividade a Web Services de terceiros. O Sherlock [10] realiza a análise de semelhança léxica entre documentos textuais, inclusive para códigos-fonte. Para encontrar trechos duplicados de cada documento, é gerada uma assinatura digital que calcula valores *hash* para palavras e sequência de palavras. Ao final, comparam-se as assinaturas geradas e identifica-se o percentual de semelhança.

Além das características citadas, o Sherlock [10] possui um bom desempenho por ser desenvolvido em C. Outra vantagem é ser de propósito geral, enquanto todas as outras ferramentas são configuradas para a comparação em linguagens de programação específicas.

3 Ferramentas de Base: Moodle, BOCA e Sherlock

3.1 O Moodle como interface do ambiente de integração

O Moodle (*Modular Object Oriented Distance Learning*) é um sistema de código aberto baseado na Pedagogia Social Construcionista [28]. Rico em recursos educacionais, oferece alta flexibilidade para configuração e uso. Seu desenvolvimento modular permite a fácil inclusão de novos recursos que podem melhor adaptá-lo às necessidades da instituição que o utiliza. Por ser um ambiente extensível e completo em termos de recursos para gerenciamento de atividades educacionais, o Moodle apresenta-se como ambiente propício para integrar ferramentas que dêem suporte ao processo de ensino e aprendizagem em disciplinas de programação.

3.2 O BOCA como recurso para Compilação e Verificação de resultados para Problemas de Programação

O BOCA [4] é um sistema de apoio a competições de programação desenvolvido para uso em maratonas promovidas pela Sociedade Brasileira de Computação. Oferece suporte *online* durante a competição, gerenciando times de alunos e juizes, permitindo a proposição de problemas de programação bem como a submissão e avaliação automática de soluções. Sendo um sistema de código aberto, o BOCA pode ser adaptado ao contexto de laboratórios de programação e integrado a um AVA, como o ambiente Moodle. As características de principal interesse para a integração do BOCA ao Moodle são

apresentadas nos próximos parágrafos.

Para cada problema cadastrado no BOCA, são necessários um arquivo contendo um conjunto de entradas e outro contendo as respectivas saídas. Os arquivos de entrada e saída são obtidos pelo professor, através de um programa executável elaborado pelo mesmo como solução ao problema, onde as entradas enviadas para o programa e as saídas geradas são armazenadas em arquivos distintos.

Ao receber o código fonte submetido por um time, o sistema o compila. Caso não ocorra nenhum erro, é realizada a sua execução. O teste do programa é realizado com o processamento da entrada cadastrada para o problema. Em seguida, o sistema efetua a comparação da saída gerada pela solução do time com aquela cadastrada para o problema. Ao final das etapas de compilação e comparação, é enviado um *feedback* para o time, contendo eventuais erros encontrados no processo de compilação ou na comparação da saída.

Para dar suporte à integração das funcionalidades dos dois ambientes, o sistema de armazenamento de dados, a submissão de arquivos e a compilação realizada pelo BOCA precisaram ser adaptados.

Em sua concepção original, o sistema BOCA [4] só permite o envio de um único arquivo por problema computacional proposto.

O envio de mais de um programa fonte pode ser facilmente resolvido através da compactação do conjunto de arquivos usando ferramentas como ARJ ou ZIP. Entretanto, essa operação resolve apenas parcialmente o problema, tendo em vista que é necessário o servidor identificar o arquivo compactado, executar a descompactação, a compilação dos programas fontes e o armazenamento de maneira adequada dos mesmos.

Para a aplicação visada neste trabalho, os problemas devem ser propostos de forma individual, sendo necessário, portanto, adaptar o BOCA para armazenar informações de forma a identificar o aluno no Moodle, rastrear as atividades do mesmo e fornecer *feedbacks*.

Para a gestão do cadastro de alunos, registro de atividades e notas, entre outros aspectos administrativos, o ambiente Moodle oferece os recursos necessários. Assim, verifica-se a complementaridade entre os ambientes a serem integrados neste trabalho, valorizando o conjunto de competências peculiares a cada um. Além das alterações propostas para o BOCA [4], um módulo de extensão deve ser criado no Moodle de maneira a permitir a integração entre os ambientes. Este módulo de extensão deve: (i) permitir o acesso à funcionalidades disponibilizadas pelo BOCA [4]; (ii) usar estruturas específicas para regis-

tro dos dados relativos aos problemas propostos; (iii) apresentar interfaces para submissão de soluções ao BOCA e para apresentação dos resultados, ambos a partir da interface do Moodle.

3.2 O Sherlock para análise de similaridade

O Sherlock [10] encontra semelhanças entre documentos textuais, através de assinaturas digitais, e apresenta os resultados das análises em tempo real. O mesmo apresenta dois modos de operação. No primeiro ele pode descobrir plágio em tarefas de linguagem natural e, no outro, ele pode descobrir plágio em tarefas de código fonte.

Para verificar a semelhança, o software Sherlock [10] analisa certa quantidade de palavras para cada linha do texto e gera uma assinatura digital que identifica essas palavras. Nesse processo, as linhas e espaços múltiplos em branco ou comentários de código são ignorados. Contudo, alguns espaços em brancos são fundamentais para determinar o início e o fim de uma palavra. Por exemplo, a expressão “for(” é avaliada diferentemente da expressão “for (”, com espaço, apesar de essa diferença não ser significativa para a compilação.

