UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

ALLYSON BONETTI FRANÇA

O CÓDIGO POR TRÁS DO CÓDIGO-FONTE: MUDANÇA DE REPRESENTAÇÃO PARA A ANÁLISE DE SIMILARIDADE

FORTALEZA

2019

ALLYSON BONETTI FRANÇA

O CÓDIGO POR TRÁS DO CÓDIGO-FONTE:
MUDANÇA DE REPRESENTAÇÃO PARA A ANÁLISE DE SIMILARIDADE

Tese apresentada ao Programa de Pós-Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Doutor em Engenharia de Teleinformática. Área de concentração: Sinais e Sistemas.

Orientador: Prof. Dr. José Marques Soares.

Coorientador: Prof. Dr. Giovanni Cordeiro Barroso.

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

F881c França, Allyson Bonetti.
O código por trás do código-fonte : mudança de representação para a análise de similaridade / Allyson Bonetti França. – 2019.
99 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Tecnologia, Programa de Pós-Graduação em Engenharia de Teleinformática, Fortaleza, 2019.
Orientação: Prof. Dr. José Marques Soares.
Coorientação: Prof. Dr. Giovanni Cordeiro Barroso.

1. Similaridade entre códigos-fonte. 2. Detecção de plágio. 3. Ferramenta de detecção de similaridade. 4. I-Sim2 e método de conformidade. I. Título.

CDD 621.38

ALLYSON BONETTI FRANÇA

O CÓDIGO POR TRÁS DO CÓDIGO-FONTE:
MUDANÇA DE REPRESENTAÇÃO PARA A ANÁLISE DE SIMILARIDADE

Tese apresentada ao Programa de Pós-Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Doutor em Engenharia de Teleinformática. Área de concentração: Sinais e Sistemas.

Aprovada em: ___/_____/_____

BANCA EXAMINADORA

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

_____
Prof. Dr. Giovanni Cordeiro Barroso (Coorientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. George André Pereira Thé
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Danielo Gonçalves Gomes
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Ig Ibert Bittencourt Santana Pinto
Universidade Federal de Alagoas (UFAL)

_____
Prof. Dr. Daniel Cardoso Moraes de Oliveira
Universidade Federal Fluminense (UFF)

RESUMO

A similaridade identificada entre códigos-fonte enviados por alunos em disciplinas de programação é frequentemente utilizada como indicativo de plágio por professores e/ou sistemas de submissão automatizados. Dentre as técnicas de se medir a similaridade, inclui-se o uso de estruturas sintáticas e de padrões léxicos. Apesar da ampla quantidade de ferramentas disponíveis que se utilizam dessa técnica, poucas são capazes de identificar a similaridade, de maneira eficaz, o que se deve à complexidade inerente a esse tipo de análise. Uma limitação dessa técnica é a sensibilidade às modificações na organização de caracteres e símbolos, mesmo quando sutis, e suas disposições ao longo das expressões. Investiga-se neste trabalho a similaridade entre códigos pré-processados, de maneira a remover os aspectos irrelevantes da codificação, bem como, ressaltar as suas características relevantes. As contribuições apresentadas inserem-se na ação do pré-processamento, que resulta em uma recodificação, e na transformação com mudança no domínio da representação: o código por trás do código-fonte. Tais contribuições visam mitigar a complexidade decorrente da análise do código através do uso de estruturas sintáticas e de padrões léxicos. Na perspectiva da recodificação, são realizados e analisados ajustes invasivos nas estruturas do código original que, mesmo em caso de modificações propositais pelo desenvolvedor, preservam características dificilmente comprometidas ou corrompidas. Essa análise é feita com o algoritmo Sherlock N-Overlap e com dois tipos de pré-processamento – denominados normalizações –, sendo que uma destas é proposta neste trabalho. Na perspectiva da transformação foi realizada a representação dos códigos-fonte por meio de imagens digitais. Para isso, foi desenvolvido o I-Sim, que traduz a organização das estruturas do programa em arranjos visuais, permitindo identificar a similaridade entre códigos-fonte. Foram realizados experimentos com 84 códigos-fonte propositalmente modificados e uma base composta por 2160 códigos criados por estudantes de cursos de engenharia em aulas de programação. No último conjunto, a situação de eventuais modificações propositais não é conhecida *a priori*, usando-se um método específico para cálculo relativo da precisão e da revocação com base em um conjunto de oráculos. Os resultados apresentados mostram que, na maioria dos casos, os índices de similaridade das soluções desenvolvidas se mostram superiores ao SIM, ao JPlag e ao MOSS, ferramentas utilizadas como referência pela literatura.

Palavras-chave: similaridade entre códigos-fonte, detecção de plágio, ferramenta de detecção de similaridade, I-Sim2 e método de conformidade

ABSTRACT

The similarity identified between source codes sent by students in programming disciplines is often used as indicative of plagiarism by teachers and / or automated submission systems. Among the techniques of measuring similarity, we include the use of syntactic structures and lexical patterns. Despite the large number of available tools that use this technique, few are able to identify the similarity, effectively, due to the inherent complexity of this type of analysis. One limitation of this technique is the sensitivity to the modifications in the organization of characters and symbols, even when subtle, and their dispositions along the expressions. We investigate the similarity between preprocessed codes in order to remove the irrelevant aspects of coding, as well as to highlight their relevant characteristics. The contributions presented are inserted in the preprocessing action, which results in a recoding, and in the transformation with change in the representation domain: the code behind the source code. These contributions aim to mitigate the complexity of code analysis through the use of syntactic structures and lexical patterns. In the perspective of recoding, invasive adjustments are made and analyzed in the structures of the original code that, even in the case of purposive modifications by the developer, preserve characteristics that are difficult to compromise or corrupt. This analysis is done with the Sherlock N-Overlap algorithm and two types of pre-processing, called normalizations. One of the normalizations is proposed in this work. In the perspective of the transformation, the representation of the source codes through digital images was performed. For this, I-Sim was developed, which translates the organization of the program's structures into visual arrangements that allow the identification of similarity between source codes. Experiments were conducted with 84 purposely modified source codes and a base composed of 2160 codes created by students of engineering courses in programming classes. In this last set, the situation of similarity was not previously known, so a method was used to calculate precision and recall, in a relative way, based on a set of reference tools, as a kind of oracle. The results show that, in most cases, the similarity indexes of the solutions developed are superior to reference tools in the literature, such as SIM, JPlag and MOSS.

Keywords: similarity between source codes, plagiarism detection, similarity investigation tool, I-Sim2, method of conformity.

LISTA DE FIGURAS

Figura 1.1 - Códigos de alunos distintos enviados como solução de uma atividade simples de programação 13
Figura 1.2 - Códigos semelhantes de alunos distintos enviados como solução de uma atividade de programação 13
Figura 2.1 - Técnicas de similaridade 19
Figura 2.2 - Trecho de código na linguagem C. 23
Figura 2.3 - Exemplo de uma árvore sintática abstrata (AST) 23
Figura 2.4 - Exemplo de código Java e os respectivos tokens. 25
Figura 2.5 - Código destacando os tokens 27
Figura 2.6 - Imagem digital 36
Figura 2.7 - Modelo de cor RGB. 37
Figura 2.8 - Modelo do Processo de Visualização 38
Figura 2.9 - Exemplo de um mapa em árvores 39
Figura 2.10 - Exemplo de um mapa de árvore de uma aplicação orientada a objetos 40
Figura 2.11 - Exemplo de um mapa de árvore aplicado para mostrar a complexidade de um software. 41
Figura 2.12 - Diagrama de blocos para a obtenção do descritor Color Spectrum. 42
Figura 3.1 – Comparação entre o código original e as normalizações. 50
Figura 3.2 - Árvore sintática abstrata colorida 51
Figura 3.3 - Etapas de geração da imagem 52
Figura 3.4 - Processo de geração de imagem 53
Figura 3.5 - Subtração de imagens iguais 54
Figura 3.6 - Subtração de imagens diferentes 55
Figura 3.7 - Conversão do código em grafo 56
Figura 3.8 - Extração do descritor Color Spectrum a partir de um subgrafo. 56
Figura 3.9 - Códigos e seus respectivos grafos. 59
Figura 3.10 - Imagem gerada tendo como base o código da Figura 3.1 (a). 60
Figura 4.1 - Média harmônica calculada por limiar para os algoritmos SIM, MOSS e JPlag de códigos do Grupo A desenvolvidos pelo programador 1. 65
Figura 4.2 - Média harmônica calculada por limiar para os algoritmos Sherlock N-Overlap com normalização 4 e 5 de códigos do Grupo B desenvolvidos pelo programador 1. 66

Figura 4.3 - Média harmônica calculada por limiar para os algoritmos I-Sim1 e I-Sim2 de códigos do Grupo B desenvolvidos pelo programador 1 67
Figura 4.4 - Imagem representativa do código não-plágio-2A.c. 68
Figura 4.5 - Imagem representativa do código original.c. 68
Figura 4.6 - Imagem resultante da sobreposição de pixels entre as Figura 4.2 e Figura 4.3. 69
Figura 4.7 - Média harmônica calculada por limiar para todos os algoritmos envolvidos de códigos do Grupo A desenvolvidos pelo programador 1. 69
Figura 4.8 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 1 (limiar> 60%). 71
Figura 4.9 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 1 (limiar> 60%). 72
Figura 4.10 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 2 (limiar> 60%). 72
Figura 4.11 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 2 (limiar> 60%) 73
Figura 4.12 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 3 (limiar> 60%). 73
Figura 4.13 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 1 (limiar> 60%). 74
Figura 4.14 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 3 (limiar> 60%). 74
Figura 4.15 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – Sherlock N-overlap - normalização 4 sob análise. 79
Figura 4.16 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – Sherlock N-overlap - normalização 5 sob análise. 80
Figura 4.17 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – I-Sim2 sob análise. 80
Figura 4.18 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe B (2012) por limiar de similaridade 81
Figura 4.19 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe C (2012) por limiar de similaridade 81

Figura 4.20 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe D (2012) por limiar de similaridade.....82
Figura 4.21 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe A (2013) por limiar de similaridade.....82
Figura 4.22 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe B (2013) por limiar de similaridade.....83
Figura 4.23 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe C (2013) por limiar de similaridade.....83
Figura 4.24 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe D (2013) por limiar de similaridade.....84
Figura 4.25 - Comparação entre dois códigos semelhantes.....86

LISTA DE TABELAS

Tabela 2.1 - Exemplo de aplicação da normalização 4 em um código desenvolvido na Linguagem C. 30
Tabela 2.2 - Representação dos parâmetros de cálculo para precisão e revocação. 32
Tabela 3.1 - Exemplo de palavras reservadas da linguagem C e seus respectivos agrupamentos. 46
Tabela 3.2 - Substituição das palavras reservadas por máscaras 47
Tabela 3.3 - Aplicação da normalização 4 e substituição das máscaras pelas palavras de agrupamento 47
Tabela 3.4 - Aplicação da normalização 5 em declarações aglomeradas 48
Tabela 3.5 - Exemplo de omissão de chaves 49
Tabela 3.6 - Associação dos atributos e suas cores correspondentes 52
Tabela 3.7 - Exemplo de palavras reservadas da linguagem C, C++ e suas respectivas palavras de agrupamento 59
Tabela 4.1 - Porcentagens de similaridades encontradas pelas ferramentas analisadas dos códigos do programador 1 desenvolvidos para o grupo B. 63
Tabela 4.2 - Relação quantitativa dos códigos analisados no ano de 2012. 76
Tabela 4.3 - Relação quantitativa dos códigos analisados no ano de 2013. 77
Tabela 4.4 - Relatório completo de quantificação de ocorrências. 78
Tabela 4.5 - Tabela contendo os percentuais de similaridade encontrados pelas ferramentas analisadas. 85
Tabela A.1 - Palavras de linguagem e suas cores correspondentes 97

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Objetivos Geral e Específicos 15
1.2 Contribuições 15
1.3 Organização do documento de tese 16
2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 Plágio ou similaridade? 17
2.2 Técnicas de Plágio 18
2.3 Técnicas de análise de similaridade 21
2.3.1 Técnicas baseadas em texto 22
2.3.2 Técnicas baseadas em tokens 22
2.3.3 Técnicas baseadas em árvores 22
2.4 Ferramentas para detecção e análise de similaridade 24
2.4.1 Jplag 24
2.4.2 SIM 25
2.4.3 MOSS 25
2.4.4 Sherlock N-overlap 26
2.4.4.1 Parâmetros de configuração 26
2.4.4.2 Geração de Assinaturas 27
2.4.4.3 Comparação de Assinaturas 28
2.4.4.4 Representação Normalizada dos Códigos 29
2.5 Comparação de Índices de Similaridade 31
2.5.1 Método Tradicional 31
2.5.2 Método de Conformidade 33
2.6 Estudo de Imagens Digitais para representação de códigos-fonte 35
2.6.1 Imagens Digitais 35
2.6.2 O Modelo RGB de Cores 36
2.6.3 Técnicas de Visualização de Dados 37
2.6.3.1 Técnicas Hierárquicas 39
2.6.4 Color Spectrum 41
2.7 Considerações Finais 43
3 RECODIFICAÇÃO E TRANSFORMAÇÃO DE CÓDIGOS-FONTE PARA A ANÁLISE DE SIMILARIDADE 45

3.1 Representação de códigos-fonte através de normalizações 45
3.1.1 Normalização 5 46
3.2 I-Sim: Representação de códigos-fonte através da imagem 50
3.2.1 Transformando código em imagem 51
3.2.2 Cálculo do índice de similaridade por sobreposição de imagens 54
3.2.3 Cálculo do índice de similaridade entre as imagens utilizando descritores 55
3.2.4 Discussões sobre a técnica de construção da imagem 57
4 RESULTADOS 61
4.1 Código-fonte com similaridade conhecida a *priori* 61
4.1.1 Cenário de Experimentação 61
4.1.2 Usando o método tradicional para a comparação entre ferramentas 63
4.1.3 Discussão dos Resultados 71
4.2 Código-fonte com similaridade não conhecida a *priori* 75
4.2.1 Cenário de Experimentação 76
4.2.2 Utilizando o método de conformidade para a comparação entre ferramentas 77
4.2.3 Discussão dos Resultados 79
4.3 Limitação do método de conformidade 84
5 CONCLUSÃO 87
REFERÊNCIAS 91
APÊNDICE A – TABELA COM AS PALAVRAS DE LINGUAGEM E SUAS CORES CORRESPONDENTES 97
APÊNDICE B – ENUNCIADOS DOS PROBLEMAS PLAGIADOS PROPOSITALMENTE 98

1 INTRODUÇÃO

A similaridade identificada entre documentos representa frequentemente uma indicação de plágio, sendo uma grande preocupação e um assunto estudado em diferentes trabalhos (BRETAG, 2016; CARROLL, 2007; CHUDA *et al*., 2012; JOY *et al.*, 2011; ORTEGO *et al.*, 2012; WEBER-WULFF, 2014). O ato de plagiar viola o senso de justiça da sociedade, bem como, as normas do mundo acadêmico. Investigar e abordar o plágio demanda tempo, recursos e profissionais, e por isso está sendo discutido em todo o mundo, seja para diferenciar o intencional do inadvertido, ou simplesmente por causa do impacto do tempo desperdiçado para a sua identificação (LUQUINI; OMAR, 2011; MANOHARAN, 2017; THURMOND, 2011).

No Brasil, existe um atraso, em termos de medidas preventivas, no que diz respeito à adoção de código de ética e de apoio institucional para lidar com o plágio (KROKOSCZ, 2011). O acompanhamento desse problema é feito muitas vezes por iniciativas isoladas de professores, evidenciando, portanto, a banalização da cópia de trabalhos entre alunos, com prejuízos na formação acadêmica e pessoal (KROKOSCZ, 2011).

Anteriormente, para se plagiar algo, o autor necessitava muitas vezes redigitar trechos completos de livros ou artigos impressos, o que exigia maior esforço. Com o surgimento da Internet, os textos podem ser copiados facilmente de artigos disponíveis ou por meio de mídias digitais. A Internet se mostra ao usuário como uma ferramenta de acesso livre e fácil para qualquer tipo de informação, o que faz com que ele pense que pode se apropriar de ideias de outros sem a devida atribuição da autoria, simplesmente porque isso pode ser feito com facilidade (EVERING; MOORMAN, 2012; GIL; PALMA; LAHENS, 2014).

A prática de se plagiar não ocorre apenas com documentos textuais, imagens ou letras de música, mas também pode ocorrer em códigos-fonte de programas de computador. Como afirma Vogts (2009), as disciplinas iniciais de programação não estão livres de tal ocorrência. Em questões nas quais o estudante tem que desenvolver um código, é comum encontrar respostas idênticas ou, ainda, com alterações mínimas realizadas na tentativa de dificultar a percepção do plágio por parte de professores e/ou monitores.

É preciso considerar, entretanto, a mudança cultural promovida pela inserção da tecnologia nos hábitos cotidianos. O uso de sistemas computacionais como ferramenta de suporte a atividades educacionais, por exemplo, vem se popularizando nos últimos anos. Consequentemente, as maneiras de interagir e conduzir o processo de ensino e aprendizagem

estão sofrendo influências não negligenciáveis (DE LA TORRE *et al.*, 2015; SONEGO *et al.*, 2014).

Esse fato vem ao encontro do que afirma Lévy (1993), segundo o qual "as relações entre os homens, o trabalho e a própria inteligência dependem, na verdade, da metamorfose incessante de dispositivos informacionais de todos os tipos" (LEVY; COSTA, 1993). Diante destas transformações, muitas instituições de ensino estão utilizando cada vez mais ferramentas computacionais para dar suporte a atividades em seus cursos e disciplinas. Para organizar o espaço de trabalho, professores fazem o uso frequente de aplicações Web e aplicativos de dispositivos móveis a fim de obter maior interação no ensino, disponibilização de notas de aula, proposição e submissão de trabalhos, bem como o registro de resultados de avaliações (CORREIA; SANTOS, 2018; FRANÇA; MACIEL; SOARES, 2013; NOZAL *et al.*, 2013).

Em disciplinas de Introdução à Programação, o uso de ferramentas, como o Moodle, para o envio e controle de problemas computacionais, melhora a gerência das atividades pelo professor/monitor, mas, por outro lado, facilita a cópia de soluções entre os alunos. Esse comportamento é difícil de ser controlado, principalmente em turmas numerosas (FRANÇA; MACIEL; SOARES, 2013; LE *et al*., 2013).

Dessa forma, é importante o emprego de ferramentas que façam o controle de plágio visando a desmotivar tal postura, quando não autorizada, ou mesmo acompanhar eventuais trocas de experiências entre os envolvidos (LE *et al*., 2013). Entretanto, tratando-se de códigos-fonte, nem sempre as similaridades observadas se explicam por ações intencionais de plágio. Outros aspectos relevantes devem ser levados em consideração, dependendo da perspectiva e do contexto da atividade.

A semelhança entre soluções pode ser também indicativa do trabalho coletivo entre alunos, de questões de simples resolução, como podemos ver ilustrado na Figura 1.1, de referências encontradas em livros ou, muitas vezes, de embasamento em exemplos disponibilizados pelo professor para problemas de natureza semelhante. Dessa maneira, ao invés de "detecção de plágio", adota-se prioritariamente neste trabalho a expressão "análise de similaridade", independentemente do propósito para o qual a similaridade é detectada.

Dentre as formas de se identificar a similaridade entre documentos, inclui-se a medição de estruturas sintáticas e léxicas coincidentes sob algum princípio de comparação. Uma limitação deste tipo de técnica, entretanto, é a sensibilidade às modificações na organização de caracteres e símbolos, ainda que sutil, e suas disposições ao longo das expressões.

Figura 1.1 - Códigos de alunos distintos enviados como solução de uma atividade simples de programação

```c
int main( void )
 {
int i,  n;
printf( "digite o numero de termos para a sequencia " );
scanf( " % d",  &n );
for( i=1; i <=n; i++ )
 {
if( i % 3 ==1 )
printf( " % d ", ( i / 3 ) + 1 );
else
if( i % 3 ==2 )
printf( " % d ", ( i / 3 ) + 4 );
else
printf( " % d ", ( i / 3 ) + 3 );
 }
return 0;
 }
```

```c
int main()
 {
int a,  b,  x,  y,  z;
printf( "digite o numero de termos " );
scanf( " % d",  &b );
printf( " " );
for( a=1; a <=b; a++ )
 {
if( a % 3 ==1 )
printf( " % d,  ", ( a / 3 ) +1 );
else
if( a % 3 ==2 )
printf( " % d,  ", ( a / 3 ) +4 );
else
printf( " % d,  ", ( a / 3 ) +3 );
 }
return 0;
 }
```

Fonte: Próprio autor.