O procedimento de geração da assinatura digital é repetido até o final de cada documento comparado. Ao terminar essa etapa, o Sherlock [10] possuirá as assinaturas digitais que identificam todo o texto. Finalmente, para determinar a semelhança entre os dois arquivos, compara-se as assinaturas digitais dos textos e retorna a porcentagem de semelhança entre eles.

Outra função importante sobre o funcionamento do Sherlock [10] está na quantidade de comparações a serem realizadas para certa quantidade de textos. Uma vez que o *software* compara os arquivos em pares e todos os arquivos devem ser comparados entre si, a quantidade de comparações a ser realizada será dada pela equação 1, em que m é a quantidade de códigos a serem comparados. Desta forma, nota-se que é indesejável comparar um arquivo A com um B se B já foi comparado com A.

$$C\left(\begin{matrix} m \\ 2 \end{matrix}\right) = \frac{m!}{2!(m-2)!} \quad (1)$$

O percentual de semelhança é, na integração com o BOCA-LAB, calculado em função da técnicas de normalização utilizada. Essas técnicas são apresentadas na próxima seção, juntamente com a arquitetura de integração.

4 Arquitetura de Integração

A arquitetura da integração é composta por três módulos que se comunicam através do protocolo SOAP usando

mensagens criptografadas no formato XML, e um módulo responsável pela análise de similaridade.

A Figura 1 ilustra a estrutura de comunicação destes módulos, ressaltando a coexistência de múltiplos servidores que dão suporte ao balanceamento de carga. Os módulos são detalhados nas próximas subseções.

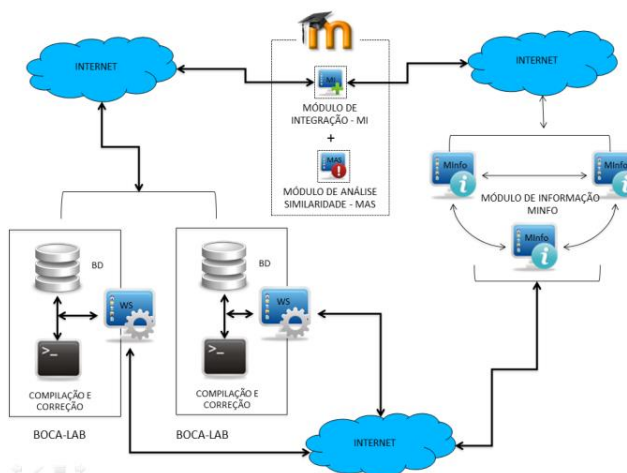


Figura 1: Arquitetura da integração

4.1 Módulo de Integração (MI)

O MI é responsável pelo acesso ao serviço de busca de servidores, registro dos dados necessários aos problemas computacionais e pelo envio e recuperação de *feedback* dos códigos fonte submetidos aos servidores MAB.

4.2 Módulo de Informação (MInfo)

O MInfo é o módulo responsável pela disponibilização dos serviços de localização e registro do estado dos servidores MAB. O armazenamento de informações sobre o estado dos servidores MAB permite efetuar o balanceamento de carga.

4.2.1 Controle e Balanceamento de Carga

O balanceamento de carga na arquitetura é realizado pelo MInfo, a cada requisição feita pelo MI, o módulo verifica dentre os servidores MAB aquele que retém menor número de submissões, visando minimizar o tempo de resposta e evitar sobrecarga.

Além de reduzir o impacto da concorrência por recursos computacionais no mesmo servidor para compilação e execução de problemas de programação, o balanceamento de carga agiliza o processo de *feedback* para o aluno, evitando que um processo permaneça tempo desnecessário nas filas de *jobs* em servidores sobrecarregados.

4.3 Módulo de Acoplamento BOCA-LAB (MAB)

O MAB é responsável pela disponibilização de serviços que permitem o recebimento e repasse ao BOCA-LAB dos dados relativos a problemas e códigos fonte. Além disso, realiza a recuperação de *feedback* e o controle secundário da carga de compilação no BOCA-LAB, evitando o recebimento de requisições caso o servidor esteja com sua carga máxima.

4.4 Módulo de Análise de Similaridade (MAS)

A análise de similaridade acontece em duas etapas: (i) pré-processamento (normalização) dos códigos; (ii) comparação pelo Sherlock [10]. A normalização atua principalmente na identificação e remoção de trechos de código sem relevância, além de padronizar os códigos de modo a otimizar os resultados obtidos com o Sherlock [10].

Assim, os critérios de normalização foram adotados para eliminar a falta de correspondência irrelevante entre os códigos, incluindo eventuais modificações inseridas para encobrir plágios. Essas modificações, em geral, não são complexas, sendo limitadas a mudanças que não alteram o funcionamento do programa.

Logo, com a finalidade de encontrar a melhor técnica de normalização, adotaram-se quatro abordagens diferentes de normalização, com graus de intervenção crescentes. Assim, a versão do código que é gerado com o uso da primeira técnica é a que está mais próxima da versão enviada pelo aluno, enquanto aquele gerado pela última

técnica realiza alterações mais invasivas. As quatro técnicas são descritas na Tabela 2.

As técnicas propostas são formadas por algumas dezenas de regras, especificadas a partir de análise lógica e empírica. Observando a técnica de normalização 1 aplicada ao código exemplo da Tabela 2, verificou-se, por exemplo, que palavras reservadas indicativas de funções parametrizadas e estruturas condicionais, como: “printf (“, e “if (“, são melhor comparadas quando se eliminam os espaços antes do parêntese, resultando em “printf(” e “if(”.

Da mesma maneira, devido a natureza da comparação feita pelo Sherlock [10] em relação a separação de palavras por espaço em branco, é necessário que, na normalização 4, os caracteres especiais sejam unidos, como em “(,);”, ao invés de “(,);”, para se obter o melhor resultado.

Em alguns experimentos, observou-se empiricamente que, para expressões como “i%3==1”, obtêm-se melhores resultados quando o valor numérico está junto ao operador do lado direito, e os elementos do lado esquerdo estão separados, com a seguinte configuração: “i % 3 ==1”. Contudo, para declaração de variáveis, por exemplo, “int i=1;”, a organização que resulta em melhores resultados é “int i = 1;”.

As quatro técnicas de normalização podem auxiliar a análise de similaridade sob óticas distintas, dependendo dos objetivos de quem a realiza. Alguns resultados relacionados ao uso desta ferramenta são apresentados na seção 6.2. A Figura 2 ilustra o resultado da análise de similaridade.

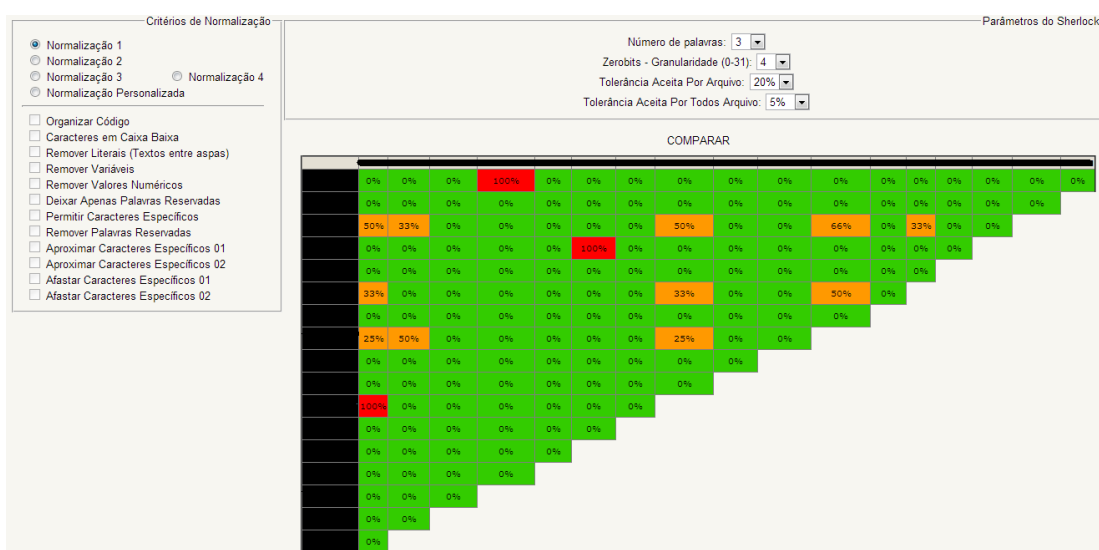


Figura 2: Página com resultado da análise de similaridade

Código Original	// Imprime resposta for (i=1;i<= n;i++){
<Sem modificações>	<pre> if (i %3== 1){ printf("Resp %d", v[i]/x+z); } } </pre>
Normalização 1	for(i=1; i<=n; i++) {
Remoção de linhas e espaços vazios; Remoção de todos os comentários; Remoção das referências aos arquivos externos (bibliotecas); Inclusão (ou remoção) de espaços em branco entre expressões, declaração de variáveis e outras estruturas.	<pre> if(i % 3 ==1) { printf (" Resp %d ", v[i] / x +z); } } </pre>
Normalização 2	for(i=1; i<=n; i++) {
Aplicação da normalização 1; Remoção de todos os caracteres situados entre aspas.	<pre> if(i % 3 ==1) { printf(, v[i] / x +z); } } </pre>
Normalização 3	for(= ; <= ; ++) {
Aplicação da normalização 2; Remoção de todos os valores literais e variáveis.	<pre> if(% ==) { printf(, [] / +); } } </pre>
Normalização 4	(=; <=; ++) {
Aplicação da normalização 3; Remoção de todas as palavras reservadas.	<pre> (%==) { (, [] / +); } } </pre>

Tabela 2: Descrição das técnicas de normalização



5 O Módulo de Integração implantado no Moodle


A aplicação desenvolvida nesse trabalho foi integrada ao Moodle, permitindo o seu uso por meio das mesmas


semântica e interface definidas para os demais recursos do ambiente virtual. Por exemplo, em seu curso, o professor deve adicionar uma atividade denominada “Envio de arquivos para compilação” que foi agregada ao Moodle para a administração da submissão de problemas, códigos fonte e recuperação de *feedbacks* por parte dos alunos. A


cadastrar seu problema através do formulário ilustrado na Figura 3.