Em muitos casos, com uma simples alteração, as ferramentas que se baseiam na medição de estruturas sintáticas e léxicas não encontram similaridades na comparação dos códigos-fonte, como podemos ver na Figura 1.2, onde temos dois códigos semelhantes em que as ferramentas não conseguiram detectar nenhuma similaridade, retornando um percentual igual a zero.

Figura 1.2 - Códigos semelhantes de alunos distintos enviados como solução de uma atividade de programação

```c
#include<stdio.h>

int main(){

float vel1, vel2, vel3, vel4;

printf("Digite a primeira velocidade em metros por segundo(m/s)\n");
scanf("%f", &vel1);

printf("Digite a segunda velocidade em kilometros por hora(km/h)\n");
scanf("%f", &vel3);

vel2=vel1*3.6;
vel4=vel3/3.6;
printf("Sua primeira velocidade é %g kilometros por hora(km/h)\n",vel2);
printf("Sua segunda velocidade é %g metros por segundo(m/s)\n",vel4);

printf("\nFIM DO PROGRAMA\n");


return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main ()
{

  float v1, v2;
  printf ("Digite a velocidade em metro por segundo:\n");
  scanf ("%f", &v1);
  v1 = v1*3.6;
  printf ("A velocidade em kilometros por hora eh:%.2f \n", v1);
  printf ("Digite a velocidade em kilometros por hora:\n");
  scanf ("%f", &v2);
  v2 = v2/3.6;
  printf ("A velocidade em metros por segundo eh:%.2f \n", v2);

  return 0;
}
```

Fonte: Próprio autor.

Para mitigar esse problema, um artifício adotado por diversos autores, como em Li *et al.*, 2014, Mubarak Ali e Sulaiman, 2014, e Sabi, Higo e Kusumoto, 2017, é a eliminação de potenciais ambiguidades por meio da submissão dos códigos originais (a serem

comparados) a um pré-processamento (normalização), de maneira a reorganizá-lo, removendo os aspectos irrelevantes da codificação e destacando mais apropriadamente as suas características relevantes. Contudo, percebe-se que a ação do pré-processamento pode ser aprofundada a ponto de implicar em uma nova codificação, eventualmente sendo possível promover uma mudança no domínio da representação (transformação): nesse caso, temos um outro *código por trás do código-fonte original*.

Na normalização, o código passa por um processo de aplicação de regras preestabelecidas; ao final desse processo, os atributos do código ainda podem ser visualizados, mesmo que modificados. Na transformação, por sua vez, temos uma nova representação para aquele código-fonte, em que muitas vezes não se consegue ter ideia do código original.

Assim, tem-se por hipótese nesta tese ser possível identificar o nível de similaridade entre códigos-fonte originais a partir de sua recodificação ou, até mesmo, da transformação em outro tipo de código, por meio de pré-processamento. Portanto, pretende-se mostrar que, por meio de um conjunto de padrões de modificação das representações dos códigos originais, é possível conservar no código transformado os elementos necessários à identificação da similaridade entre os objetos originais.

Este trabalho foi iniciado com o estudo e aprimoramento de técnicas de pré-processamento no contexto do Sherlock N-overlap (FRANÇA; MACIEL; SOARES, 2013; MACIEL, 2014), uma ferramenta baseada no Sherlock (PIKE; LOKI, 2017) na qual foi adicionado suporte a técnicas de pré-processamento e substituído o coeficiente de similaridade original pelo *overlap.* Tais técnicas propõem a normalização dos códigos antes da comparação efetiva, buscando remover os aspectos irrelevantes e destacar os importantes, permitindo, assim, uma comparação mais dirigida e objetiva.

Neste trabalho, um novo tipo de normalização foi proposto e analisado para o Sherlock N-Overlap. Em seguida, estudou-se a comparação dos códigos através de novas normalizações que permitem a mudança de representação do código original no domínio das imagens. Assim, com o uso de técnicas de processamento de imagens, pode-se identificar a similaridade entre os códigos originais.

As investigações e experimentações realizadas estão contextualizadas em ambientes educacionais, podendo fornecer ao professor uma ferramenta de identificação de pares de códigos com altas similaridades, seja para denunciar situações de suspeita de plágio – a ser utilizada para analisar o tipo de solução geralmente empregada pelos alunos –, seja para

outro propósito educacional qualquer. Entretanto, as técnicas desenvolvidas podem ser estendidas para outros domínios, embora, para isso, necessitem de avaliações específicas.

Os resultados do aprimoramento das técnicas de normalização utilizadas pelo algoritmo Sherlock N-overlap, bem como, a técnica de busca de similaridade através da transformação do código na linguagem C em imagem, foram comparados com os obtidos com o uso do JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2002), do MOSS (SCHLEIMER; WILKERSON; AIKEN, 2003) e do SIM (GRUNE, 2018a), ferramentas amadurecidas e destacadas na literatura sobre detecção de plágio entre códigos (RAGKHITWETSAGUL; KRINKE; CLARK, 2018) .

Os objetivos específicos e geral perseguidos com este trabalho são detalhados na Subseção 1.1.

1.1 Objetivos Geral e Específicos

O objetivo geral deste trabalho é conceber, implementar e avaliar métodos e técnicas que permitam identificar a similaridade entre códigos-fonte com duas abordagens: (i) pré-processamento com recodificação e (ii) transformação para domínio das imagens.

Como objetivos específicos, elencam-se:

1. Propor técnicas para realizar o pré-processamento de códigos, removendo informações irrelevantes e/ou destacando características peculiares dos códigos a serem comparados, de forma a melhorar, assim, a sua avaliação pelo Sherlock N-Overlap;
2. Apresentar técnicas para transformar um código-fonte em uma imagem, a fim de conservar nesta as propriedades visuais necessárias que representem as peculiaridades do código original, visando à busca por similaridades;
3. Propor métricas que, por meio das imagens resultantes das transformações propostas (objetivo 2), permitam capturar o índice de similaridade entre os códigos-fonte originais.

1.2 Contribuições

As principais contribuições deste trabalho são:

1. A proposição da normalização 5, no contexto do Sherlock-N-overlap. Trata-se de um algoritmo de pré-processamento configurável que efetua a recodificação de atributos de linguagem e utiliza *templates* para a exclusão da comparação de trechos de código predefinidos, visando destacar apenas as características mais relevantes a serem consideradas na análise de similaridade.
2. A criação de uma abordagem para a transformação de códigos-fonte em imagens, permitindo a identificação da similaridade por meio da comparação em outro domínio;
3. A proposição de duas métricas para o cálculo de similaridade: uma utilizando a comparação por identificação da superfície de recobrimento e outra baseada na extração e comparação de descritores topográficos;

Uma parte dos resultados deste trabalho está publicado no artigo *Sherlock N-overlap: invasive normalization and overlap coefficient for the similarity analysis between source code* no periódico *IEEE Transactions on Computers*, DOI: 10.1109/TC.2018.2881449.

1.3 Organização do documento de tese

Este documento está organizado em 5 capítulos. No Capítulo 1, foi feita a contextualização do trabalho, bem como, foram apresentados os objetivos da pesquisa. No Capítulo 2, são discutidos os aspectos que fundamentam e dão suporte à análise de similaridade e descritores de imagens. O método proposto para uma nova normalização de códigos e sua transformação em imagem, por sua vez, é detalhado no Capítulo 3. Já no Capítulo 4, são discutidos os resultados alcançados. Finalmente, no Capítulo 5, são elencadas as conclusões e as perspectivas de aprimoramento deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os principais conceitos que envolvem a área de análise de similaridade em código-fonte e também alguns aspectos importantes para representação de códigos por imagens, extração de características e comparação entre imagens.

2.1 Plágio ou similaridade?

O plágio pode ser caracterizado como a ação de se apropriar ilegitimamente de conteúdos existentes em uma obra sem dar os devidos créditos ao autor (GIL; PALMA; LAHENS, 2014). Esse ato pode ser encontrado em diversas áreas do conhecimento, tais como educação, artes, fotografia, música, publicidade, entre outras, e muitas vezes o indivíduo que cometeu tal atitude desconhece que se trata de um crime (ALI; DAHWA; SNÁSEL, 2011; GIL; PALMA; LAHENS, 2014).

A similaridade, por sua vez, mede o alto grau de semelhança entre objetos, determinando quão parecido um objeto é de outro. Vale notar que nem sempre um alto grau de similaridade representa que ocorreu um plágio; por isso, é preciso avaliar de forma mais minuciosa para se chegar a essa conclusão.

Em se tratando de exercícios de programação, é comum que alunos de turmas iniciais de disciplinas que envolvem este tipo de conteúdo copiem e/ou alterem códigos-fonte de colegas durante práticas realizadas em laboratório. Isso se deve, muitas vezes, à dificuldade de alguns em resolver os problemas propostos ou por dificuldade em administrar o tempo disponível para o desenvolvimento.

É importante considerar, porém, que nem sempre as coincidências entre dois códigos são devidas ao plágio. A similaridade pode ser ocasionada pela simplicidade do problema proposto, pelo "vocabulário" pouco desenvolvido da linguagem de programação, pela pouca experiência dos estudantes ou até mesmo pela adoção de exercícios de referência propostos pelos professores ou encontrados na literatura específica (CHUDA *et al*., 2012; JOY *et al*., 2011).

Assim, neste trabalho, prefere-se usar o termo similaridade ao plágio, não se procurando rotular ou julgar os motivos das semelhanças entre os códigos analisados. Entretanto, como uma das aplicações principais deste tipo de análise é voltada para a detecção

de plágio, não há como evitar, nesta fundamentação, explorar as técnicas e as referências associadas a este assunto.

2.2 Técnicas de Plágio

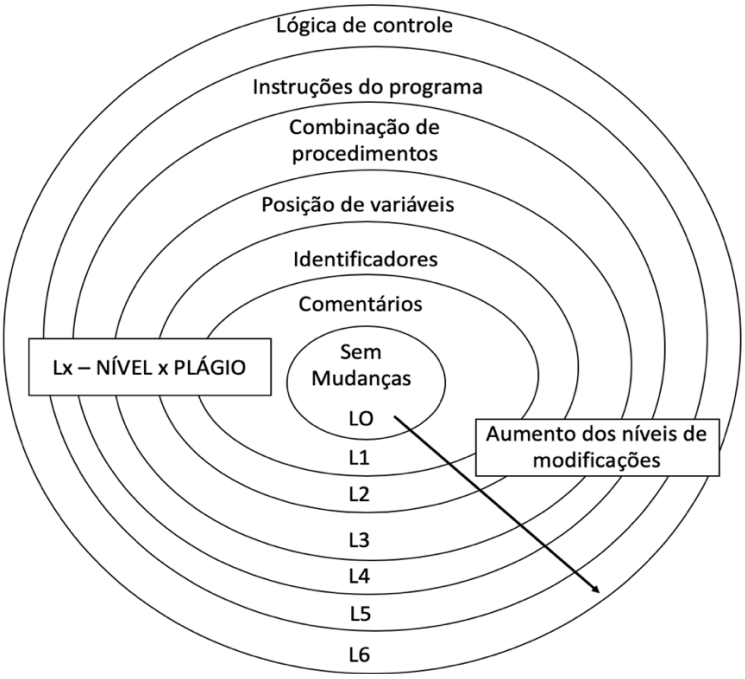
A capacidade de tentar esconder a semelhança entre códigos-fonte copiados aumenta de maneira proporcional ao conhecimento do estudante sobre a linguagem de programação estudada. A experiência em laboratório e/ou sala de aula revela que, nas primeiras aulas de programação, não é raro encontrar semelhanças entre trechos de códigos de alunos que fizeram a cópia, revelando certa ingenuidade na tentativa de dissimular a ação.

Em geral, os poucos alunos que possuem a intenção de copiar ilicitamente o trabalho dos colegas o fazem principalmente por não entender completamente como programar. Além disso, as modificações que eles fazem - uma vez detectadas - são geralmente suficientemente óbvias (JOY; LUCK, 1999).

Se grandes modificações são feitas em um código a ponto de alterar radicalmente a estrutura do programa, é difícil, senão impossível, identificar a origem do mesmo. No entanto, é pequena a possibilidade de isso ocorrer em disciplinas introdutórias de programação, uma vez que o tempo e o esforço gastos para tais modificações seriam de volume similar àqueles envolvidos na implementação do programa em si (JOY; LUCK, 1999).

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.

Figura 2.1 - Técnicas de similaridade



Fonte: Adaptado de Faidhi; Robinson, 1987.

Em programação, pode-se definir como técnicas de plágio as possíveis alterações empregadas para encobrir a cópia entre códigos. Faidhi e Robinson (1987) estão entre os primeiros autores a listar tais modificações.

Na Figura 2.1, são ilustrados os níveis de técnicas utilizadas para esconder a cópia, de acordo com a modificação empregada. O primeiro nível corresponde a códigos sem modificação, e o último, por sua vez, refere-se ao tipo de modificação mais complexa: alteração lógica.

Whale (1990) lista as seguintes técnicas de modificações mais empregadas para se omitir a cópia:

1. Alteração de comentários e/ou formatação;
2. Modificação de nomes de identificadores;
3. Alteração da ordem de operandos e expressões;
4. Alteração de tipos de dados;
5. Substituição de expressões por equivalentes;
6. Adição de instruções redundantes ou variáveis (exemplo: adição de laços que não tem nenhuma função no código);
7. Alteração na ordem de instruções que não alteram o funcionamento;

8. Alteração das estruturas de *loop* (exemplos: "*do*" por "*while*"; "*while*" por "*for*" entre outras);
9. Alteração das estruturas das instruções de seleção (exemplo: trocar "*if-else*" por "*switch-case*");
10. Substituição de chamadas a funções pelo respectivo conteúdo;
11. Adição de instruções que não influenciam o fluxo do programa (por exemplo, funções de impressão);
12. 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 das estruturas 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:

1. Reescrita, adição ou omissão de comentários;
2. Alteração de formatação;
3. Modificação de nomes de identificadores;
4. Alteração do número de linhas, para linguagens como o FORTRAN.

Já as estruturais são as seguintes:

1. Substituição de estruturas de *loops* (por exemplo, "*for*" por "*while*");
2. Substituição de "*ifs*" cascateados por "*switch-case*";
3. Alteração na ordem de instruções que não afetam o funcionamento do programa;
4. Substituição de múltiplas chamadas de procedimentos por chamadas de uma função única, ou vice-versa;
5. Substituição de chamada de procedimento pelo conteúdo do procedimento;
6. Alteração na ordem de operandos (por exemplo, "x<y" por "y>x").

Mozgovoy (2006) compilou a seguinte lista das transformações tipicamente tomadas pelos alunos para ocultar a similaridade:

1. Alteração de comentários (reescrita, adição, alteração se sintaxe e omissão);
2. Alteração de espaços em branco e *layout*;
3. Modificação de nomes de identificadores;
4. Reordenação de blocos de código;
5. Reordenação de instruções dentro de blocos de códigos;
6. Alteração na ordem de operadores/operandos em expressões;
7. Mudança de tipos de dados;
8. Adição de instruções redundantes ou variáveis;
9. Substituição de estruturas de controle por equivalentes ("*while*" por "*do-while*", "*if*" por "*switch-case*");
10. Substituição da chamada a uma função pelo conteúdo da mesma.

Percebe-se que as técnicas listadas e enumeradas por cada autor apresentam muitas semelhanças. Para o contexto deste trabalho, foi importante fazer uma seleção das principais técnicas, segundo a literatura, a fim de se construir uma base de dados onde se sabe a *priori* o *status* quanto à similaridade dos pares implementados. Tal seleção será apresentada na Seção 4.1.1.

A partir dessa base, com os códigos que representam as principais modificações, foi possível elaborar um conjunto de testes, abordados no Capítulo 4.

2.3 Técnicas de análise de similaridade

De acordo com Lancaster (2003) e Clough (2000), as ferramentas de detecção de similaridade podem ser divididas em duas classes gerais: baseadas em atributos ou em estrutura. Nas técnicas baseadas em atributos, a ideia é criar uma impressão digital para cada documento da coleção; já nas técnicas baseadas em estrutura, a estrutura do programa é representada através de *tokens* que são, então, comparados.

Outra categoria que pode ser adicionada a outras duas é a baseada em texto que, apesar de toda sintaxe de codificação ser ignorada, pode algumas vezes ser efetiva na comparação de códigos, como mostrado por Burrows *et al.* (2007).

Já ZHAO *et al*. (2015) classificam como as mais populares técnicas de análise de similaridade as baseadas em texto, *tokens* e árvores de análise sintática, as quais são mais detalhadamente descritas a seguir.

2.3.1 Técnicas baseadas em texto

São várias as técnicas de detecção de similaridade que se baseiam em texto puro (BAKER, 1993; DUCASSE; RIEGER; DEMEYER, 1999; JOHNSON, 1993), sendo o código-fonte considerado nelas uma sequência de linhas, e cada linha, por sua vez, uma sequência de caracteres (ROY; CORDY, 2007). Basicamente, se, ao comparar dois fragmentos de código entre si, for encontrada uma sequência de mesmo texto em máxima extensão possível, eles serão retornados como um par com alta similaridade.

Por se tratar de texto puro, as similaridades detectadas não correspondem a elementos estruturais da linguagem, sendo, portanto, uma abordagem léxica. Geralmente, se faz pouca ou nenhuma normalização no código antes de ser realizada a comparação.

Algumas das transformações que podem ser feitas no texto em algumas abordagens são: remoção de comentários e de espaços em branco, que não agregam nenhuma informação relevante para a comparação, e normalizações básicas, como remoções de literais.

Apesar de muitas semelhanças, existem diferenças óbvias entre detectar similaridade em linguagem de programação e texto escrito (CLOUGH, 2000). Provavelmente, a diferença mais importante é que o código-fonte é mais estruturado do que a linguagem natural. As linguagens de programação possuem regras restritas de formação de instruções, o que torna mais provável que duas linhas sejam semelhantes.

2.3.2 Técnicas baseadas em tokens

Nessa abordagem de detecção de similaridade, o código-fonte é transformado em uma sequência de *tokens*, que correspondem a qualquer elemento significativo para a análise de similaridade, podendo ser palavras, frases ou símbolo. Essa sequência é, então, processada, a fim de se encontrar subsequências duplicadas de *tokens*, que são retornadas como similares (ROY; CORDY, 2007).

É uma abordagem mais robusta contra alterações de código, como formatação e espaçamento, quando comparada a abordagens baseadas em texto. Com essa técnica, é possível descartar informações desnecessárias, tais como nomes de variáveis e de métodos, sendo isso considerado por Cornic (2008) a principal vantagem dessa abordagem.

2.3.3 Técnicas baseadas em árvores

Nessa técnica, o código é transformado em uma árvore de sintaxe abstrata (AST) de acordo com a estrutura da linguagem de programação usada nele. A ordem estabelecida entre os nós da árvore é baseada na hierarquia inerente à própria linguagem de programação. Segundo Feist *et al*. (2016), uma AST é uma estrutura de dados em árvore formada pela quebra das construções sintáticas do código-fonte.