- **Submeter Problema**

* Linguagem:  Java 

* Classe base: 

Arquivo de descrição:  Choose File no file selected

* Arquivo de entrada:  Choose File no file selected

* Arquivo de saída:  Choose File no file selected


* Validade para o problema no servidor:  23 / 08 / 2011 20:10

Figura 3: Formulário de cadastro de um problema

Submeter código fonte

Enunciado do problema: [Pratica Extra.pdf](#)

Linguagem de programação: C

O nome do arquivo contendo o código com a solução deve se chamar **imc.c**

Submeta aqui o seu código fonte: no file selected ou, se preferir edite a sua solução utilizando o [Editor de Códigos](#)

Figura 4: Formulário de envio de código fonte

Administrador Usuário
quarta, 29 junho 2011, 15:15

Nota : 100 / 100 :
Média final: 100,00

Trebuchet 3 (12 p0) Língua

Parabéns ! Seu programa foi compilado com sucesso :)

Caminho: body

☒ Enviar notificação via email

Salvar mudanças Cancelar Salvar e mostrar o próximo Próximo

aluno
terça, 28 junho 2011, 23:01 (53 minutos 54 segundos antepiado)

— 1* Feedback —

Resposta da Compilação: Resposta correta
Resposta da Comparação: Files match exactly

imc.c [Download](#) [Visualizar](#)
stderr.txt [Download](#) [Visualizar](#)
stdout.txt [Download](#) [Visualizar](#)

Figura 5: Interface de atribuição de notas

blema até ser disponibilizado o *feedback* para a última submissão. As informações retornadas ao aluno pelo BOCA-LAB são compostas por uma resposta do compilador, um arquivo contendo os erros da compilação, caso ocorram, e um outro contendo a saída gerada pelo programa. Os tipos de resposta retornados pelo compilador são mostrados pela Tabela 3.

Reposta	Descrição
YES	Programa foi aceito sem erros.
NO: Incorrect Output	Saída dos testes incorreta.
NO: Time-limit Exceeded	O programa excedeu o tempo estipulado.
NO: Runtime Error	Durante o teste ocorreu um erro de execução.
NO: Compilation Error	Programa possui erros de sintaxe.

Tabela 3: Tipo de *feedbacks* retornados pelo BOCA-LAB

Os resultados de todas as submissões são armazenados pelo sistema e apresentados na interface do professor, como apresentado na Figura 5, permitindo ao mesmo analisar o desempenho do aluno, facilitando assim, a atribuição da nota.

A nota atribuída às atividades de programação figuram junto ao conjunto de notas de atividades regulares de um curso Moodle, como Fóruns, Chats e outras atividades, compondo, assim, a nota final do aluno.

6 Discussões e Testes

A avaliação e os testes do ambiente de integração são apresentados nas próximas subseções seguintes.

6.1 Discussões do BOCA-LAB

O ambiente de integração desenvolvido foi, inicialmente, implantado em um servidor do Departamento de Engenharia de Teleinformática (DETI) da Universidade Federal do Ceará (UFC). Experimentações iniciais foram realizadas no ano de 2011 em turmas da disciplina de Técnicas de Programação para Engenharia I, do curso de Engenharia de Teleinformática, e de Fundamentos de Programação, do Departamento de Computação. Foi avaliada a percepção dos alunos sobre o ambiente e os resultados podem ser vistos em [29].

O ambiente encontra-se atualmente instalado em um servidor mantido em colaboração entre o DETI e a UFC Virtual, e vem sendo usado, com acompanhamento da equipe responsável por esta pesquisa, em turmas numerosas de primeiro ano do curso de Engenharia de Teleinformática desde 2012. As experiências de alunos e professores com a ferramenta tem permitindo o amadurecimento de aspectos relacionados à usabilidade e à interface. Essa experiência é importante para realizar os ajustes necessários e garantir a confiabilidade antes que a ferramenta seja disponibilizada para a comunidade em vários formatos: integração com Moodle, versão *standalone* do BOCA-LAB e módulo independente para a Análise de Similaridade.

Uma dificuldade que se apresentou para o uso sistemático do BOCA-LAB nas práticas de laboratório semanal é a necessidade de elaboração de exercícios com as restrições impostas pelo sistema BOCA, em que um programa deve ser avaliado para uma entrada e uma saída padronizadas. Isso requer que o professor não só elabore o enunciado, mas prepare os arquivos de entrada e de saída que serão usados como insumo para a validação das soluções dos alunos. Além disso, para testar diversas condições de um problema, é preciso preparar diversas pares de arquivos de entrada e saída. A fim de certificar-se que as entradas e as saídas estão absolutamente corretas, faz-se necessário, nessa abordagem, que o professor

desenvolva também a solução para o problema. Essa situação não ocorre usualmente em abordagens tradicionais com a proposição de listas de problemas em turmas de programação, em que o professor, pela sua própria experiência, precisa inferir apenas o nível de dificuldade de uma questão antes de apresentá-la aos alunos.

Pelo fato de a validação da saída ser efetuada por comparação de *strings* (comando *diff*), outra dificuldade se revela: o sistema é muito sensível a pequenas alterações de formatação e apresentação de saídas (ex.: uso de diferentes textos como *prompt* de entrada, uso de caracteres especiais como separadores de valores de saída, etc.). Isso requer uma elaboração de enunciado muito detalhada e precisa por parte do professor, e atenção especial por parte do aluno no desenvolvimento de sua solução. Essas restrições acrescentam dificuldades que não são diretamente associadas à natureza do problema. Em uma maratona de programação, esse tipo de restrição não representa obstáculos aos usuários, em geral, programadores mais experientes. Entretanto, para um público de iniciantes, restrições dessa natureza exigem um acultramento que apenas se constrói em múltiplas práticas.

Do ponto de vista do professor, as limitações discutidas, entretanto, podem ser mitigadas com o desenvolvimento de algumas rotinas de apoio e melhorias na interface. Requisitos associados a este tipo de limitação vem sendo elencados, discutidos e estão servindo de insumo para a melhoria da ferramenta. O uso repetido da ferramenta vem permitindo, também, que se forme uma base de exercícios e enunciados que atendem aos requisitos do ambiente desenvolvido, o que irá facilitar, sobremaneira, a utilização em diferentes turmas ao longo do tempo. É importante salientar que, se o BOCA-LAB, por um lado, requer um tempo maior para elaboração e preparação de atividades, por outro lado, vem demonstrando grande vantagem para validação de trabalhos propostos aos alunos em aulas de laboratório, otimizando o tempo de professores e monitores. Além disso, o sistema vem sendo utilizado com eficácia para a proposição de atividades em forma de prova, dispensando o tempo usualmente empregado para correção e atribuição de notas, situação crítica para o professor em turmas numerosas.

Pela perspectiva do aluno iniciante, uma dificuldade de cunho operacional se manifestou logo nas primeiras atividades. O desenvolvimento da prática de laboratório não é feita diretamente no BOCA-LAB. A metodologia empregada é a de utilização do compilador e testes na máquina local e submissão do programa fonte ao BOCA-LAB apenas após o desenvolvimento supostamente correto por parte do aluno. Percebeu-se que, sistematicamente, o aluno submetia o arquivo errado para validação remota. Esse problema foi mitigado com a submissão do código através de um editor embutido na atividade criada

no Moodle, em que o aluno pode “colar” o seu código-fonte e, em seguida, realizar a submissão. Outra dificuldade inicial é a interpretação das mensagens de erro apresentadas pelo sistema, principalmente quando a mensagem indica um problema de apresentação, o que requer as adequações de formatação já discutidas anteriormente. Apesar do tempo necessário para ambientação com esse tipo de ferramenta, percebe-se o desenvolvimento progressivo de maior autonomia do aluno para o desenvolvimento das soluções aos problemas propostos, em que ele se torna capaz de reagir às mensagens da ferramenta e corrigir os problemas sem a necessidade contínua de interação com professores e monitores.

A subseção seguinte apresenta os resultados de testes relativos ao balanceamento de carga.

6.2 Testes sobre o Controle e Balanceamento de Carga

Para avaliar a eficácia do balanceamento de carga, dois cenários de teste, ambos com 3 Servidores BOCA-LAB foram configurados. No cenário I, o Servidor I foi configurado com o compilador C e C++, o Servidor II foi configurado com os compiladores C e Java e o Servidor III foi configurado apenas com o compilador C++. Nesse cenário, todos os servidores foram limitados para o processamento de até 50 *jobs* simultâneos.

Para o primeiro teste, foram submetidos 80 códigos fonte divididos da seguinte maneira: 70 códigos em C e 10 códigos em Java.

Ao final do primeiro teste, a divisão dos códigos entre os servidores se deu da seguinte maneira: 33 *jobs* processados no Servidor I, 47 *jobs* processados no servidor II e 0 no servidor

No cenário de teste I, portanto, temos que: o servidor I recebeu 33 códigos da linguagem C; dos 47 códigos recebidos pelo servidor II, 37 foram da linguagem C e 10 da linguagem Java e; como esperado, o servidor III não recebeu nenhum código fonte, pois somente apresentava o compilador para a linguagem C++.

No segundo cenário, os servidores I e II foram configurados de maneira idêntica, contendo compiladores C e C++ e com capacidade de processar até 50 *jobs* simultâneos. O Servidor III foi configurado com apenas o compilador Java e com o limite de 25 *jobs* simultâneos.

Para esse segundo cenário, foram submetidos simultaneamente 90 códigos, divididos da seguinte maneira: 45 códigos em C, 25 códigos em Java e 20 em C++.

Conforme especificado, a distribuição dos códigos pelos servidores ocorreu da seguinte maneira: os Servidores

I e II receberam todos os 65 *jobs* relativos aos programas em C e C++, sendo distribuídos em número de 30 para o primeiro e 35 para o segundo. Já o Servidor III, único que disponibiliza o compilador Java, recebeu todos os 25 *jobs* nessa linguagem.

Dos códigos recebidos pelo servidor I, 17 foram em linguagem C e 13 em C++. No servidor II, 28 códigos em linguagem C e 7 em C++ foram recebidos. No servidor III, único que possui compilador na linguagem Java, os 25 códigos nessa linguagem foram recebidos.

Através desses dois testes em cenários diferentes, foi possível verificar o funcionamento adequado da distribuição e do balanceamento de carga da arquitetura, em função da capacidade configurada e dos compiladores disponíveis em cada servidor, bem como a capacidade de endereçamento executada pelo módulo Minfo.

6.3 Avaliação do Módulo de Análise de Similaridade

A análise de similaridade foi avaliada em uma turma iniciante em programação, com a linguagem C, durante o 1º semestre de 2012. A turma é formada por 97 alunos divididos em 4 grupos. Cada grupo realizou 4 atividades contendo 2 problemas diferentes, perfazendo, ao todo, 32 problemas.

Para avaliar as quatro técnicas de normalização de códigos desenvolvidas, os resultados da análise de similaridade foram comparados entre si. Além disso, compararam-se, também, aos resultados apresentados pelas ferramentas JPlag e MOSS.

	Norm 1		Norm 2		Norm 3		Norm 4	
	> 20%	> 70%	> 20%	> 70%	> 20%	> 70%	> 20%	> 70%
Problema 1	13	2	33	1	52	0	10	1
Problema 2	5	0	22	0	19	1	4	0
Problema 3	5	0	23	1	32	4	12	10
Problema 4	4	0	12	0	30	0	30	3
Problema 5	15	0	45	4	38	3	72	10
Problema 6	13	0	59	5	57	3	101	22
Problema 7	12	3	42	4	65	4	15	6
Problema 8	16	0	120	55	71	1	10	5

Tabela 4: Comparação entre as 4 técnicas de normalização

A Tabela 4 relaciona a quantidade de códigos que apresentam índice de similaridade maior do que 20% e 70% para cada tipo de normalização. A análise da Tabela mostra que, em geral, a sensibilidade à identificação de similaridade apresenta um comportamento crescente da técnica 1 para a técnica 4. Entretanto, pode-se observar que, para alguns problemas, como o de número 8, essa característica não se confirma em relação às técnicas 2, 3

e 4, sendo, na verdade, invertidas. Isso mostra que as técnicas podem revelar semelhanças que vão além da organização estrutural do código, fornecendo elementos de análise diferenciados para o professor.