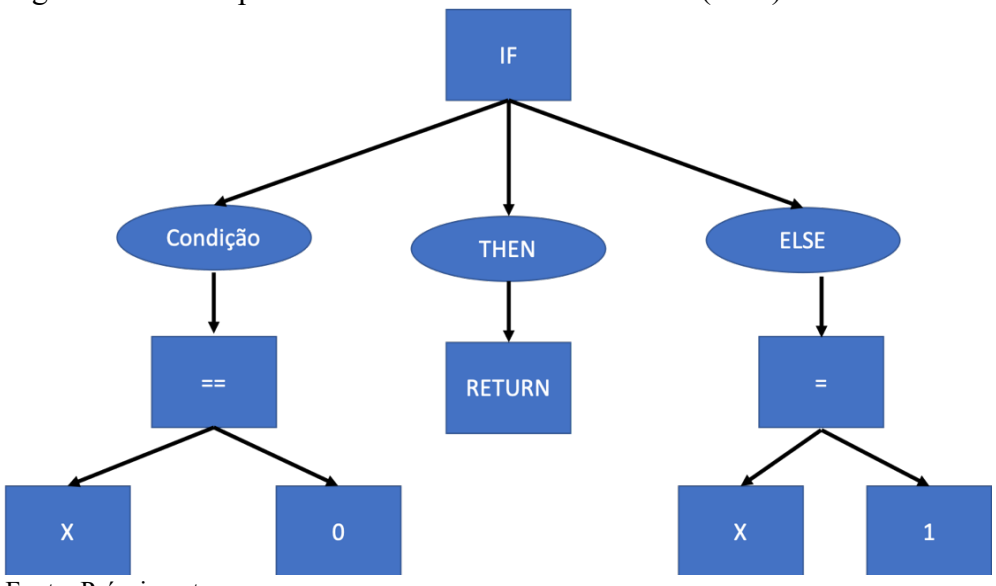
Assim, seus nós representam pedaços de código sintaticamente válidos. Exemplos de nós são operadores matemáticos, chamadas de funções, ou estruturas de programação, e de folhas, sendo variáveis ou constantes (CORNIC, 2008). Em consequência disso, a estrutura do código é capturada de forma abstrata e tratável. O trecho de código ilustrado na Figura 2.2 é representado na Figura 2.3 em forma de AST.

Figura 2.2 - Trecho de código na linguagem C.

```
if(x==0){
  return;
}
else{
  x=1;
}
```

Fonte: Próprio autor.

Figura 2.3 - Exemplo de uma árvore sintática abstrata (AST)



Fonte: Próprio autor.

Em uma AST, o significado da sentença está diretamente relacionado a sua estrutura sintática representada na árvore. Dessa forma, a posição de cada nó na árvore tem relação direta com os demais estabelecidos na estrutura.

Para detectar similaridade utilizando essa abordagem, subárvores semelhantes são procuradas em ASTs de dois códigos. É possível torná-la mais robusta quanto à renomeação de variáveis e constantes, eliminando os nomes destas. Entretanto, Cornic (2008) aponta duas desvantagens dessa abordagem: ASTs são vulneráveis à inserção de código ou a códigos reestruturados, e sua complexidade é elevada. Isso se dá pelo fato de que a procura de uma subárvore semelhante entre duas árvores pode exigir a comparação entre todas as subárvores, tornando essa técnica impraticável para grandes conjuntos de dados.

2.4 Ferramentas para detecção e análise de similaridade

Nas técnicas de tokenização, o código é convertido em um conjunto de *tokens*. A detecção da similaridade consiste, portanto, na comparação entre as sequências de *tokens* obtidas objetivando encontrar a sequência comum entre os códigos comparados (RAGKHITWETSAGUL; KRINKE; CLARK, 2018).

Entre as ferramentas disponíveis para a detecção e análise de similaridade utilizando *tokens,* e que realizam pré-processamento antes da comparação, são utilizadas neste trabalho o Sherlock N-overlap (MACIEL, 2014), o SIM (GRUNE, 2018a), o MOSS (BOWYER; HALL, 1999) e o JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2002), , sendo todas descritas nas subseções de 2.4.1 a 2.4.4.

2.4.1 Jplag

O JPlag é uma ferramenta desenvolvida em Java para análise de similaridade entre códigos (NOVAK, 2016; RAGKHITWETSAGUL; KRINKE; CLARK, 2018). Ele é uma das principais ferramentas utilizadas para a comparação de códigos-fonte e possui suporte para várias linguagens de programação como C, C++, C# e Java.

O algoritmo utilizado na detecção é uma versão otimizada do *Running Karp-Rabin Greedy String Tiling* (RKR-GST) (WISE, 1996).

Segundo Prechelt *et al* (2002), o algoritmo incorporado no JPlag trabalha em duas etapas:

i. O código-fonte é submetido a um analisador que interpreta as estruturas de linguagem, pré-processa o código e gera *tokens* correspondentes, como ilustrado na Figura 2.4;

ii. Os *tokens* gerados são comparados em pares (um em cada um dos dois documentos de código) e o índice de similaridade é calculado.

Figura 2.4 - 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; Philippsen, 2002.

2.4.2 SIM

O sistema SIM (GRUNE, 2018b, 2018a; GRUNE; HUNTJENS, 1989) é baseado em um algoritmo personalizado que identifica *substrings* comuns encontradas em dois documentos. As únicas documentações encontradas sobre o funcionamento do SIM são o próprio código-fonte e uma descrição resumida realizada pelo autor (GRUNE, 2018a).

O algoritmo é insensível à ordem em que as subsequências aparecem. Para acelerar o processo, apenas as *substrings* de um determinado tamanho mínimo são consideradas, enquanto as outras são descartadas.

Esta ferramenta pode ser aplicada às seguintes linguagens: C, Java, Pascal, Modula-2, Lisp, Miranda e linguagem natural, e para isso é gerado um binário para cada linguagem suportada.

2.4.3 MOSS

O MOSS (*Measure of Software Similarity*) é acessado exclusivamente por meio de um *webservice*, disponibilizado por uma universidade da Califórnia, nos Estados Unidos, sendo necessário requisitar autorização para utilizá-lo (BOWYER; HALL, 1999; SCHLEIMER; WILKERSON; AIKEN, 2003).

No MOSS, o código-fonte é convertido em *tokens* e o algoritmo de *Winnowing* (SCHLEIMER; WILKERSON; AIKEN, 2003) é usado para selecionar um subconjunto de *hashes*, relativo aos *tokens* gerados, para criar um tipo de assinatura, como uma impressão digital, representando a estrutura do documento. O cálculo da distância entre as impressões

digitais determina quão semelhantes são os arquivos correspondentes (JONYER; APIRATIKUL; THOMAS, 2005; MOZGOVOY, 2006).

Esta ferramenta pode ser usada com os seguintes idiomas: C, C ++, Java, Pascal, Ada, LISP, entre outros.

2.4.4 Sherlock N-overlap

A ferramenta Sherlock N-overlap (MACIEL, 2014) é baseada no Sherlock (PIKE; LOKI, 2017). Para implementá-lo, foi adicionado um suporte para normalizações (técnicas de pré-processamento) e o coeficiente de similaridade original foi substituído pelo *overlap* (DIPTI *et al*., 2007).

Como uma das contribuições desse trabalho é focada nessa ferramenta, ela é descrita em maiores detalhes nas próximas subseções.

2.4.4.1 Parâmetros de configuração

O Sherlock original (PIKE; LOKI, 2017) compara os documentos sem antes realizar nenhum tipo de pré-processamento, apenas convertendo texto em assinaturas (*tokens*). Para isso, três parâmetros precisam ser configurados:

1. *Zerobits* (z): controla o número de assinaturas que serão comparadas. Quando *zerobit* é igual a 0, todas as assinaturas geradas serão comparadas. Quanto menor for esse número, mais exata será a comparação, porém será mais custosa computacionalmente. Além disso, pequenas modificações podem resultar em índices baixos de similaridade.
2. Número de *tokens* (n): define o número de *tokens* utilizados para formar uma assinatura.
3. *Threshold* (t): indica o percentual de tolerância para a similaridade. Apenas as semelhanças maiores que esse limiar serão mostradas.

Nenhuma alteração foi feita nesses conjuntos de parâmetros relativos ao Sherlock (PIKE; LOKI, 2017). Dessa forma, Sherlock N-overlap é configurado da mesma maneira que no original.

2.4.4.2 Geração de Assinaturas

O Sherlock original gera as assinaturas identificando inicialmente as estruturas contíguas (*tokens*) no código. Os *tokens* estão sublinhados no código ilustrado na Figura 2.5.

Depois que os *tokens* são identificados, o código é convertido em um conjunto de assinaturas calculadas por uma função *hash*. Uma assinatura corresponde à representação numérica de um conjunto de **n** *tokens*, de acordo com a configuração utilizada para a comparação. No final desta etapa, o código-fonte é representado por um vetor de assinaturas selecionadas que serão usadas na próxima etapa, no processo de comparação.

Figura 2.5 - Código destacando os tokens

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

Fonte: Maciel, 2014.

Zerobit é um parâmetro incluído no Sherlock original usado para acelerar o processo e controlar a granularidade da comparação. Consequentemente, somente as assinaturas que, ao final de sua representação binária, contêm o número de zeros correspondentes ao valor definido para *zerobits* são consideradas para o conjunto de comparação, enquanto as outras são descartadas.

Por exemplo, com o número de *tokens* (**n**) igual a três, uma assinatura é gerada para cada sequência de três *tokens*. Para *zerobit* (**z**) igual a 2, apenas assinaturas contendo pelo menos dois zeros consecutivos ao final de sua representação binária são selecionadas.

Ao contrário da linguagem natural, a maneira como uma palavra é usada em um código-fonte da perspectiva do Sherlock é relevante. A compilação e interpretação de linguagens de programação, em muitas situações, não são afetadas pela presença ou ausência de espaços em branco entre os elementos da estrutura. Por exemplo, para o compilador, é irrelevante escrever:

```
for (i=0;i<n;i++){result+=v[i];}                    (1.1)
for ( i = 0 ; i < n ; i++ ) { result + = v [ i ] ; }   (1.2)
```

No entanto, para o Sherlock, o número de estruturas identificadas neste segmento é diferente e, assim, os índices de similaridade entre esses dois códigos serão comprometidos. Consequentemente, o agrupamento ou segmentação de algumas estruturas do código pode favorecer a comparação feita pelo Sherlock. Desta forma, estudos foram realizados procurando a maneira mais eficaz de reorganizar essas estruturas no código-fonte que será comparado (FRANÇA; MACIEL; SOARES, 2013).

As regras para a organização dessas estruturas criaram algumas técnicas de pré-processamento de código que foram adicionadas ao Sherlock N-overlap e estão descritas na Subseção 2.4.4.4.

2.4.4.3 Comparação de Assinaturas

As assinaturas selecionadas de um código-fonte **A** são comparadas, uma a uma, àquelas selecionadas de um código-fonte **B**. Isso é feito para todos os pares formados por uma assinatura de **A** e outra de **B**.

A similaridade entre dois códigos no Sherlock original é calculada através do coeficiente de similaridade de Jaccard, que, por sua vez, é definido como a porcentagem de similaridade entre as assinaturas nos códigos **A** e **B**, conforme formulado na Equação 2.1, em que **a** é o número de assinaturas encontradas exclusivamente em **A**, **b** o número de assinaturas encontradas exclusivamente em **B** e **c** é o número de assinaturas semelhantes encontradas em ambos os conjuntos (**A** e **B**).

$$\mathrm{Sim}\,(\mathrm{A},\mathrm{B}) = 100\ \mathrm{x}\frac{\mathrm{c}}{(\mathrm{a}+\mathrm{b}+\mathrm{c})} \qquad (2.1)$$

O coeficiente de similaridade de Jaccard, no entanto, é sensível ao aumento de elementos incomuns nos conjuntos, elevando o denominador e reduzindo a similaridade entre os conjuntos. Comparando os códigos-fonte, este tipo de coeficiente não é, portanto, efetivo em detectar as semelhanças baseadas na inclusão de códigos inúteis no arquivo, que são adicionados como um instrumento para esconder uma semelhança.

Para minimizar este problema, o coeficiente de similaridade utilizado pelo Sherlock é o *overlap*, demonstrado na Equação 2.2. (DIPTI *et al*., 2007)

$$\text{Overlap}\,(A, B) = \frac{|A \cap B|}{\min(|A|,|B|)} \quad (2.2)$$

Esse coeficiente foi escolhido porque é capaz de identificar o número de assinaturas semelhantes (numerador) em comparação com o menor dos dois conjuntos de assinaturas (denominador), o que reduz a sensibilidade à inclusão de estruturas e instruções inúteis.

2.4.4.4 Representação Normalizada dos Códigos

Para aperfeiçoar a comparação feita pelo Sherlock N-overlap, foi estabelecido um conjunto de regras de pré-processamento. A implementação dessas regras, para uma ou outra linguagem de programação, é completamente separada do algoritmo de comparação e é aplicada como um pré-processamento.

A nomenclatura "normalização" foi adotada para nomear esses conjuntos de regras. Primeiro as normalizações 1, 2 e 3 foram criadas (FRANÇA; MACIEL; SOARES, 2013), sendo que cada normalização inclui as modificações anteriores. Dessa forma, a normalização 2 aplica as regras estabelecidas pela normalização 1 e assim por diante.

Experimentos com o Sherlock N-overlap mostraram que as normalizações de 1 a 3 apresentam melhores resultados do que submissões sem pré-processamento; entretanto, tais resultados ainda foram inferiores aos encontrados pelas demais ferramentas utilizadas para comparação (FRANÇA; MACIEL; SOARES, 2013).

Portanto, foram adotadas regras mais invasivas para obter um pré-processamento mais agressivo, mas permitindo que as características intrínsecas ao estilo e à lógica da programação individual permanecessem intactas. Seguindo esse raciocínio, foi concebida a normalização 4 (FRANÇA; MACIEL; SOARES, 2013). Esta, por sua vez, é a mais invasiva, preservando apenas operadores e pontuadores no código. Nesta modificação, a estrutura do código-fonte é parcialmente preservada, mas sem, por exemplo, instruções, declarações e variáveis.

Stamatatos (2011) utilizou um método análogo ao da normalização 4, no domínio da linguagem natural, em que são retiradas dos textos a serem comparados todas as palavras que não compõem a lista das cinquenta palavras mais comuns do inglês.

As regras adotadas pelas normalizações de 1 a 4 são as seguintes:

A. Regras da normalização 1:

i. Eliminação de linhas e espaços vazios;
ii. Eliminação de todos os comentários;
iii. Eliminação de referências a arquivos externos (bibliotecas);
iv. Inclusão (ou eliminação) de espaço em branco entre expressões, declaração de variáveis e outras estruturas.

B. Regras da normalização 2:
   i. Aplicação de todas as regras da normalização 1;
   ii. Eliminação de todos os caracteres entre aspas.

C. Regras da normalização 3:
   i. Aplicação de todas as regras da normalização 2;
   ii. Eliminação de todos os valores literais e variáveis.

D. Regras da normalização 4:
   i. Aplicação de todas as regras da normalização 3;
   ii. Eliminação de todas as palavras reservadas;
   iii. Adição de regras específicas para inserir ou remover espaços em branco entre os caracteres.

O conjunto de regras específicas para inserir ou remover espaços em branco (regra 3 da normalização 4), foi estabelecido empiricamente através de observações e testes realizados em submissões ao Sherlock N-overlap (FRANÇA; MACIEL; SOARES, 2013). Essas regras ajudam a melhorar a aquisição de *tokens* e a comparação de assinaturas.

Na Tabela 2.1, a coluna à esquerda apresenta o código original, sem normalização. Já a coluna da direita mostra o mesmo código quando submetido à normalização 4.

Tabela 2.1 − Exemplo de aplicação da normalização 4 em um código desenvolvido na Linguagem C.

Código Original	Código Normalizado
for (i=1;i<= n;i++){	(=; <=; ++) {
if(i%3== 1){	(%==) {
printf("answer:%d", v[i]/x+z);	(, [] / +);
}	}
}	}

Fonte: Próprio autor.

Quando a normalização 4 é aplicada, somente operadores e pontuadores são mantidos. Todos os outros caracteres, incluindo as palavras reservadas (*stopwords*), são excluídos do código que será comparado. Para filtrar os códigos e remover os caracteres, são usadas expressões regulares.

Todas as regras apresentadas nas normalizações são implementadas usando expressões regulares e, em alguns casos, dependem da linguagem de programação (como remover comentários). Quaisquer outras modificações necessárias devem ser adaptadas para serem utilizadas, a depender da linguagem de programação.

2.5 Comparação de Índices de Similaridade

Os índices de similaridade resultantes da aplicação de diferentes ferramentas elencadas não são facilmente comparáveis uns aos outros. Isso se deve ao fato destas usarem medidas de similaridade que, muitas vezes, consideram propriedades distintas, ou de executarem pré-processamento que ressaltam e/ou omitem elementos desiguais antes da comparação.

Isso faz com que os índices de similaridade não sejam absolutos, mas que sejam capazes de captar a semelhança entre os códigos comparados de forma relativa, dependente da capacidade inerente à técnica em identificar semelhanças. A qualidade de uma ferramenta para a análise de similaridade, entretanto, pode ser avaliada com base em sua aplicação a um conjunto de pares de códigos em que um deles é previamente modificado sob diferentes critérios. Dessa forma, é possível verificar quando uma ferramenta falha e outra não, por exemplo, para uma técnica de plágio.

Nas próximas subseções, são apresentados dois métodos utilizados para a comparação entre ferramentas elencadas e as propostas. Neste trabalho, denomina-se “Método Tradicional” aquele que se utiliza dos cálculos da precisão e revocação absolutos. Este método é amplamente utilizado na literatura específica de detecção de plágio entre códigos-fonte, sendo utilizado quando se conhece a *priori* a situação de similaridade dos códigos comparados. O “Método de Conformidade”, por sua vez, proposto em Franca *et al*. (2018) e em Maciel (2014), foi criado para calcular a precisão e a revocação de forma relativa, com base em um conjunto de ferramentas de referência usadas como oráculos.

2.5.1 Método Tradicional

Em geral, para fazer a comparação, são utilizadas medidas de precisão e revocação, bem como, a média harmônica entre elas, conforme encontrado nas obras de Prechelt, Malpohl e Philippsen (2002), Burrows, Tahaghoghi e Zobel (2007), Duric e Gasevic (2013), Cosma e Joy (2012) e Ohmann (2013).

Antes de se discutir as métricas geralmente usadas ao se comparar códigos-fonte, é necessário compreender os possíveis resultados retornados pelas ferramentas de comparação, ilustrado pela Tabela 2.2.

Tabela 2.2 - 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 recupedaros (D) (Falso Negativo)	Pares relevantes recupedaros (A) (Verdadeiro Positivo)
Pares não relevantes não recupedaros (C) (Verdadeiro negativo)	Pares não relevantes recuperados (B) (Falso positivo)

Fonte: Próprio autor.

Considerando casos identificados como semelhantes, eles podem ser compostos daqueles que são de fato semelhantes (A) e aqueles que não são, mas foram considerados pelo algoritmo de classificação como similares, doravante chamados de falsos positivos (B).

Por outro lado, nos casos em que não são detectadas semelhanças, pode haver aquelas que não são semelhantes (C) e aquelas que são semelhantes, mas que não foram consideradas pelo algoritmo como similares, que chamaremos de falsos negativos (D).

O sistema ideal é aquele que identifica semelhança, considerando um limiar percentual, para todos os verdadeiros positivos e somente para eles (A). Juntos, esses parâmetros determinam a eficácia de um algoritmo através da precisão e revocação.

A precisão (P), $P \in [0,1]$, é a proporção de pares relevantes de um código-fonte recuperados do total de códigos recuperados, incluindo os falsos positivos, e é calculada de acordo com a Equação 2.3. Dessa forma, o valor da precisão é 1 quando não há falsos positivos para um determinado limiar de busca.

$$P = \frac{\text{Total de pares relevantes recuperados (A)}}{\text{Total de todos os pares recuperados (A+B)}} \tag{2.3}$$

Revocação (R), $R \in [0,1]$, por sua vez, é a proporção de pares de códigos que foram identificados como semelhantes em um total que também considera falsos negativos, e é calculada de acordo com a Equação 2.4. Aqui, o valor de revocação é 1 quando todos os pares semelhantes, para um determinado limite, são detectados completamente ou, em outras palavras, quando não há nenhum falso negativo.

$$R = \frac{\text{Total de pares relevantes recuperados (A)}}{\text{Total de pares relevantes (A+D)}} \quad (2.4)$$

No entanto, se uma ferramenta recuperar todos os pares semelhantes (R = 1), isso não significa que a ferramenta é ideal, porque pode haver falsos positivos. Além disso, se uma ferramenta recuperar apenas pares semelhantes (P = 1), isso não significa que a ferramenta é ideal, porque pode haver falsos negativos. Assim, o desempenho geral de uma ferramenta é medido pela combinação de precisão e revocação.