As Figuras 6, 7, 8 e 9 apresentam os gráficos comparativos do número de ocorrência de similaridades para os grupos A, B, E e D da turma de programação, levando-se em consideração apenas as técnicas de normalização 1 e 4, além do JPlag e do MOSS. Pela baixa incidência de similaridade apontada pela técnica de normalização 1 em relação às demais, infere-se que a técnica 1 produz os resultados menos significativos. Além disso, o JPlag, praticamente em todos os problemas, indica o maior número de casos de similaridade, e, por outro lado, o MOSS sempre aponta o menor número de similaridades registra-

das. Isso demonstra que a sensibilidade utilizada pelo MOSS é a mais conservadora para indicação de similaridade. A técnica de normalização 4, com o uso do Sherlock, apresenta resultados intermediários entre os do MOSS e do JPlag em 62% dos problemas para os registros de similaridades acima de 70%. demonstrando, assim, constituir-se em uma promissora ferramenta de análise. Um aspecto importante da técnica 4 associada ao Sherlock é a redução expressiva do tamanho do código a ser comparado devido à eliminação de grande parte do código, o que, na maioria das vezes, permitirá o processamento mais rápido da comparação.

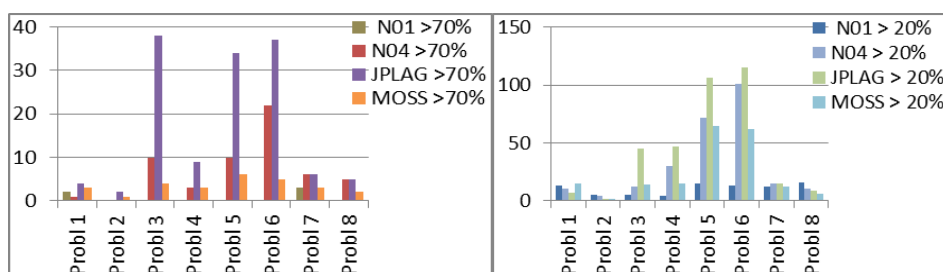


Figura 6: Grupo A: relação da técnica 1 e 4 com JPlag e MOSS

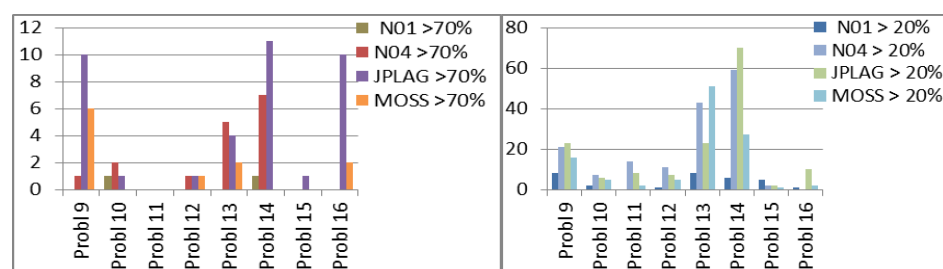


Figura 7: Grupo B: relação da técnica 1 e 4 com JPlag e MOSS

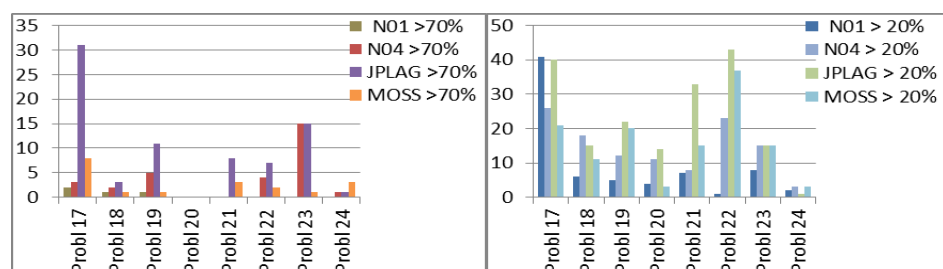


Figura 8: Grupo C: relação da técnica 1 e 4 com JPlag e MOSS

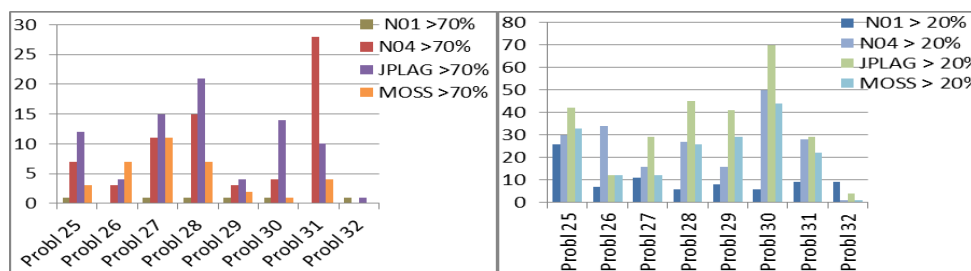


Figura 9: Grupo D: relação da técnica 1 e 4 com JPlag e MOSS

O uso do Sherlock com a técnica de normalização 4, em algumas situações, pode apresentar maior ocorrência de similaridade do que o JPlag, como pode ser observado pelo problema 31, que propõe a impressão por extenso do valor de um algarismo inteiro. A análise visual dos códigos dos alunos mostrou que quase todos desenvolveram uma solução com o uso do *swith*, com código bastante semelhante, o que foi feito por orientação do professor (e poderia também ter sido consequência da uso de algum exemplo encontrado em fontes de referência). Na perspectiva da similaridade, os resultados não são, portanto, considerados falsos positivos para essa questão. Assim, verifica-se que os resultados do Sherlock com a normalização 4 foram mais significativos do que os apresentados pelo JPlag e MOSS.