Uma boa maneira de obter um único valor é calculando a média harmônica (DURIC; GASEVIC, 2013; OHMANN, 2013). A média harmônica entre precisão e revocação é calculada de acordo com a Equação 2.5, em que $F \in [0, 1]$.

$$F = 2 \text{ x } \frac{P \text{ x } R}{P+R} \quad (2.5)$$

Quão mais próximo o valor de F for de 1, melhor será o desempenho da ferramenta.

2.5.2 Método de Conformidade

Quando o estado de cada par de códigos não é conhecido a priori, não é possível contar os verdadeiros positivos, falsos negativos e falsos positivos diretamente. Portanto, para avaliar as ferramentas de detecção de similaridade em tais cenários, é necessário encontrar a conformidade dos resultados entre as ferramentas de referência escolhidas, utilizando-as como uma espécie de oráculo.

Não é incomum que uma ferramenta tenha um desempenho melhor do que outra, em situações diferentes (MOZGOVOY; KARAKOVSKIYZ; KLYUEV, 2007); portanto, neste método, nenhuma ferramenta isolada deve ser usada como oráculo. O método de conformidade leva em consideração a concordância entre os resultados entre três ferramentas de referência e, além disso, a ferramenta em análise.

Deve-se notar que, com conhecimento prévio de similaridade ou não similaridade de cada par de códigos, as medidas das Equações 2.3, 2.4 e 2.5 são absolutas. No entanto, no método de conformidade, devido à ausência desta informação, a precisão e a revocação são calculadas de forma relativa, adaptando as Equações 2.3 e 2.4 com as seguintes alterações:

1. Número de ocorrências de similaridade (NOS) - número de pares contados como similares;
2. Número de ocorrências isoladas de similaridade (NOIS) - número de pares em que a ferramenta foi a única a considerar uma semelhança;
3. Número de ausências isoladas de similaridade (NAIS) - número de pares em que uma ferramenta foi a única a **NÃO** contar semelhança.

Esses parâmetros são calculados após a ferramenta ter sido avaliada junto com os oráculos, considerando um limite predefinido.

Faz-se necessário obervar que as duas últimas métricas são alteradas de acordo com o resultado das outras ferramentas. Desse modo, se o número de ausências isoladas for demais, o desempenho de um determinado algoritmo em relação às ferramentas analisadas será considerado pior.

Uma vez que essas quantidades tenham sido obtidas, o número total de verdadeiros positivos (NTVP) para cada ferramenta usada é calculado, como descrito na Equação 2.7.

$$\mathrm{NTVP} = \mathrm{NOS} - \mathrm{NOIS} \tag{2.7}$$

No método de conformidade, cada ferramenta existente no conjunto tem sua precisão e revocação relativas calculadas separadamente. A Equação 2.8 é usada para calcular a precisão relativa, chamada de precisão de conformidade (P_{conf}). O numerador considera o número total de verdadeiros positivos da ferramenta analisada, conforme Equação 2.7, e o denominador, por seu turno, o maior valor entre as ferramentas analisadas do número total de verdadeiros positivos, somado este ao número de ocorrências isoladas de similaridade. Assim, o $P_{conf} \in [0, 1]$.Note-se que P_{conf} é igual a 1 quando a própria ferramenta possui o maior número total de verdadeiros positivos e a mesma não apresenta nenhuma ocorrência isolada.

$$\mathrm{P}_{conf} = \frac{\mathrm{NTVP}}{\max(NTVP)+NOIS} \tag{2.8}$$

A revocação relativa, chamada revocação de conformidade (R_{conf}), é calculada de acordo com a Equação 2.9.

$$R_{conf} = \frac{\text{NTVP}}{\max(NTVP)+NAIS} \quad (2.9)$$

No cálculo da revocação apresentado na Equação 2.9, o numerador considera o número total de verdadeiros positivos da ferramenta analisada, conforme Equação 2.7, e o denominador, por outro lado, o maior valor entre as ferramentas analisadas do número total de verdadeiros positivos, somado este ao número total de ausências isoladas de similaridade. $R_{conf} \in [0, 1]$. Destaque-se que R_{conf} é igual a 1 quando a ferramenta analisada não apresenta ausências isoladas e possui o maior número de verdadeiros positivos.

A média harmônica continua sendo calculada de acordo com a Equação 2.5. Os valores de P_{conf} e R_{conf} são usados para este cálculo. Quanto mais próximo o valor F for de 1, melhor será o desempenho da ferramenta de detecção.

2.6 Estudo de Imagens Digitais para representação de códigos-fonte

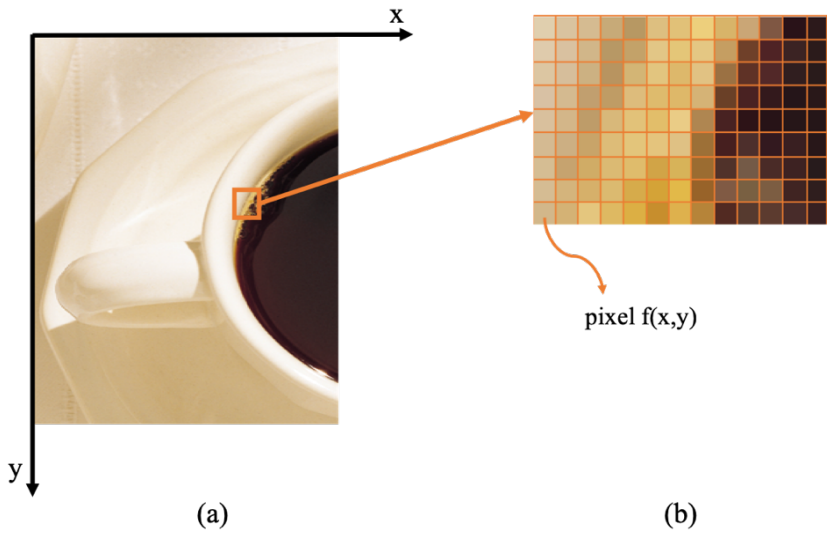
Nesse trabalho, além das normalizações de código-fonte apresentadas, estudou-se a representação dos códigos a serem comparados em outros domínios, como o das imagens. Com esse objetivo, alguns fundamentos sobre imagem digital e visualização são discutidos nesta seção.

2.6.1 Imagens Digitais

Uma imagem pode ser determinada por f(x,y), uma função bidirecional em que x e y são as coordenadas do plano e f determina a amplitude em qualquer par de coordenadas. Quando essa imagem é composta por valores finitos de f e de qualquer ponto (x,y), dizemos que se trata de uma imagem digital (GONZALEZ; WOODS, 2006).

Dessa forma, uma imagem digital é composta por um número finito de elementos com localização e valores específicos. Tal elemento, em sua unidade, é conhecido como *pixel* (*picture element*). A Figura 2.6 ilustra tais conceitos.

Figura 2.6 - Imagem digital



Legenda: (a) Imagem digital e (b) sua a representação através de uma matriz de pixels de 13x9 de uma região de interesse.
Fonte: Adaptado de Gonzalez; Woods, (2006, Adaptado).

No caso de imagens digitais coloridas, cada *pixel* é descrito por meio de um conjunto de propriedades como, por exemplo, matiz, saturação, brilho e a quantidade total de *pixels* que determina a resolução.

Assim, a cor de cada *pixel* é representada por um ponto no espaço definido pelo modelo de cor (GONZALEZ; WOODS, 2006). O modelo de cores RGB, comumente utilizado, é discutido na Seção a seguir.

2.6.2 O Modelo RGB de Cores

Um modelo de cor é um sistema de coordenadas tridimensionais em que cada cor é representada por um ponto nesse espaço (GONZALEZ; WOODS, 2006). Existem diversos modelos de cores, sendo o RGB (Red, Green, Blue) um dos mais conhecidos.

No modelo RGB, cada cor é descrita pela intensidade das três cores básicas componentes: vermelho, verde e azul. Assim, para representar uma cor, atribui-se um valor de 0 a 255 para cada um dos três componentes do modelo.

A cor branca é definida com os valores (255, 255, 255), a preta pelos valores (0, 0, 0), o vermelho puro é dado pelos valores (255, 0, 0), e assim por diante. Quando duas das três cores correspondentes do modelo se sobrepõem, são formadas as cores secundárias, conforme apresentado na Figura 2.7.

Por exemplo, quando as cores vermelha e verde se sobrepõem, a cor amarela é criada. Com a variação das componentes do modelo RGB, é possível se obter cerca de 16 milhões de cores diferentes.

Uma cor no modelo RGB depende dos valores de cada componente, bem como, das especificações dos dispositivos. Dessa forma, diferentes dispositivos podem ter cores com variações sensíveis ao olho humano (GONZALEZ; WOODS, 2006).

Figura 2.7 - Modelo de cor RGB.

Fonte: Gonzalez; Woods (2006).

2.6.3 Técnicas de Visualização de Dados

Os seres humanos possuem dificuldades em processar dados no formato de textos e tabelas; por isso, frequentemente, recorremos a meios visuais para interpretar informações a nossa volta. A visualização de dados e informações, pois, está baseada na capacidade humana de interpretação visual das informações, e através dessa interpretação, perceber relacionamentos e padrões que auxiliam na descoberta de novos conhecimentos (PERNOMIAN, 2008).

Técnicas de visualização de dados são muito importantes no processo de extração de informações, uma vez que, com o crescimento tecnológico e a automação de atividades em todas as áreas, a quantidade de informação que pode ser de interesse para a tomada de decisões aumenta muito rapidamente. Assim, ter a informação certa no momento correto é de suma importância (KEIM; KRIEGEL, 1996).

Segundo Keim (2000), a integração da visualização com o processo de decisão humana pode facilitar o entendimento dos resultados, bem como, contribuir com uma tomada de decisão mais eficaz. É importante frisar que o formato utilizado visualmente para

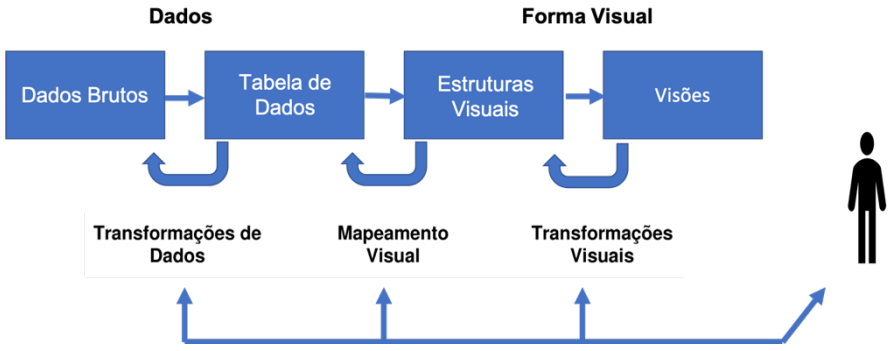
apresentar os dados pode também dificultar a análise que se deseja fazer, por isso a importância de se utilizar a técnica correta para cada tipo de apresentação.

Para que os dados possam ser apresentados visualmente, eles devem passar por um conjunto de etapas, como pode ser visto na Figura 2.8 (CARD; MACKINLAY; SHNEIDERMAN, 1999), que mostra o processo de visualização da informação, composto de três etapas:

1. Transformação dos dados, em que os dados brutos são tabulados para que possam ser utilizados pela aplicação;
2. Mapeamento virtual, no qual os dados são atribuídos às formas que serão utilizadas para serem visualizados;
3. Transformação visual, etapa que a imagem é mostrada na tela.

As setas, que vão da direita para a esquerda, indicam a retroalimentação que pode ser feita pelo ser humano.

Figura 2.8 - Modelo do Processo de Visualização



Fonte: Adaptado de Card; Mackinlay; Shneiderman (1999)

Segundo Oliveira e Minghim (1997), uma técnica de visualização envolve vários passos: a construção de um modelo empírico a partir dos dados, a seleção de um mecanismo de mapeamento e a representação gráfica das informações.

Tais técnicas se apropriam das características dos humanos em encontrar padrões visuais. Keim e Kriegel (1996) classificaram as técnicas em:

1. *Orientadas a pixel*: Nessa técnica, os atributos dos dados são mapeados em forma de pixels e coloridos de acordo com seus valores, utilizando uma classificação para tal.

2. Geométricas: essa técnica busca representar o conjunto de dados em sua projeção bidimensional. Um exemplo são as matrizes de dispersão.
3. Iconográficas: nessa técnica, cada atributo de um dado multidimensional é representado por ícones ou seus atributos, como nos ícones em formato de face utilizada por Chernoff (1973).
4. Hierárquicas: essas técnicas dividem o espaço n-dimensional em subespaços de dimensão menor. Os subespaços são apresentados de modo hierárquico.

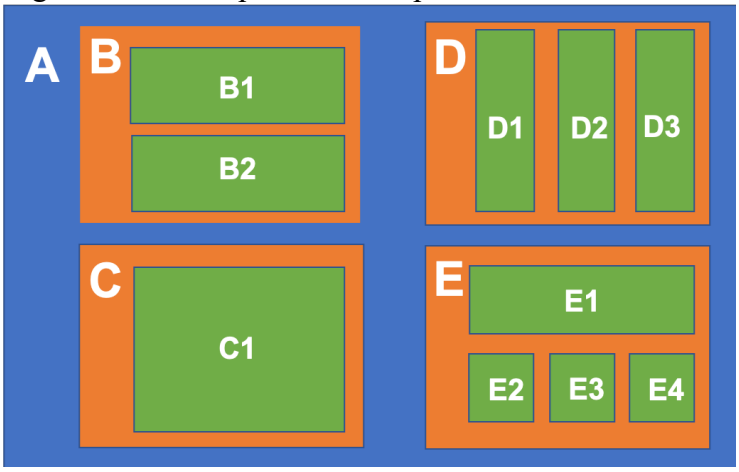
A técnica de visualização baseada em hierarquia, utilizada nesse trabalho para a conversão de um código em imagem, é detalhada a seguir.

2.6.3.1 Técnicas Hierárquicas

Essa técnica pressupõe que existe uma hierarquia nos dados e, assim, representa essa hierarquia graficamente. Este tipo de organização pode ser visto em um sistema de arquivos ou em uma estrutura de pacotes, dados e métodos de um *software*.

Um outro exemplo são os mapas de árvores, denominadas do inglês *treemaps* (SHNEIDERMAN, 1992). Os mapas de árvores são boas técnicas para se representar visualmente grandes conjuntos de dados mapeando a hierarquia através da divisão de forma recursiva de retângulos aninhados como visto na Figura 2.9.

Figura 2.9 - Exemplo de um mapa em árvores

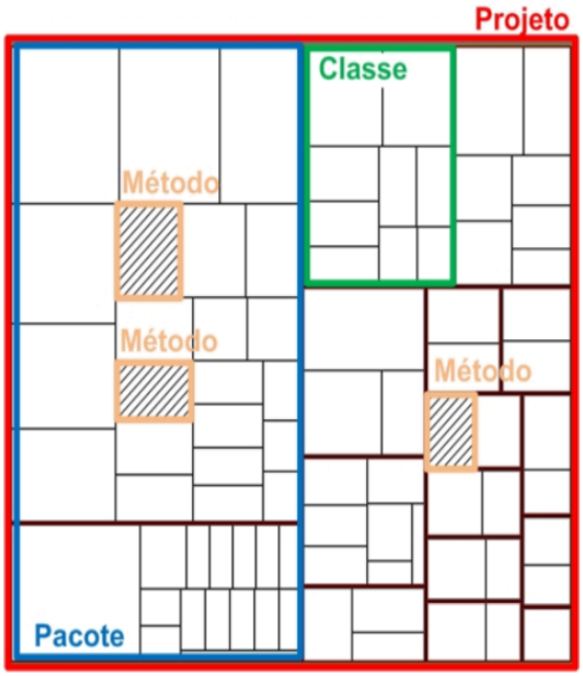


Fonte: Próprio autor.

O tamanho dos nós folha são representados geralmente em um espaço proporcional a um atributo escolhido, sendo que a utilização de cores e rótulos pode ser apresentada nos nós como forma de representar outros atributos.

Na Figura 2.10, é apresentado um exemplo do mapeamento da estrutura de uma aplicação orientada a objetos. O retângulo maior representa o projeto e as subdivisões, por sua vez, representam os pacotes, as classes e os métodos, como ilustrados abaixo.

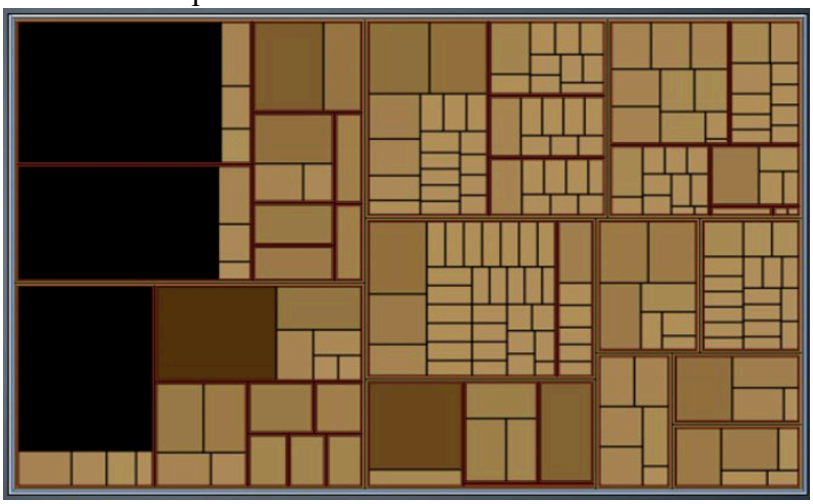
Figura 2.10 - Exemplo de um mapa de árvore de uma aplicação orientada a objetos



Fonte: CARNEIRO, 2011.

Indo além, a Figura 2.11 ilustra outra utilização dos mapas de árvores, em que o tamanho do retângulo na figura representa o número de linhas de um projeto, pacote, classe ou método, enquanto que o tom da cor marrom indica a complexidade de cada módulo (quanto mais escuro maior a complexidade).

Figura 2.11 - Exemplo de um mapa de árvore aplicado para mostrar a complexidade de um software.



Fonte: CARNEIRO, 2011.

Essa técnica possui algumas limitações, como, por exemplo, quando os dados apresentam muitas hierarquias ou, ainda, quando os valores dos atributos são ora muito pequenos − o que pode fazer com que a informação fique ilegível −, ora muito grandes, fazendo com que outras estruturas sejam comprimidas, e assim dificultando a visibilidade da informação, pois, dependendo dos valores, resulta-se em maior ou menor área ocupada na imagem.

2.6.4 Color Spectrum

Mesmo a cor sendo uma das características com maior poder descritivo, muitas vezes ela por si só não é suficiente para descrever o conteúdo de uma imagem. A organização espacial das regiões de cores também é fator importante para a percepção da imagem como um todo.

Color Spectrum é um descritor proposto por Santos (2012) que combina informações sobre a cor e sua organização espacial na imagem como base para descrever o seu conteúdo e proporcionar, dentro de um banco de imagens, a busca parcial destas. Dessa forma, é possível encontrar objetos em imagens que contenham vários objetos, além da pesquisa total, onde é encontrado imagens similares à original.

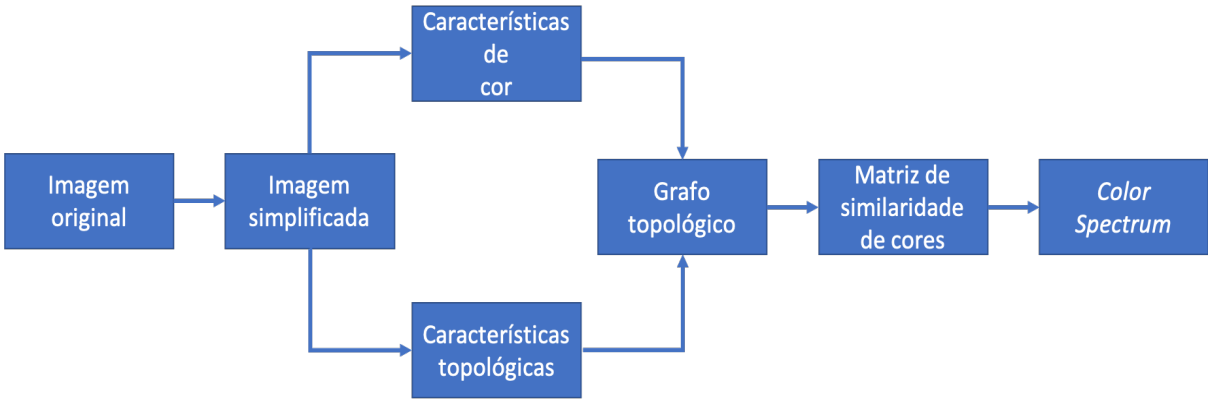
No *Color Spectrum*, primeiramente, um grafo de topologia é utilizado para descrever o relacionamento espacial entre as regiões de cor identificadas na imagem. Então, o descritor topológico é calculado a partir do grafo de topologia utilizando a teoria espectral de

grafos (CHUNG, 1997), utilizada em trabalhos como os apresentados por Oliveira, (2016); Rodrigues (2011) e Santos (2010).

A Figura 2.12 mostra um diagrama de como o *Color Spectrum* é obtido. Os passos para isso são descritos a seguir:

1. A imagem original é simplificada, ou seja, as regiões de cor que a compõem são identificadas;
2. Um grafo topológico é gerado utilizando-se da organização espacial das regiões de cor, onde os vértices representam as regiões de cor e as arestas representam as conexões entre essas regiões;
3. A partir desse grafo topológico, com as características de cor armazenadas nos vértices, é calculada a matriz de adjacência, que representa a similaridade de cor entre os vértices do grafo.
4. Os autovalores são calculados a partir da matriz de adjacência e ordenados em ordem decrescente, resultando no descritor *Color Spectrum*.

Figura 2.12 - Diagrama de blocos para a obtenção do descritor Color Spectrum.



Fonte: Adaptado de SANTOS, 2012.

Como regra de incorporação das cores à representação matricial, objeto do passo 3, é utilizada a Equação 2.10.

$$M_{uv} = \begin{cases} r & \text{se } u \text{ ou } v \text{ é o vértice raiz e } u \neq v \\ r - d(u,v) & \text{se } u \text{ ou } v \text{ são vértices conectados} \\ 0 & \text{caso contrário} \end{cases} \quad (2.10)$$

Onde **u** e **v** são vértices do grafo, **d(u,v)** é a distância Euclidiana entre seus valores RGB e **r** é uma constante calculada da seguinte forma: $r = \sqrt{\sum_{i=1}^{3} 255^2}$.

Os vértices conectados possuem similaridade de cor inversamente proporcional à distância Euclidiana entre os valores RGB das suas regiões conectadas. A exceção apresenta-se para aqueles conectados ao vértice raiz, que possuem valor de similaridade máxima (**r**), enquanto os vértices não conectados possuem o valor de similaridade mínima (**0**).

Para que a solução proposta seja capaz de suportar a pesquisa de um objeto dentro de uma imagem, o grafo principal é, então, decomposto em subgrafos, para os quais se geram novos descritores. Cada subgrafo representa um objeto distinto contido na imagem. Assim, ao final da decomposição temos, para cada objeto contido na imagem, seu respectivo descritor.

Para saber se um objeto está contido dentro de uma imagem, é calculada a distância Euclidiana entre o descritor do objeto e os descritores de cada subgrafo que compõe o grafo principal. Uma distância Euclidiana igual a 0 indica que a imagem de consulta está contida no conjunto de subpartes de uma outra imagem. Dessa forma, o problema da correspondência entre as imagens, representadas por grafos, é reduzido ao cálculo de similaridade entre vetores de características.

2.7 Considerações Finais

Considerando os objetivos perseguidos nessa tese, em que se avalia a similaridade entre códigos-fonte após a sua transformação para outro domínio, os elementos apresentados neste capítulo de fundamentação foram usados da seguinte maneira:

1. Técnicas de plágio foram estudadas com o intuito de se criar uma base de dados de códigos onde se sabe a *priori* o *status* quanto à similaridade;
2. Técnicas de normalização foram apresentadas, as quais foram tomadas como base para o desenvolvimento da normalização 5;
3. Baseado em técnicas de visualização de informações e conhecimentos sobre imagens, foi elaborado uma nova abordagem de transformação do código-fonte para representá-lo em forma de imagem digital.
4. JPlag, MOSS e SIM são as ferramentas elencadas para se comparar os resultados obtidos com as técnicas propostas neste trabalho.

5. Para comparação de resultados são utilizadas as duas abordagens apresentas: (i) o método tradicional (quando se sabe a *priori* o *status* quanto ao plágio dos códigos testados) e (ii) o método de conformidade (quando esse *status* não é conhecido a *priori*).

No Capítulo 3, são apresentadas as técnicas desenvolvidas para representar código-fonte em outro domínio em que seja possível proceder à análise de similaridade.

3 RECODIFICAÇÃO E TRANSFORMAÇÃO DE CÓDIGOS-FONTE PARA A ANÁLISE DE SIMILARIDADE

A forma de representação de um objeto tem impacto direto no desempenho da análise de similaridade com outro objeto da mesma natureza. Uma representação ruim pode ocasionar a eliminação de informações de relevância, inviabilizando o processo de comparação.

As técnicas de análise de similaridade entre objetos, sejam eles códigos, imagens ou elementos de qualquer natureza, necessitam, com frequência, de uma etapa de pré-processamento que consiste em remover informações irrelevantes ou destacar características importantes para a comparação.

Neste capítulo, são apresentadas duas novas técnicas. Inicia-se com a proposição de uma nova normalização para o Sherlock N-overlap, denominada “normalização 5”. Nela, são preservadas características léxicas importantes da linguagem de programação. Em seguida, é descrito um novo método para se identificar a similaridade entre códigos-fonte com base na transformação e comparação entre suas representações no domínio das imagens.

3.1 Representação de códigos-fonte através de normalizações

Como demonstrado por Maciel (2014), a maioria das técnicas de plágio utilizadas por alunos não afeta as estruturas do código presente após a aplicação da normalização 4. Como visto no Capítulo 2, para essa normalização foram adotadas regras invasivas, que permitem eliminar qualquer representação léxica ou literal do código, de modo a preservar apenas operadores e pontuadores.

A normalização 4 já demonstrou seu potencial para identificar a similaridade entre códigos-fonte no contexto do Sherlock N-overlap, apesar de promover uma modificação muito invasiva no código original. Percebe-se que, apesar dessa normalização ser boa para um algoritmo computacional, ela reduz significativamente a capacidade do ser humano em fazer comparação com os códigos resultantes.

Assim, investiga-se se o efeito de preservar algumas palavras reservadas da linguagem de programação, o que fornece aos olhos humanos uma melhor base comparativa, resulta também em melhores resultados para comparação no contexto do Sherlock N-Overlap.

3.1.1 Normalização 5

A normalização 5 permite relacionar palavras reservadas a serem conservadas, além do conjunto de caracteres aceitos pela normalização 4. Além disso, palavras reservadas com semântica parecidas (ex.: *while* e *for*) são convertidas para uma palavra de agrupamento comum. O motivo é impedir a geração de diferentes assinaturas distintas devido a diferenças nessas palavras, evitando a consequente redução dos índices de similaridade na comparação dos códigos normalizados. Nos testes desenvolvidos, foram utilizadas palavras de agrupamento, como os elencados na Tabela 3.1.

Tabela 3.1 - Exemplo de palavras reservadas da linguagem C e seus respectivos agrupamentos.

Palavras Reservadas	Palavras de agrupamento
for, while, do	*loop*
int, float, char	*declarationType*
printf, puts	*echo*
case, if, else	*conditional*

Fonte: Próprio autor.

Para a aplicação da normalização 5, primeiramente, as palavras reservadas que forem encontradas no código-fonte são codificadas por um caractere não literal que representa internamente uma palavra reservada específica. Por se tratarem de caracteres especiais, a normalização 4 não os remove. Assim, após a remoção dos caracteres literais pela normalização 4, os caracteres especiais são substituídos pelas palavras de agrupamento correspondentes.

Um exemplo é apresentado na Tabela 3.2, em que um conjunto de caracteres “@” é utilizado para representar as palavras reservadas contidas no arquivo de configuração, como visto no exemplo da Tabela 3.1. Observa-se aqui que qualquer caractere especial pode ser configurado, incluindo aqueles que não possuem representação visual. Utilizou-se o “@” no exemplo para facilitar a compreensão da técnica.

Tabela 3.2 - Substituição das palavras reservadas por máscaras

Original	Aplicação da máscara

int main () {
  int iSecret, iGuess;
  do{
  printf("Adivinhe o numero (0 a 9): ");
  scanf("%d",&iGuess);
  if(iSecret<iGuess)
  printf("O numero secreto eh menor...\n");
  else if(iSecret>iGuess)
  printf("O numero secreto eh maior...\n");
  } while (iSecret != iGuess);
}
	@@ main() {
	  @@ iSecret,
	  @@iGuess;
	  @@@ {
	  @("adivinhe o numero(0 a 9) : ");
	  @(" % d", &iGuess);
	  @@@@(iSecret<iGuess) {
	  @("o numero secreto eh menor... ");
	  } @@@@{
	  @("o numero secreto eh maior... ");
	  }
	  } @@@ (iSecret != iGuess);
	}

Fonte: Próprio autor.

Após a substituição das palavras reservadas por suas máscaras internas e aplicação das regras da normalização 4, as palavras codificadas da Tabela 3.2 são substituídas pelas suas palavras de agrupamento, como pode ser visto na Tabela 3.3.

A normalização 5 também possui um conjunto de regras de espaçamento e organização de estruturas particular, cuja necessidade foi estabelecida empiricamente a partir de observações realizadas ao longo dos experimentos.

Tabela 3.3 - Aplicação da normalização 4 e substituição das máscaras pelas palavras de agrupamento

Aplicação da máscara e regras específicas da normalização 5	Aplicação da normalização 4	Substituição por *tokens*

@@ main() {
  @@iSecret,
  @@iGuess;
  @@@ {
  @("adivinhe o numero(0 a 9) : ");
  @(" % d", &iGuess);
  @@@@(iSecret<iGuess) {
  @("o numero secreto eh menor... ");
  } @@@@{
  @("o numero secreto eh maior... ");
  }
  } @@@ (iSecret != iGuess);
}
	
	@@ (){
	@@;
	@@;
	@@@ {
	@();
	(, &);
	@@@@ {
	@(); }
	@@@@ {
	@(); }
	}@@@ (!=); }
		
		declarationType () {
		declarationType;
		declarationType;
		loop {
		echo ();
		(, &);
		conditional {
		echo (); }
		conditional {
		echo (); }
		} loop(!=); }

Fonte: Próprio autor.

Primeiramente, observou-se que a declaração de múltiplas variáveis em uma mesma linha compromete a representação e, consequentemente, a análise de similaridade.

Dessa forma, na normalização 5 cada declaração é separada em uma linha distinta, como podemos exemplificado na Tabela 3.4.

Tabela 3.4 - Aplicação da normalização 5 em declarações aglomeradas

Código original	Código normalizado
int a,b,c,d,e;	int a; int b; int c; int d; int e;

Fonte: Próprio autor.

Essa mesma situação afetava a normalização 4, que não tratava este tipo de ocorrência, diminuindo a similaridade entre um par de código-fonte. Essa regra foi, portanto, inserida igualmente na normalização 4.

A segunda observação feita na normalização 5 é que a presença ou a ausência de chaves em trechos de repetição ou de estruturas condicionais nas situações em que há apenas uma instrução implica em diferença na contabilização da similaridade.

Exemplos dessas situações podem ser observados na Tabela 3.5. Apesar dos códigos da esquerda e da direita não apresentarem diferenças em termos de compilação, quando comparados por meio do Sherlock N-overlap com normalização 4, possuem um percentual de similaridade reduzido. Para contornar essa situação, embora dispensáveis pela linguagem de programação, a normalização 5 insere sempre as chaves omitidas.

Realizando o teste com os códigos da Tabela 3.5, o Sherlock N-overlap com normalização 4 apresentou similaridade igual a 73%. Já com a normalização 5, o resultado subiu para 83%.

Tabela 3.5 - Exemplo de omissão de chaves

Código original	Código plagiado

int main()
{
int a=0, b=0, x=0, y=0, z=0;
printf("digite o numero de termos ");
scanf(" % d", &b);
printf(" ");
for(a=1; a <=b; a++){
if(a % 3 ==1)
printf(" % d, ", (a / 3) +1);
else
if(a % 3 ==2)
printf(" % d, ", (a / 3) +4);
else
printf(" % d, ", (a / 3) +3);
}
return 0;
}
	int main()
	{
	int a=0, b=0, x=0, y=0, z=0;
	printf("digite o numero de termos ");
	scanf(" % d", &b);
	printf(" ");
	for(a=1; a <=b; a++){
	if(a % 3 ==1){
	printf(" % d, ", (a / 3) +1);
	}
	else{
	if(a % 3 ==2){
	printf(" % d, ", (a / 3) +4);
	}
	else{
	printf(" % d, ", (a / 3) +3);
	}
	}
	}
	return 0;
	}

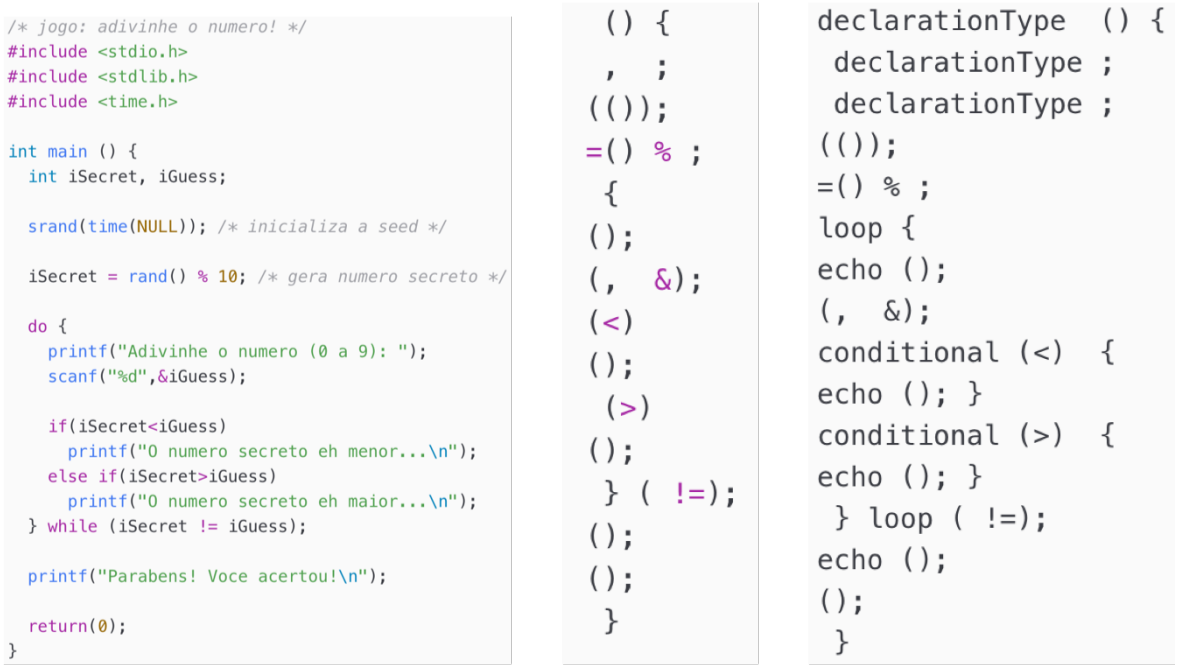
Fonte: Próprio autor.

Finalmente, a normalização 5 permite excluir da comparação blocos de código-fonte predefinidos. No contexto do ensino de programação, essa funcionalidade é usada para casos em que uma porção de código é previamente fornecida pelo professor em conjunto com o enunciado do problema a ser resolvido.

Denomina-se *template* este código parcial fornecido, e o objetivo de sua eliminação é a possibilidade de verificação de similaridade apenas das linhas de código desenvolvidas como solução ao problema proposto, evitando-se, assim, que o percentual de similaridade seja elevado devido a sua presença. Para uso desta funcionalidade, um arquivo auxiliar contendo o *template* deve ser fornecido juntamente com os códigos a serem comparados. Durante o pré-processamento, este conteúdo é excluído do código-fonte original.

Na Figura 3.1, é mostrado um código-fonte original e o resultado das normalizações 4 e 5.

Figura 3.1 – Comparação entre o código original e as normalizações.



```c
/* jogo: adivinhe o numero! */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main () {
  int iSecret, iGuess;

  srand(time(NULL)); /* inicializa a seed */

  iSecret = rand() % 10; /* gera numero secreto */

  do {
    printf("Adivinhe o numero (0 a 9): ");
    scanf("%d",&iGuess);

    if(iSecret<iGuess)
      printf("O numero secreto eh menor...\n");
    else if(iSecret>iGuess)
      printf("O numero secreto eh maior...\n");
  } while (iSecret != iGuess);

  printf("Parabens! Voce acertou!\n");

  return(0);
}
```

(a)

```
 () {
 ,  ;
(());
=() % ;
 {
();
(,  &);
(<)
();
 (>)
();
 } ( !=);
();
();
 }
```

(b)

```
declarationType  () {
 declarationType ;
 declarationType ;
(());
=() % ;
loop {
echo ();
(,  &);
conditional (<)  {
echo (); }
conditional (>)  {
echo (); }
 } loop ( !=);
echo ();
();
 }
```

(c)

Legenda: (a) Código original; (b) Resultado da normalização 4 aplicada ao código original e (c) Resultado da normalização 5 aplicada ao código original.
Fonte: Próprio autor.

Os resultados da aplicação da normalização 5 e a sua comparação aos resultados da normalização 4 são encontrados no Capítulo 4, com discussões sobre as vantagens efetivas de se ter um código mais ou menos inteligível ao ser humano para a comparação proposta pelo Sherlock N-overlap.

Na Seção 3.2, explora-se a mudança de representação do código-fonte para o domínio das imagens.

3.2 I-Sim: Representação de códigos-fonte através da imagem

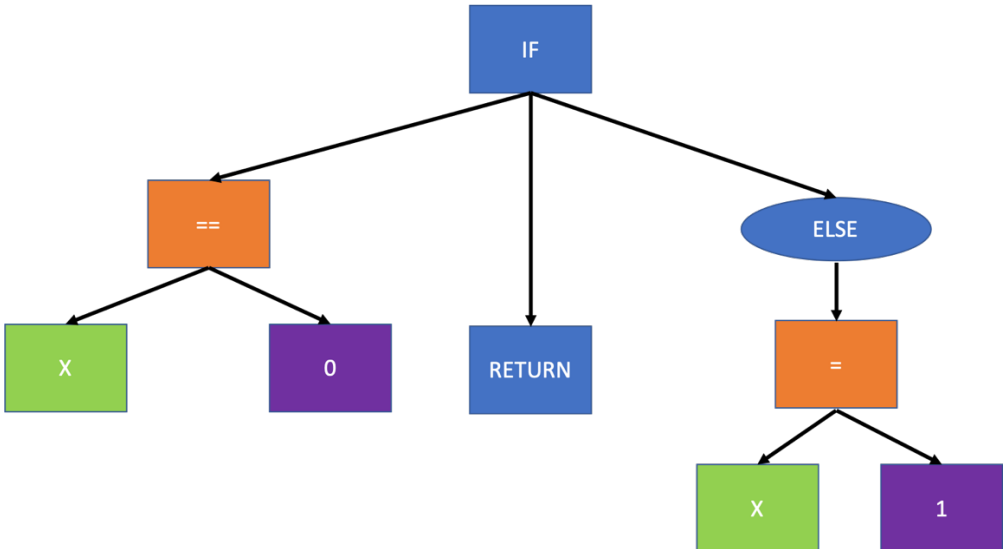
Os estudos das normalizações propostos para o Sherlock N-Overlap, em que os códigos comparados são submetidos previamente a modificações bastante invasivas, resultando em uma recodificação parcial, mostraram ser possível conservar a essência que permite a análise de similaridade.

Decidiu-se investigar, em seguida, uma nova abordagem para a análise de similaridade que ressalte aspectos do código possíveis de serem visualizados. A hipótese é ser viável representar códigos-fonte em forma de imagens e destacar determinados padrões para a avaliação da similaridade.

3.2.1 Transformando código em imagem

A primeira abordagem avaliada foi a representação de códigos-fonte pelas imagens geradas pelas árvores sintáticas abstratas, a exemplo da que é mostrada na Figura 2.3. Nessa abordagem, os atributos configuráveis dessas imagens são os rótulos, suas cores e as formas de seus delimitadores (quadrado, retângulo, círculo), bem como, a cor dos delimitadores e de suas setas.