6 Conclusões e Perspectivas

Este trabalho representa uma iniciativa em termos de instrumentalização para o professor de disciplinas de programação de computador com vistas a mitigar um problema que ocorre com frequência: um grande número de alunos em turmas de laboratório, situação recorrente em disciplinas iniciais em alguns cursos universitários.

Propõe-se, assim, uma tecnologia capaz de auxiliar o professor a melhor gerenciar os seus esforços para o acompanhamento das práticas de programação dos alunos. Sendo uma atividade que exige, com frequência, atenção individualizada, a dificuldade de atendimento a turmas numerosas pode comprometer um aprendizado de qualidade. O auxílio se constitui na automatização de algumas das etapas inerentes ao acompanhamento, sem prescindir da importante e insubstituível ação do professor, buscando também o comportamento mais autônomo do aluno.

O ambiente BOCA-LAB oferece recursos em forma de serviços para dar suporte a estas atividades práticas. Integrado ao ambiente Moodle, as funcionalidades de compilação e correção automática de códigos fonte são disponibilizadas como tarefas do próprio ambiente virtual. A arquitetura proposta pode ser estendida de maneira a permitir a integração do BOCA-LAB a outros ambientes, bastando, para isso, que seja construído um módulo de

integração (MI) específico à plataforma.

A integração do BOCA-LAB a um AVA visa, assim, diminuir a sobrecarga de trabalho do professor no que concerne ao acompanhamento e à avaliação dos resultados das atividades propostas. Como consequência, prospecta-se a melhoria na qualidade do aprendizado de programação, pois se, por um lado, o tempo do professor com atividades de administração e de gestão de recursos pode ser reduzido, por outro lado, sua disponibilidade em termos de atenção aos alunos pode ser maior.

Tendo em perspectiva a administração de recursos técnicos institucionalmente compartilhados entre múltiplos cursos e usuários, atribuindo à solução a características de escalabilidade necessária a esse tipo de situação o balanceamento de carga se mostrou funcional no testes realizados. Na versão atual, a distribuição dos programas se baseia somente na quantidade de códigos fonte ainda não processados e armazenados nos servidores. Entretanto, o modelo pode ser adaptado facilmente para técnicas de balanceamento que levem em consideração outros parâmetros, como a complexidade dos códigos enviados ou as características físicas dos servidores, como quantidade de memória livre e uso de CPU, entre outros fatores.

Expandindo as perspectivas de análise e acompanhamento do professor, o ambiente desenvolvido conta com a análise de similaridade entre códigos-fonte. Foram desenvolvidas e avaliadas técnicas de normalização que, de acordo com os resultados apresentados, são comparáveis as duas principais ferramentas de detecção de plágio referenciadas na literatura. A possibilidade de configuração de uma das quatro técnicas de normalização propostas permite alterar a sensibilidade da comparação, ressaltando características diferenciadas em contextos particulares. Apesar de a técnica 4 ser a mais sensível em termos estruturais, é possível que seja importante para o professor, por exemplo, considerar os nomes de funções ou de palavras reservadas na comparação entre códigos, o que o remete para as outras técnicas. Além disso, é importante frisar que a ferramenta pode ser utilizada para finalidades diferentes da detecção de plágio. É possível que a similaridade seja até mesmo desejável. Para exemplificar, o professor pode usar a comparação para verificar se as

soluções apresentadas foram baseadas em um exemplo fornecido ou em uma orientação dada aos alunos, ou, ainda, se foram usadas lógicas e estruturas diferentes daquelas que foram propostas.

Além das melhorias no ambiente BOCA-LAB que estão sendo continuamente desenvolvidas em consequência das observações feitas por professores e alunos, paralelamente, estão sendo realizadas pesquisas mais aprofundadas sobre a análise de similaridade. Outros algoritmos de comparação foram integrados ao ambiente e estão sendo estudados de maneira a buscar maior qualidade em termos de identificação de semelhanças entre códigos-fonte. Pesquisa-se, ainda, abordagens de análise integrando grupos de alunos, não só com a perspectiva de identificação de possíveis situações de plágio, mas também para identificar grupos de alunos que trabalham em parceria, erros comuns ou comportamentos diversos que possam auxiliar na compreensão de como a aprendizagem vem se desenvolvendo e como se pode reagir a eventuais dificuldades identificadas.