Figura 3.2 - Árvore sintática abstrata colorida



Fonte: Próprio autor.

Dessa forma, as ASTs podem ser manipuladas para que certos atributos da linguagem sejam representados pela imagem, como o exemplo na Figura 3.2, em que as palavras reservadas são representadas por retângulos na cor azul, os operadores por retângulos de cor laranja, as variáveis por retângulos na cor verde e os valores das atribuições pela cor lilás.

Porém, a comparação entre imagens das ASTs coloridas não se mostrou viável, pois pequenas alterações no código geravam grandes alterações nas árvores, além disso, códigos grandes geram também grandes estruturas de árvore, dificultando a compreensão e a comparação.

Como alternativa, adotou-se a técnica de visualização baseada em hierarquia, as chamadas *treemaps* (SHNEIDERMAN, 1992), que se baseia no preenchimento de espaços visando:

1. Garantir a separação entre os nós;

2. Assegurar a visibilidade de cada nó;
3. Otimizar a ocupação.

Figura 3.3 - Etapas de geração da imagem

Fonte: Próprio autor.

Para a utilização das *treemaps,* primeiramente se precisa extrair as informações hierárquicas dos códigos. Para isso, o código passa por um extrator de informações que transforma o código em uma árvore hierárquica. Após esse processo, a imagem é gerada como ilustra a Figura 3.3.

A solução adotada para diferenciar os elementos do código foi associar atributos, funções e instruções de linguagem a cores específicas. Dessa forma, funções, operadores, argumentos, condições e palavras reservadas da linguagem como *if, else, printf, scanf*, entre outras, recebem cores distintas para representá-los na imagem. Cada cor é simbolizada por seus componentes RGB, conforme ilustrado na Tabela 3.6.

Logo nos primeiros testes, verificou-se ser preciso que os atributos tabulados em forma hierárquica passassem por uma espécie de normalização, visando remover nós considerados irrelevantes por trazerem informações com pouca ou nenhuma importância para a comparação entre os códigos.

Tabela 3.6 - Associação dos atributos e suas cores correspondentes

NÓ	RGB	COR
operadores	191, 191, 191	
declarações	127, 127, 127	
argumentos	100, 0, 0	
condição	127, 0, 0	
instrução *do*	255, 255, 0	
chamada de função (*call*)	127, 127, 0	

NÓ	RGB	COR
instrução *if*	0, 255, 0	
instrução *else*	0, 127, 0	
funções	0, 255, 255	
instrução *return*	0, 0, 255	
instrução *while*	0, 127, 127	
instrução *switch*	255, 0, 255	
instrução *typedef*	127, 0, 127	
instrução *for*	0, 0, 127	

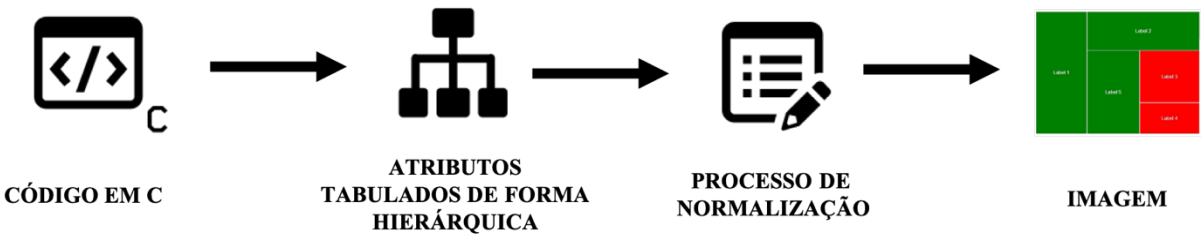
Fonte: Próprio autor.

Portanto, no processo de transformação proposto neste trabalho, os códigos passam por uma etapa adicional relativa à construção da imagem, como ilustrado na Figura 3.4.

Os nós removidos são listados a seguir:

1. Símbolos como:), (, }, {,], [, etc;
2. Rótulos de estruturas como: *case*, *default*, *break*, *continue*;
3. Declaração de variáveis e funções;
4. Operadores aritméticos;
5. Tipos de parâmetros de funções;
6. Retorno de funções;
7. Nomes de variáveis, valores de inicialização, etc;
8. Argumentos de inicialização/parada de estruturas como *for*, *do*, *while* etc.

Figura 3.4 - Processo de geração de imagem



Fonte: Próprio autor.

Após gerar e normalizar as imagens que representam os códigos-fonte a serem comparados, pode-se obter o índice de similaridade entre as duas. Para isso, foram utilizadas duas técnicas cujos desempenhos podem ser vistos nos resultados descritos no Capítulo 4, e que serão abordadas nas próximas subseções, a saber:

1. Sobreposição das imagens e identificação da superfície de recobrimento;
2. Utilização dos descritores *Color Spectrum* (SANTOS *et al*., 2012).

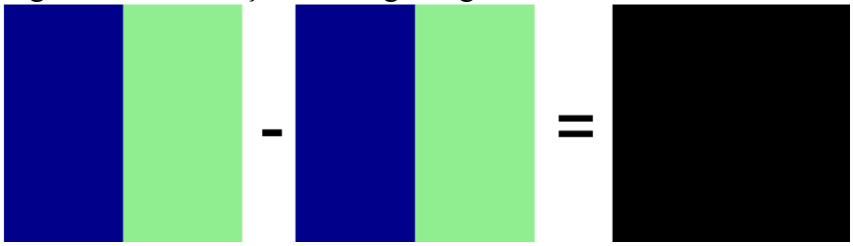
3.2.2 Cálculo do índice de similaridade por sobreposição de imagens

A primeira forma proposta para verificar a semelhança entre as imagens geradas foi baseada na superfície de recobrimento, fazendo-se a comparação de equivalência *pixel* a *pixel* de cada imagem. Para isso, é preciso que elas possuam a mesma resolução, isto é, que o número de *pixels* equivalentes seja o mesmo nas duas imagens. Adotou-se uma resolução padrão de 1024x1024 *pixels*.

Para determinar a similaridade, é realizada a subtração dos índices de cada *pixel* correspondente, gerando-se uma terceira imagem. Contabiliza-se, em seguida, o número de *pixels* iguais a zero na imagem resultante e divide-se este valor pela resolução da imagem. Assim, caso uma imagem seja idêntica à outra, após a subtração, teremos a quantidade de *pixels* com valor zero igual à resolução, indicando 100% de similaridade. A imagem resultante neste caso seria preta, como ilustrado na Figura 3.5.

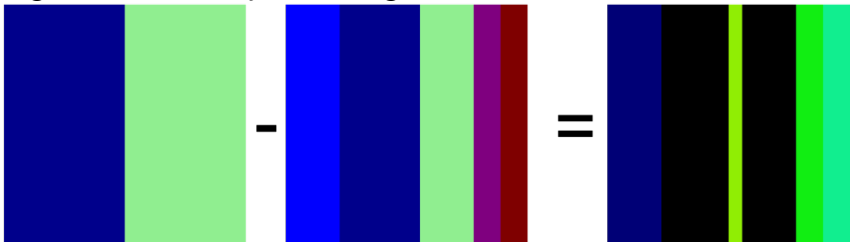
Caso as imagens sejam diferentes, o resultado da subtração resultará em uma imagem com regiões em preto, que identificam regiões sobrepostas, e regiões em outras cores, que indicam aquelas que não são sobrepostas. Estas últimas correspondem à diferença efetiva entre as imagens, que se presume ser a mesma entre os códigos que as geraram. Um exemplo é apresentado na Figura 3.6, na qual a similaridade calculada entre as imagens é de 50%.

Figura 3.5 - Subtração de imagens iguais



Fonte: Próprio autor.

Figura 3.6 - Subtração de imagens diferentes



Fonte: Próprio autor.

A ferramenta que incorpora a transformação do código em imagens, bem como, a sua comparação utilizando o método de sobreposição de imagens, foi denominada no trabalho de I-Sim1. Os resultados alcançados da ferramenta e outras discussões são apresentados no Capítulo 4.

3.2.3 Cálculo do índice de similaridade entre as imagens utilizando descritores

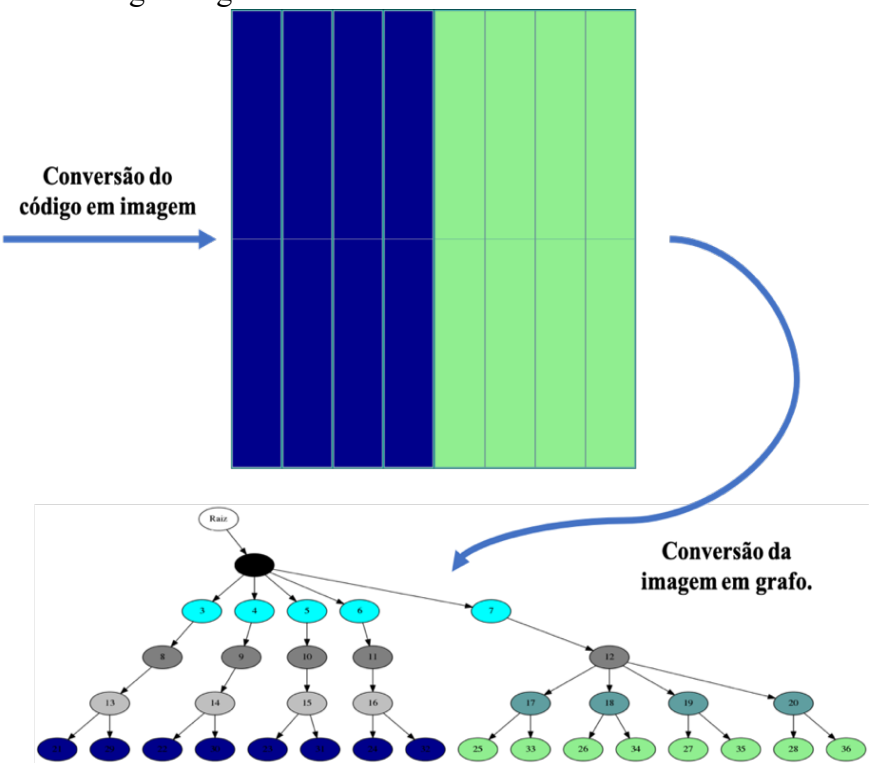
A segunda técnica proposta para a comparação é feita por meio da extração dos descritores *Color Spectrum* (SANTOS *et al*., 2012). Primeiramente, é gerado o grafo topológico da imagem, de acordo com o ilustrado na Figura 3.7. Em seguida, são extraídos os subgrafos que o compõem, sendo cada um deles convertido em uma matriz de similaridade de cor. Desta matriz, são extraídos os autovalores que, depois de ordenados, compõem o respectivo descritor, conforme explicado na Seção 2.6.4 e ilustrado na Figura 3.8.

O grafo que representa a imagem é processado para que sejam extraídos todos os subgrafos presentes em sua composição. Depois de gerados todos os subgrafos e extraídos os seus respectivos descritores, o processo de comparação entre as imagens é iniciado.

Para se obter uma baixa complexidade na comparação entre as imagens, primeiramente se comparam os espectros dos subgrafos por meio do cálculo da distância Euclidiana. Uma distância igual a zero indica que o par de subárvores comparados é igual.

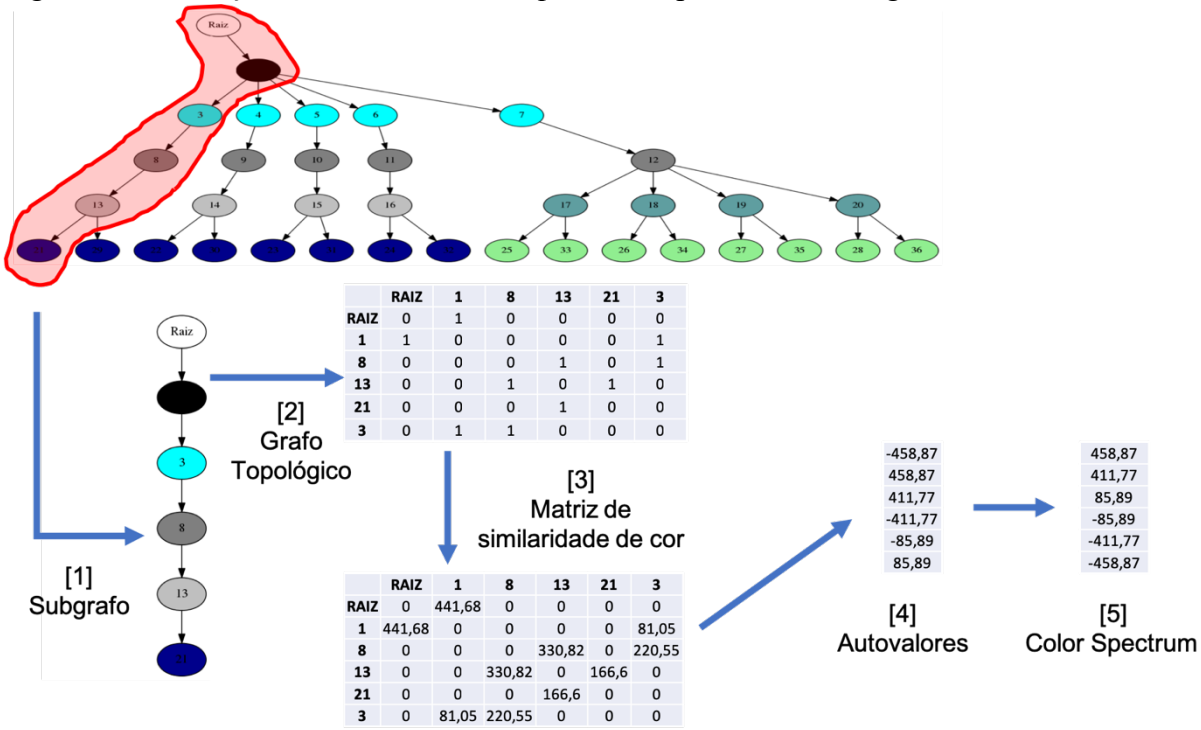
Figura 3.7 - Conversão do código em grafo

```c
#include <stdio.h>

int main(){
  int x = 0;
  int y =10;
  sum(x,y);
  sub(x,y);
  div(x,y);
  mul(x,y);
}

int sum(int a, int b){
  printf("%d", a+b);
}

int sub(int a, int b){
  printf("%d", a-b);
}

int div(int a, int b){
  printf("%d", a/b);
}

int mul(int a, int b){
  printf("%d", a*b);
}
```



Fonte: Próprio autor.

Figura 3.8 - Extração do descritor *Color Spectrum* a partir de um subgrafo.



	RAIZ	1	8	13	21	3
RAIZ	0	1	0	0	0	0
1	1	0	0	0	0	1
8	0	0	0	1	0	1
13	0	0	1	0	1	0
21	0	0	0	1	0	0
3	0	1	1	0	0	0

	RAIZ	1	8	13	21	3
RAIZ	0	441,68	0	0	0	0
1	441,68	0	0	0	0	81,05
8	0	0	0	330,82	0	220,55
13	0	0	330,82	0	166,6	0
21	0	0	0	166,6	0	0
3	0	81,05	220,55	0	0	0

Fonte: Próprio autor.

Após a realização de todas as comparações entres os subgrafos das imagens que representam os códigos, é calculada a similaridade entre elas. Utiliza-se, para isso, o coeficiente de similaridade de sobreposição (overlap), como formulado na Equação 3.1.

$$\text{Overlap}\,(G_a, G_b) = \frac{|G_a \cap G_b|}{\min(G_a, G_b)} \tag{3.1}$$

Nesta equação, o numerador representa a interseção entre os grafos do código **a** (Ga) e do código **b** (Gb), ou seja, o número de subgrafos comparados entre os dois códigos em que distância euclidiana é igual a zero. No denominador, tem-se o número de subgrafos formados pelo menor dos dois códigos.

A ferramenta que incorpora a transformação do código em imagens, bem como, a sua comparação utilizando o método envolvendo a extração dos descritores *Color Spectrum,* foi denominada no trabalho de I-Sim2. Os resultados alcançados com esta ferramenta e outras discussões são apresentados no Capítulo 4.

3.2.4 Discussões sobre a técnica de construção da imagem

Inicialmente, foram utilizadas cores distintas para as estruturas de programação mais comuns da linguagem C, conforme as definições propostas na Tabela 3.6.

Porém, algumas instruções e estruturas possuem funções semelhantes, como, por exemplo, *switch-case e if, do e while*. Lembra-se, aqui, que a substituição por instruções de propósitos semelhantes é uma das técnicas de plágio muito utilizadas e, portanto, devem ser consideradas comuns na análise de similaridade.

Assim, caso essas estruturas recebam cores distintas, a simples substituição de uma por outra geram grafos diferentes, impactando no percentual de similaridade obtido, como se pode ver na Figura 3.9.

Nesta, em (a) tem-se um trecho de código utilizando a estrutura *if* como decisão, gerando assim o grafo em (b). Na figura, em (c) tem-se o mesmo código utilizando como estrutura de decisão o *switch-case*, gerando o grafo em (d). Pode-se ver nesta imagem, portanto, que a única diferença entres os grafos é a cor que corresponde as estruturas *if* (lima) e *switch-case* (magenta). A geração de grafos distintos impacta, assim, no cálculo da similaridade.

Devido à situação exposta, algumas modificações foram propostas no primeiro modelo de cores, de forma que estruturas que executam funções semelhantes recebam cores iguais. Assim, os nós relativos ao *if* recebem a mesma cor dos nós relativos ao *switch-case*. O mesmo ocorre com as estruturas *do* e *while,* e assim por diante.

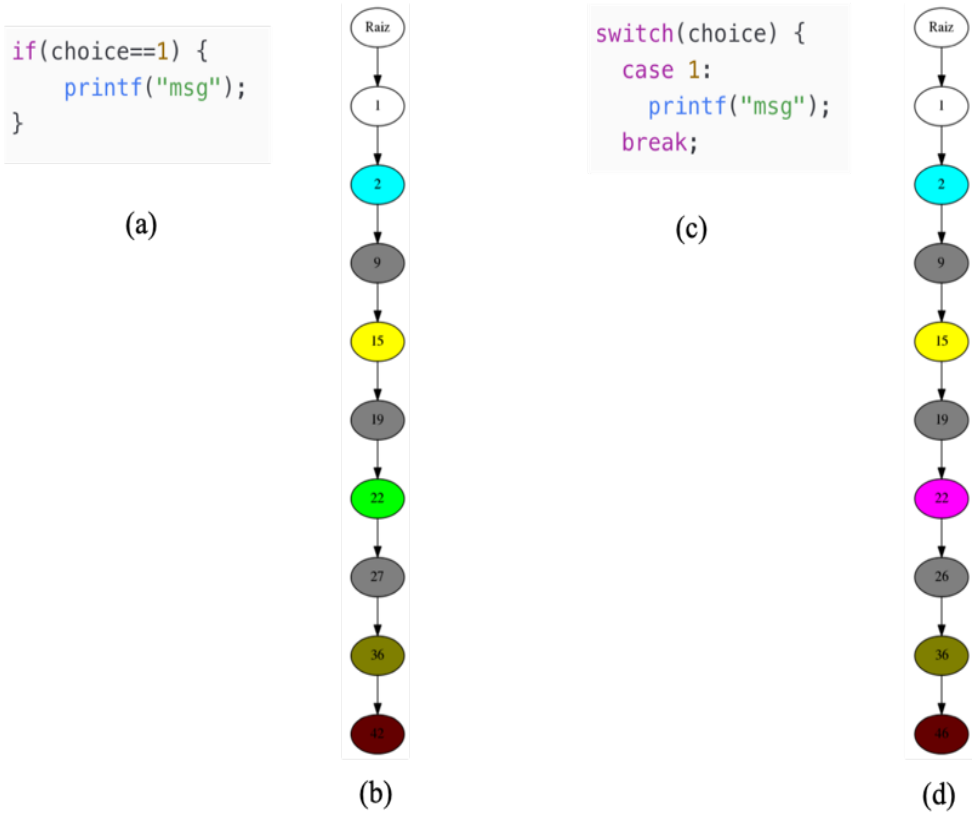
Vale lembrar também que algumas funções da linguagem também apresentam finalidades semelhantes. Por exemplo, as funções *printf* e *scanf* são utilizadas para interagir com dispositivos de entrada e saída. Funções como *pow* e *floor*, por sua vez, realizam operações matemáticas.

O nível de similaridade investigado pode ou não considerar o uso de funções alternativas da linguagem de programação. Para que seja possível capturar e analisar aspectos de semelhança de formas alternativas, propõe-se usar configurações específicas.

Assim, de maneira semelhante à normalização 5, que usa uma tabela de configuração em que estruturas são mapeadas e substituídas pelas palavras de agrupamento, propôs-se para a abordagem baseada em imagens uma tabela em que funções de linguagem que sejam consideradas semelhantes (funcionalmente) pelo avaliador podem ser mapeadas e substituídas por suas palavras de agrupamento, como exemplificado na Tabela 3.7.

Dessa forma, temos uma mesma cor representando estruturas funcionais iguais, o que mitiga o problema visualizado pela Figura 3.9.

Figura 3.9 - Códigos e seus respectivos grafos.



Fonte: Próprio autor.

Tabela 3.7 - Exemplo de palavras reservadas da linguagem C, C++ e suas respectivas palavras de agrupamento

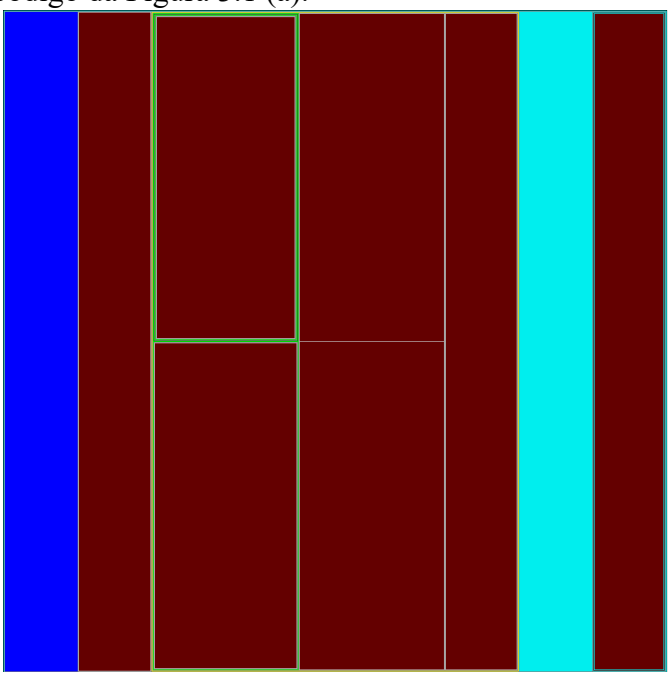
Palavras Reservadas	Palavras de agrupamento
sprintf, scanf, printf, puts, getc, getchar, putc, putchar, etc	*call_io*
labs, pow, abs, ceil, sqrt, floor, log, etc	*call_math*
stpcpy, strncpy, strcmp, strlen, strcat, strncat, strncmp, strncpy, strcasecmp, strncasecmp, strupr, strlwr, etc	*call_string*
fout, fin, freopen ,fopen, fclose, fflush, fscanf, fprintf, fgets, fgetc, fputs, fputc, fread, fwrite, fseek, rewind, ftell, feof, etc	*call_file*
isalpha, erase, sort, clear, push_back, size, end, begin, etc	*call_array*
rand, srand,goto, malloc, system, sleep, difftime, time, gettimeofday, memset, calloc, getpid, exit, etc	*call_system*
pthread_exit, pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, etc	*call_threads*

Fonte: Próprio autor.

A tabela com todos os argumentos considerados, bem como, as instruções e a associação com as cores utilizadas nos experimentos apresentados no capítulo 4 podem ser consultada no Apêndice A.

Já na Figura 3.10, pode-se ver a imagem adaptada referente a *treemap* com as regras propostas nesta tese para o código da Figura 3.1 (a).

Figura 3.10 - Imagem gerada tendo como base o código da Figura 3.1 (a).



Fonte: Próprio autor.

4 RESULTADOS

Neste capítulo, são apresentados e discutidos os resultados da aplicação do Sherlock N-overlap com a normalização 4 e 5, do I-Sim1 e do I-Sim2. Os experimentos foram realizados utilizando dois cenários.

No primeiro cenário, a análise foi realizada sobre um conjunto de códigos controlados, que foram propositadamente modificados utilizando técnicas de plágio com critérios pré-estabelecidos. Neste cenário, em que a situação de cada par de códigos é previamente conhecida, utilizou-se o método tradicional, descrito na Subseção 2.5.1.

No segundo cenário, os experimentos foram realizados com códigos produzidos por alunos de um curso de engenharia, em atividades de práticas de programação envolvendo a linguagem C. Os alunos autorizaram formalmente a utilização de seus códigos-fonte, tendo sido informados de que os mesmos seriam usados apenas em projetos de pesquisa para avaliar ferramentas de análise de similaridade.

Estes códigos foram coletados ao longo dos anos de 2012 e 2013. No caso desse experimento, por não serem conhecidas a *priori* as eventuais similaridades entre pares de códigos, foi empregado o método de conformidade, descrito na Subseção 2.5.2.

4.1 Código-fonte com similaridade conhecida a *priori*

Para este teste, foi criado um repositório de códigos, de maneira semelhante ao trabalho realizado por Dujric e Gasevic (2013), Prechelt, Malpohl e Philippsen (2002), Burrows, Tahaghoghi e Zobel (2007) e Mozgovoy, Karakovskiy e Klyuev (2007).

4.1.1 Cenário de Experimentação

Baseado no trabalho de Mozgovoy (2006), um conjunto de regras de modificações de códigos foi definido com o objetivo de nortear a construção de uma base de códigos em que a situação sobre a similaridade é conhecida a *priori*. Tal base serviu para testar se as ferramentas poderiam identificar as mudanças e assim, conseguir avaliá-las.

Para isso, os seguintes critérios foram adotados:

1. Cópia do código original sem alterações;
2. Inclusão / modificação de comentários;

3. Alteração de nomes de identificadores;
4. Mudança de posição de variáveis e funções;
5. Mudança de escopo variável;
6. Mudança no recuo;
7. Inclusão de informação sem relevância, incluindo bibliotecas, variáveis não utilizadas, comentários;
8. Mudança de expressões matemáticas;
9. Todas as alterações anteriores combinadas;
10. Código que pertence a uma tarefa diferente (sem similaridade).

Para este experimento, três programadores desenvolveram um conjunto de códigos. Cada programador recebeu duas tarefas diferentes de programação pré-estabelecidas, para as quais ele tinha que desenvolver seu código utilizando a linguagem C. As questões podem ser consultadas no Apêndice B.

Para cada questão, um código original foi gerado e alguns códigos modificados, de acordo com os critérios definidos acima. Os códigos modificados foram criados com base no original e compilam, executam e geram os mesmos resultados corretamente para as mesmas entradas, assim como os códigos originais.

Cada programador modificou seu respectivo código original sem consultar outro programador, considerando cada um dos critérios listados anteriormente. Ao todo, cada programador fez dez códigos, um original e nove códigos modificados para cada atribuição, totalizando sessenta códigos-fonte.

Além disso, três códigos que pertencem a questões diferentes foram adicionados ao conjunto (critério 10). Então, ao todo, nosso conjunto de dados tem 84 códigos (14 códigos por questão). Para garantir a autenticidade dos plágios, e certificar que seguem todos os critérios estabelecidos, os códigos foram avaliados posteriormente por dois monitores de programação.

Os códigos e as questões foram separados em dois grupos, denominados A e B. O grupo A possui as questões mais simples que resultam em códigos pequenos (entre 10 e 20 linhas). O grupo B, por sua vez, é composto de questões um pouco mais difíceis e que resultaram em códigos contendo entre 60 e 70 linhas.

Os códigos gerados foram submetidos à análise de similaridade utilizando as seguintes ferramentas: I-Sim1, I-Sim2, Sherlock N-overlap com normalizações 4 e 5, SIM, MOSS e JPlag. A porcentagem de similaridade retornada por cada uma foi agrupada em uma

única tabela, como visto na Tabela 4.1. A precisão, a revocação e a média harmônica foram calculadas a partir dos resultados de similaridade para a par contidos nessa tabela.

Nestes experimentos, uma vez que a similaridade entre códigos utilizados é conhecida a *priori*, uma alta similaridade em todos os resultados é esperada, exceto para os códigos gerados pelo critério 10, visto pertencerem a questões diferentes.

4.1.2 Usando o método tradicional para a comparação entre ferramentas

Os códigos de cada questão foram analisados por cada ferramenta, e a porcentagem de similaridade retornada foi agrupada em uma tabela para facilitar a verificação, como visto na Tabela 4.1. A precisão, a revocação e a média harmônica foram calculadas a partir desses dados.

Na Tabela 4.1, encontram-se registrados os resultados da comparação entre os códigos modificados e original do Grupo A desenvolvido pelo programador 1. São apresentados os índices de similaridade que as ferramentas analisadas encontraram (em porcentagem). Os resultados das ferramentas Sherlock N-Overlap, normalização 4 e 5, I-Sim1, I-Sim2, JPlag, SIM e MOSS foram inseridos na mesma tabela com o objetivo de facilitar a comparação.

Tabela 4.1 - Porcentagens de similaridades encontradas pelas ferramentas analisadas dos códigos do programador 1 desenvolvidos para o grupo B.

	original.c						
Algoritmo	**Sherlock N-overlap**	**Sherlock N-overlap**	**I-Sim1**	**I-Sim2**	**JPlag**	**SIM**	**MOSS**
normalização	**4**	**5**	**-**	**-**	**-**	**-**	**-**
mod0.c	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	93,00%
mod1.c	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	93,00%
mod2.c	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	58,00%
mod3.c	100,00%	100,00%	100,00%	100,00%	82,70%	89,00%	76,00%
mod4.c	33,00%	80,00%	82,26%	100,00%	0,00%	43,00%	0,00%
mod5.c	100,00%	100,00%	100,00%	100,00%	50,00%	56,00%	93,00%
mod6.c	100,00%	100,00%	100,00%	100,00%	0,00%	100,00%	0,00%
mod7.c	66,00%	100,00%	100,00%	100,00%	100,00%	51,00%	74,00%
mod8.c	33,00%	100,00%	80,98%	90,91%	0,00%	0,00%	0,00%
nao-plagio-2B.c	0,00%	33,00%	40,17%	8,33%	0,00%	0,00%	0,00%
nao-plagio-2A.c	0,00%	16,00%	73,33%	20,00%	0,00%	0,00%	0,00%
nao-plagio-3B.c	0,00%	50,00%	74,69%	8,33%	0,00%	0,00%	0,00%

original.c

Algoritmo	Sherlock N-overlap	Sherlock N-overlap	I-Sim1	I-Sim2	JPlag	SIM	MOSS
normalização	**4**	**5**	**-**	**-**	**-**	**-**	**-**
nao-plagio-3A.c	0,00%	33,00%	66,71%	8,33%	0,00%	0,00%	0,00%

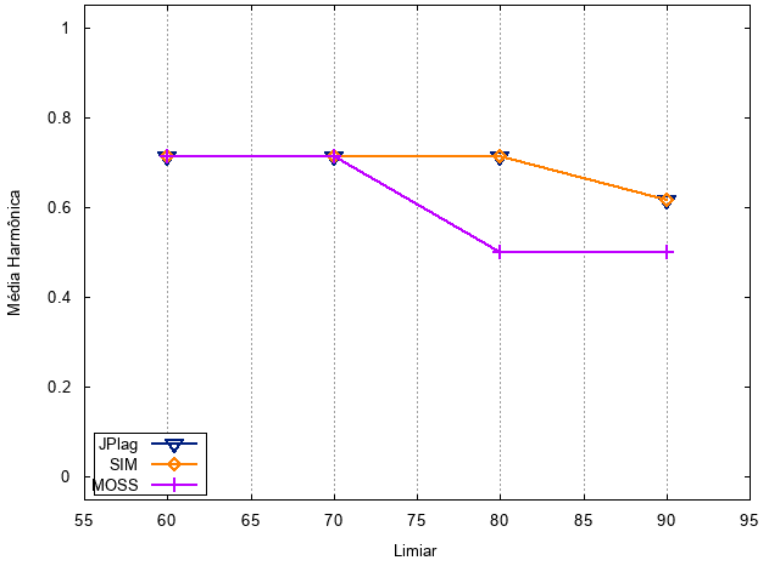
Fonte: Próprio autor.

Os índices de similaridade com o valor de 100% na Tabela 4.1 indicam que, após o pré-processamento, todos os *tokens* gerados pelas ferramentas Sherlock N-overlap, SIM, MOSS e JPlag para o código modificado foram encontrados dentre os gerados para o código original. Destaque-se que, no caso da ferramenta I-Sim1, ocorreu uma sobreposição completa das imagens geradas; já no caso da ferramenta I-Sim2, todos os subgrafos de uma imagem foram encontrados na outra imagem comparada.

Os índices de similaridade são utilizados para calcular as métricas de precisão (P), de revocação (R) e a média harmônica (F) para cada ferramenta, considerando um limiar que estabelece uma porcentagem mínima para descarte por irrelevância. Por exemplo, para um limiar de 90%, os resultados abaixo desse valor são considerados irrelevantes e, portanto, descartados no cálculo das métricas.

Os gráficos das Figura 4.1, Figura *4.2* e Figura *4.3* mostram os resultados obtidos pelo cálculo das médias harmônicas (F) para cada ferramenta. Os parâmetros de configuração utilizados pelo Sherlock N-overlap foram: t = 10, z = 2, n = 3, que foram os mesmos parâmetros utilizados em Maciel (2014) e considerados as melhores configurações. Esses gráficos podem ser usados para avaliar, dentro de cada limiar, o desempenho de cada ferramenta.

Figura 4.1 - Média harmônica calculada por limiar para os algoritmos SIM, MOSS e JPlag de códigos do Grupo A desenvolvidos pelo programador 1.



Fonte: Próprio autor.

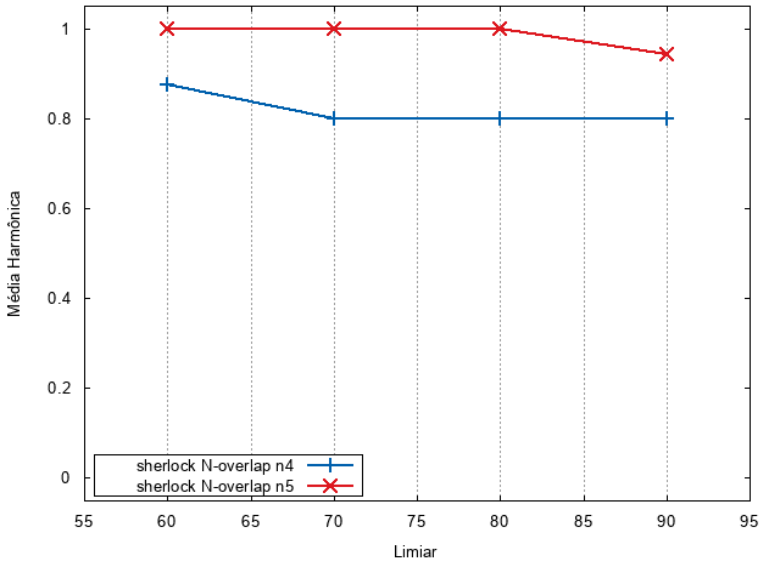
No gráfico da Figura 4.1, é possível verificar que a ferramenta MOSS para o limiar menor que 70% apresenta as mesmas taxas que o JPlag e SIM. Acima desse limiar, vê-se que o MOSS apresenta uma queda, ficando abaixo das outras ferramentas.

Isso se dá pelo fato de o MOSS ter apresentado percentuais mais baixos que outras ferramentas para alguns dos códigos modificados a partir do original como, por exemplo, o código mod4.c, em que esta ferramenta não conseguiu encontrar similaridade.

No gráfico da Figura 4.2, que compara os resultados da ferramenta Sherlock N-Overlap com as normalizações 4 e 5, pode-se verificar que a normalização 5 apresenta resultados superiores aos encontrados quando o algoritmo é submetido à normalização 4.

Isso se deve ao fato de o algoritmo submetido à normalização 4 detectar uma similaridade baixa para os códigos mod4.c, mod7.c e mod8.c. Isso mostra que, no caso específico, a manutenção de palavras reservadas faz com que ocorra a melhora na detecção de similaridade.

Figura 4.2 - Média harmônica calculada por limiar para os algoritmos Sherlock N-Overlap com normalização 4 e 5 de códigos do Grupo B desenvolvidos pelo programador 1.

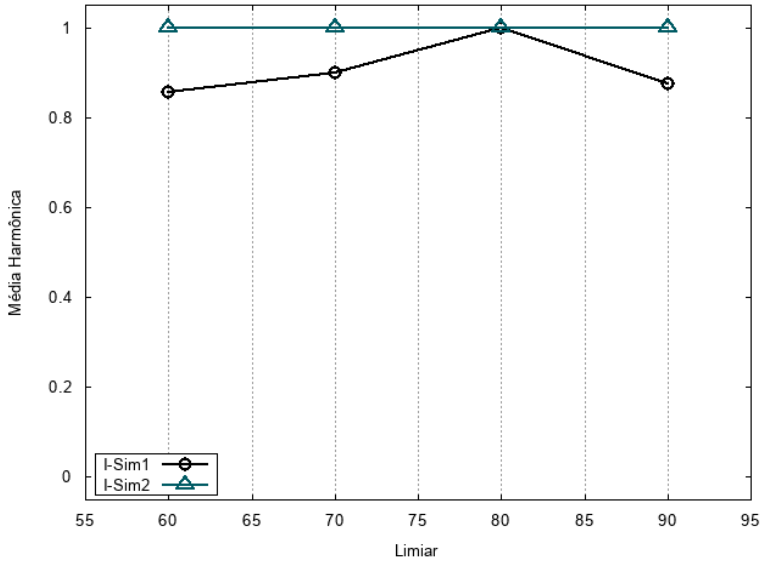


Fonte: Próprio autor.

Já no gráfico da Figura 4.3, que compara os algoritmos I-Sim1 e I-Sim2, pode-se perceber que o algoritmo I-Sim2 apresenta resultados superiores ao I-Sim1. Isso ocorre porque o I-Sim1 apresenta índices de similaridades altos para alguns códigos que pertencem a outras questões, como o código nao-plagio-3B.c.

Analisando o funcionamento da ferramenta I-Sim1, percebe-se que muitas vezes ela apresenta índices de similaridade muito altos para códigos pertencentes a outros exercícios. Como a ferramenta se utiliza da sobreposição de *pixels* para determinar a similaridade, ocorrem casos em que existe a sobreposição da mesma estrutura, em posições hierárquicas diferentes, sendo contabilizada como a mesma região de similaridade. Essa situação pode ser observada nos exemplos mostrados nas Figura 4.4, Figura *4.5* e Figura *4.6*.

Figura 4.3 - Média harmônica calculada por limiar para os algoritmos I-Sim1 e I-Sim2 de códigos do Grupo B desenvolvidos pelo programador 1

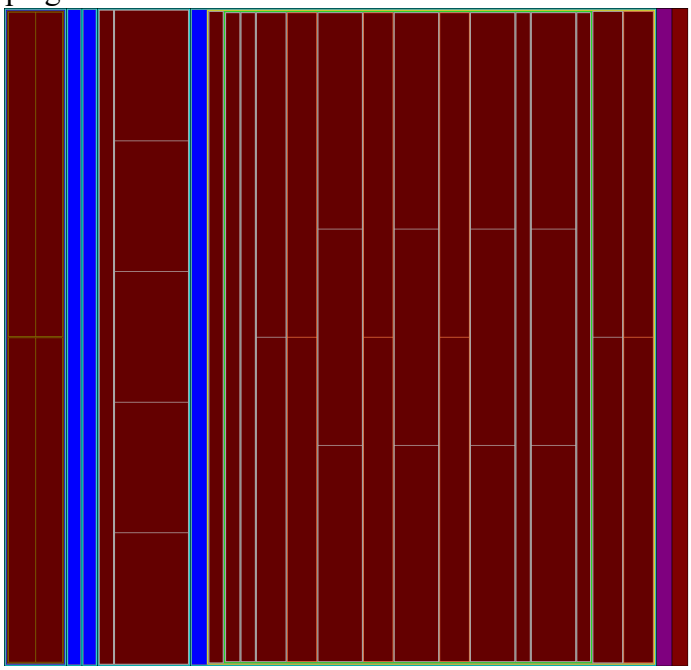


Fonte: Próprio autor.