Referências

- [1] Essi L, Kirsti Ala-Mutka, Hannu-Matti J. (2005). "A study of the difficulties of novice programmers". In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05).ACM, New York, NY, USA, 14-18.
- [2] Tan, Phit-Huan; Ting, Choo-Yee; Ling, Siew-Woei. (2009). "Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception," Computer Technology and Development, 2009. ICCTD '09. International Conference on Computer Technology and Development, vol.1, no., pp.42-46, 13-15 Nov. 2009.
- [3] Papazoglou, Mike P.; Heuvel, Willem-Jan van den. (2007) "Service Oriented Architectures: approaches, technologies and research issues. The VLDB Journal, v. 16, p. 389-415, 2007.
- [4] De Campos, C. P. ; Ferreira, C. E. (2004). "BOCA: Um Sistema de Apoio para Competições de Programação." Workshop de Educação em Computação, 2004, Salvador. Anais do Congresso da SBC, 2004.
- [5] Hage, J.; Rademaker, P.; Vugt, N. V. A comparison of plagiarism detection tools. Utrecht University. Utrecht, The Netherlands, p. 28. 2010.
- [6] A. S. Bin-Habtoor and M. A. Zaher, "A Survey on Plagiarism Detection Systems," International Journal of Computer Theory and Engineering vol. 4, no. 2, pp. 185-188, 2012.
- [7] Đurić, Z., & Gašević, D. (2012). A Source Code Similarity System for Plagiarism Detection . The Computer Journal . doi:10.1093/comjnl/bxs018
- [8] Prechelt, L.; Malpohl, G. & Phlippsen, M. (2002). "Finding plagiarisms among a set of programs with JPlag". J. UCS – Journal of Universal Computer Science.
- [9] "A System for Detecting Software Plagiarism". Univ. California, Berkeley. Disponível em "http://theory.stanford.edu/~aiken/moss/". Acesso em: 01 de outubro de 2012.
- [10] The Sherlock Plagiarism Detector. Disponível em:"http://sydney.edu.au/engineering/it/~scilect/sherlock/" Acesso em: 01 de outubro de 2012.
- [11] Moodle – "A Free, Open Source Course Management System for Online Learning." Disponível em http://moodle.org/.
- [12] Sakai: Collaborative and Learning Environment for Education. Disponível em https://confluence.sakaiproject.org/display/WEB+SVCS/Home. Acesso em: 01 de outubro de 2012.
- [13] Ng, S.C., Choy, S.O., Kwan, R., Chan, S.F.: A Web-Based Environment to Improve Teaching and Learning of Computer Programming in Distance Education. In: Lau, R., Li, Q., Cheung, R., Liu, W. (eds.) ICWL 2005. LNCS, vol. 3583, pp. 279–290. Springer, Heidelberg (2005).
- [14] Wang, J., Chen, L., Zhou, W.: Design and Implementation of an Internet-Based Platform for C Language Learning. In ICWL(2008) 187-195.
- [15] Sistema Susy. Disponível em: www.ic.unicamp.br/~susy/ Acesso em: 01 de outubro de 2012.
- [16] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The boss online submission and assessment system. J. Educ. Resour. Comput. 5, 3, Article 2 (September 2005). DOI=10.1145/1163405.1163407 http://doi.acm.org/10.1145/1163405.1163407
- [17] Gareth Thorburn, Glenn Rowe, PASS: An automated system for program assessment, Computers & Education, Volume 29, Issue 4, December 1997, Pages 195-206, ISSN 0360-1315, 10.1016/S0360-1315(97)00021-3.
- [18] A. Zeller, "Making Students Read and Review Code", SIGCSE Bull., ACM, New York, NY,

- USA, 32(2000)3, pp. 89-92
- [19] VPL – “Virtual Programming Lab Disponível” em: “<http://vpl.dis.ulpgc.es/>”. Acesso em 01 de outubro de 2012.
- [20] Onlinejudge. Disponível em: <https://github.com/hit-moodle/onlinejudge>. Acessado em 1 de outubro de 2012.
- [21] Sphere Research Labs – IDE ONE Disponível em <http://ideone.com/>. Acesso em 1 de outubro de 2012.
- [22] Grune, D. and Vakgroep, M. 1989. Detecting copied submissions in computer science workshops. Tech. rep., Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit.
- [23] Lancaster, Thomas; Culwin, Fintan. (2001). "Towards an error free plagiarism detection process". In Proceedings of the 6th annual conference on Innovation and technology in computer science education (ITiCSE '01). ACM, New York, NY, USA, 57-60.
- [24] Chen, Xin; Francia, Brent; Li, Ming; Mckinnon, Brian; Seker, Amit; (2004). “Shared information and program plagiarism detection,” IEEE Transactions on Information Theory, vol. 50, no. 7, pp. 1545–1551, 2004.
- [25] Ahtiainen, Aleks; Surakka, Sami; Rahikainen, Mikko. (2006). "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises". In Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research, pages 141–142, New York, NY, USA, 2006. ACM.
- [26] Burrows, S.; Tahaghoghi, S.M.M.; Zobel, J. (2007). "Efficient and effective plagiarism detection for large code repositories". Software-Practice & Experience, 37(2), 151-175.
- [27] Schleimer, S., Wiljerson, D. S., & Aiken, A. (2003) "Winnowing: local algorithms for document fingerprinting". In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). ACM, New York, NY, USA.
- [28] Alves, L. and Brito, M. (2005) “O Ambiente Moodle como Apoio ao Ensino Presencial.” Disponível em: www.abed.org.br/congresso2005/por/pdf/085tcc3.pdf. Acesso em: 01 de outubro de 2012.
- [29] FRANÇA, A. B; SOARES, J. M. Sistema de apoio a atividades de laboratório de programação via Moodle com suporte ao balanceamento de carga. In: Simpósio Brasileiro de Informática na Educação, 22., 2011, Aracaju. Anais do 22º SBIE. Aracaju, 2011.