Na Figura 4.4, é mostrada a imagem que representa o código do grupo B desenvolvido pelo programador 2, e na Figura 4.5 a imagem relativa ao código original do grupo A desenvolvido pelo programador 1. Pode-se perceber a diferença de complexidade entre os códigos pela análise visual das imagens, possuindo a primeira um número bem mais elevado de estruturas do que a segunda.

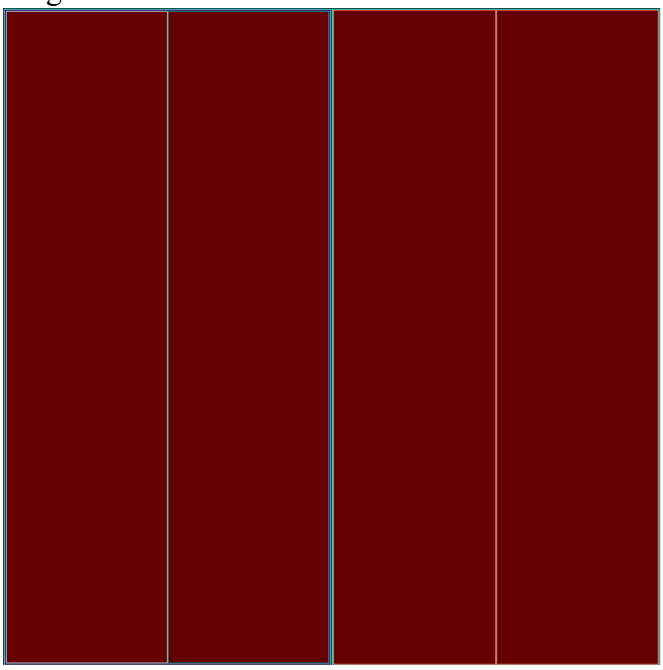
Na Figura 4.6, por sua vez, é mostrada a imagem gerada pela comparação *pixel* a *pixel* resultante da comparação entre as imagens da Figura 4.4 e da Figura 4.5. Assim, percebe-se que, mesmo se tratando de imagens completamente diferentes, como a cor de preenchimento de maior superfície é o vermelho (que representa a atributos passados para funções), a sobreposição resultou em um alto índice de similaridade.

Figura 4.4 - Imagem representativa do código não-plágio-2A.c.



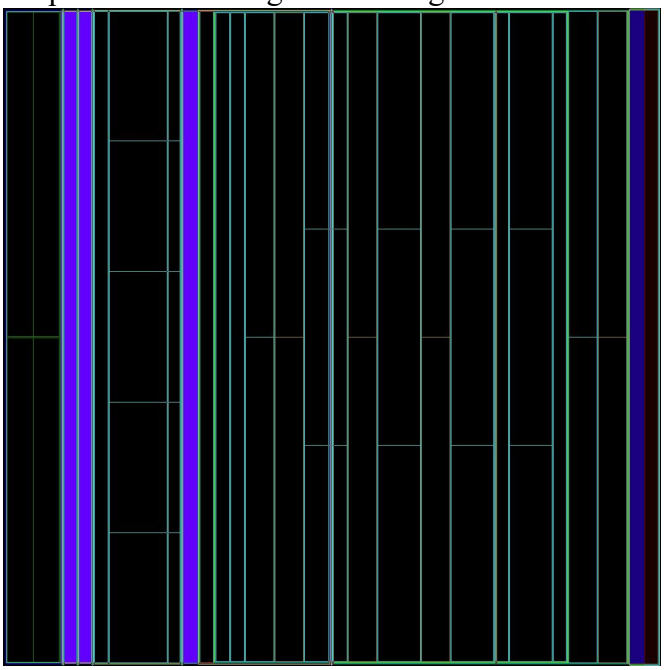
Fonte: Próprio autor.

Figura 4.5 - Imagem representativa do código original.c.



Fonte: Próprio autor.

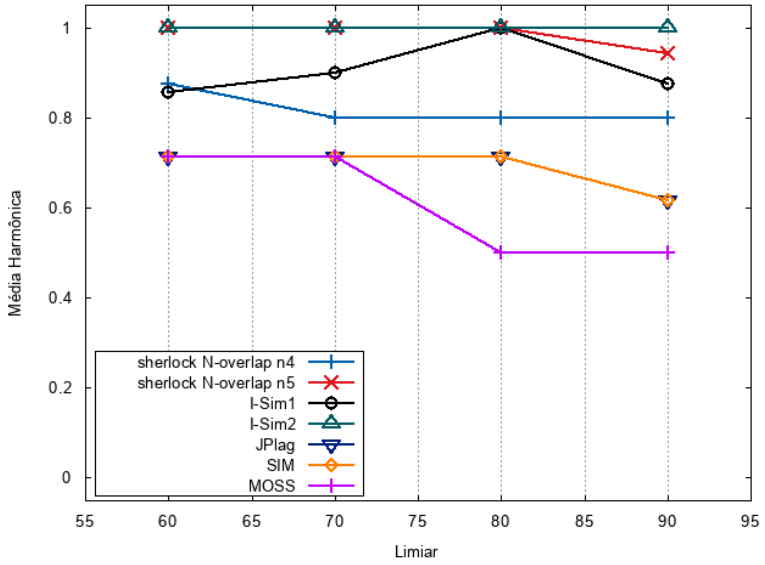
Figura 4.6 - Imagem resultante da sobreposição de pixels entre as Figura 4.2 e Figura 4.3.



Fonte: Próprio autor.

No gráfico da Figura 4.7, em que o desempenho de todas as ferramentas é comparado, percebe-se que, no caso específico, as ferramentas I-Sim2 e Sherlock N-overlap sob a normalização 5 são as que se destacam.

Figura 4.7 - Média harmônica calculada por limiar para todos os algoritmos envolvidos de códigos do Grupo A desenvolvidos pelo programador 1.



Fonte: Próprio autor.

Para comparar todos estes resultados de forma objetiva, usando gráficos sem sobrecarga de informação, uma métrica de qualidade foi utilizada para calcular a área sob a curva de interpolação entre os valores das médias harmônicas, assumindo que, na região inter-limiares calculada, esta média varia linearmente.

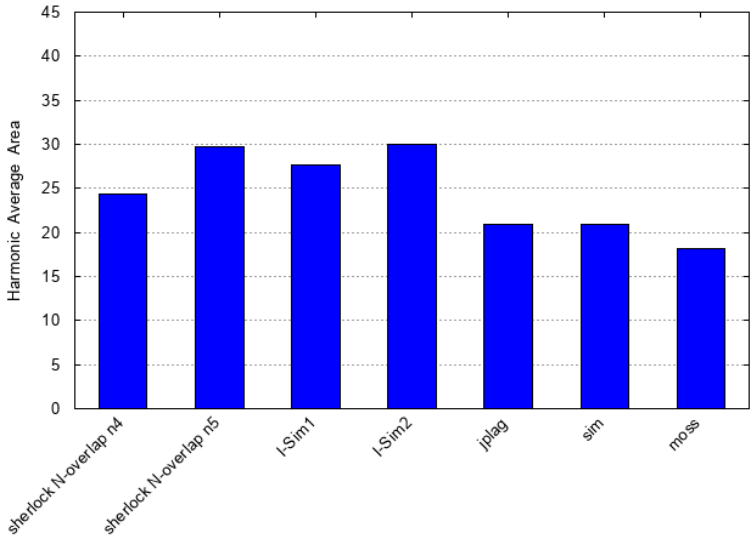
Assim, com dois pares ordenados subsequentes – P_A (x_i, y_i) e P_B (x_{i+1}, y_{i+1}) – a área é calculada pela soma do produto entre os módulos das diferenças entre os respectivos valores de x e y, dividido por 2, conforme a Equação 4.1.

$$Area = \sum_{i=1}^{n} \frac{|x_i - x_{i+1}| \times |y_i - y_{i+1}|}{2} \quad (4.1)$$

O gráfico da Figura 4.8 compara as áreas dos gráficos da Figura 4.7 usando um limite maior que 60%. Esse limite é aplicado porque, para códigos desenvolvidos em classes de laboratório de noventa minutos, índices de similaridade inferiores podem ser devidos à própria natureza da tarefa proposta e / ou ao uso de funções comuns de linguagem e bibliotecas.

Observa-se pela Figura 4.7 que o Sherlock N-overlap com normalizações 4 e 5, bem como, as ferramentas I-Sim1 e I-Sim2, apresentam os melhores resultados, sendo superiores aos algoritmos SIM, MOSS e JPlag. A ferramenta I-Sim2 se destaca, e o Sherlock N-overlap com a normalização 5 ocupa a 2ª colocação. Também é possível observar que o MOSS tem o pior desempenho, como já comentado na análise da Figura 4.1.

Figura 4.8 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 1 (limiar> 60%).



Fonte: Próprio autor.

Os passos abaixo resumem os procedimentos para obter os resultados e gráficos já apresentados:

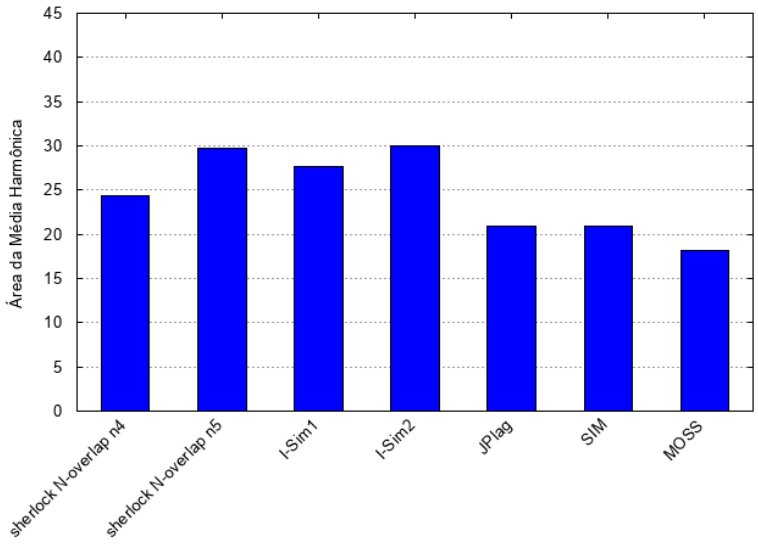
1) Execução dos algoritmos e obtenção das tabelas com os índices de similaridade calculados para cada algoritmo (de acordo com o exemplo da Tabela 4.1).
2) Contagem dos falsos positivos e falsos negativos.
3) Cálculo da precisão, revocação e média harmônica.
4) Geração do gráfico da média harmônica (de acordo com o exemplo da Figura 4.1).
5) Cálculo das áreas sob o gráfico calculado no passo 3 para um limite específico (como mostrado na Figura 4.8).

4.1.3 Discussão dos Resultados

Os resultados dos experimentos realizados com o primeiro conjunto de códigos, em que se conhece a *priori* serem modificações ou não de códigos originais, mostra uma pequena vantagem do algoritmo Sherlock N-overlap sob a normalização 5 em relação à normalização 4, como ilustrado nos gráficos das Figuras 4.9, 4.10, 4.11 e 4.12. Isso sugere que a manutenção de palavras reservadas ou de agrupamento pode melhorar não só a

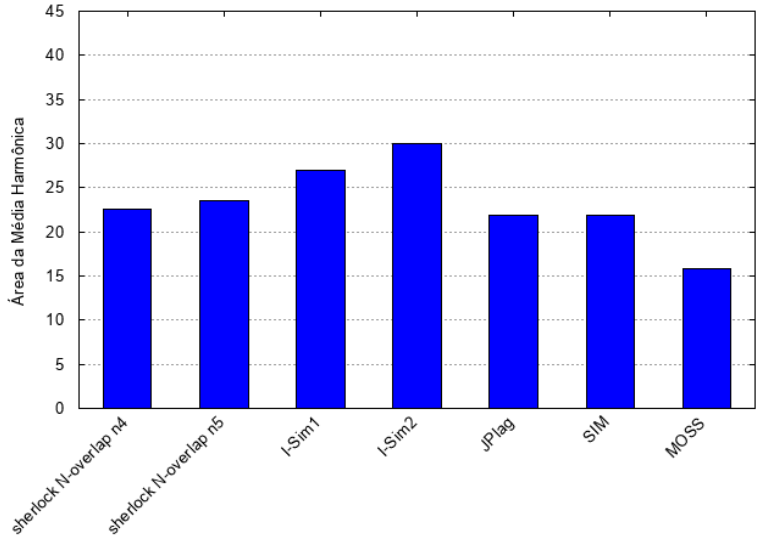
inteligibilidade para o ser humano no código pré-processado, mas também a busca de similaridade por algoritmos.

Figura 4.9 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 1 (limiar> 60%).



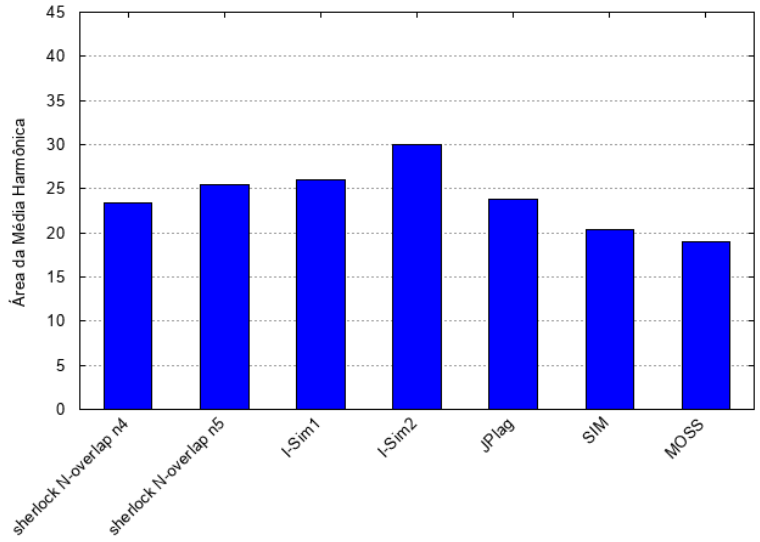
Fonte: Próprio autor.

Figura 4.10 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 2 (limiar> 60%).



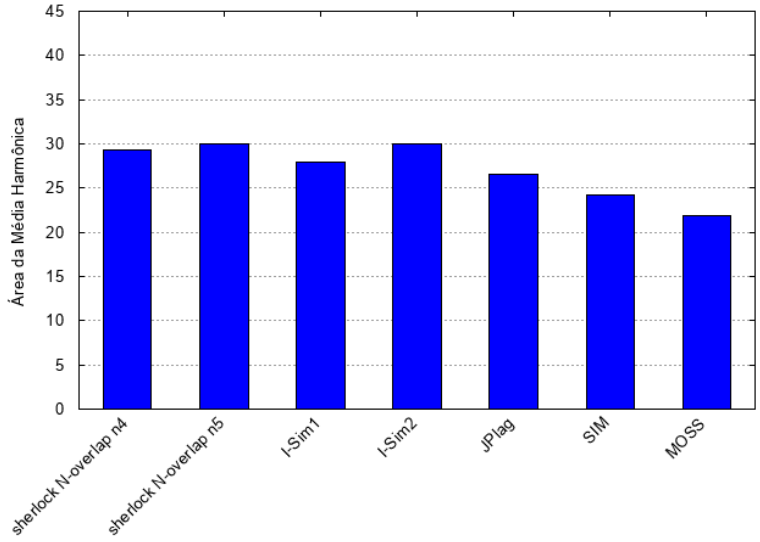
Fonte: Próprio autor.

Figura 4.11 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 2 (limiar> 60%)



Fonte: Próprio autor.

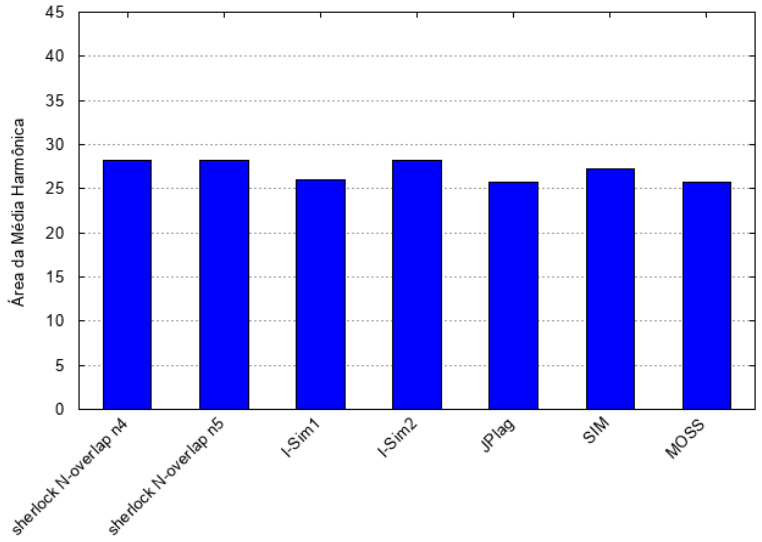
Figura 4.12 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo A desenvolvidos pelo programador 3 (limiar> 60%).



Fonte: Próprio autor.

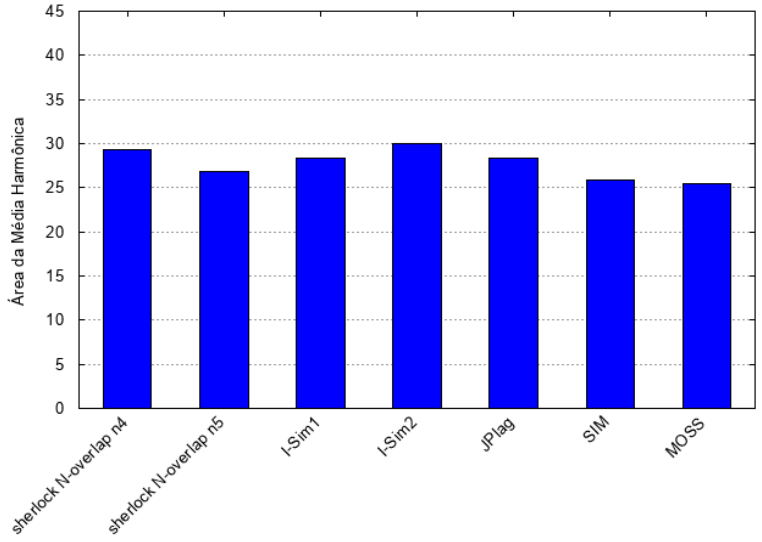
Entretanto, esse resultado não se reproduz em todas as situações, como se pode ver nos gráficos das Figuras 4.13 e 4.14. Na primeira, a normalização 5 ficou abaixo da normalização 4 por resultar em um percentual inferior para o código mod4.c. Já na segunda, a normalização 4 possui resultados superiores em 3 casos específicos (mod4.c, mod7.c e mod8.c), o que faz com que seu desempenho total seja superior ao da normalização 5.

Figura 4.13 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 1 (limiar> 60%).



Fonte: Próprio autor.

Figura 4.14 – Área da média harmônica calculada por ferramenta a partir dos códigos do Grupo B desenvolvidos pelo programador 3 (limiar> 60%).



Fonte: Próprio autor.

Além disso, os testes que são apresentados na Seção 4.2 mostram que, na maioria dos casos, a normalização 4 gera melhores resultados do que a normalização 5, o que, diferentemente dos testes realizados no primeiro cenário, mostra que a inteligibilidade do código pelo ser humano para a sua comparação uns com os outros não traz, em todos os casos, benefícios evidentes para as ferramentas.

Comparativamente aos resultados das ferramentas JPlag, SIM e MOSS, os resultados obtidos por Sherlock N-overlap sob a normalização 5 foram os superiores em cinco das seis tarefas implementadas pelos programadores, sendo superada em apenas uma tarefa pelo JPlag.

O I-Sim2, entretanto, se destaca dentre as demais ferramentas analisadas, mostrando que a abstração e comparação baseada em imagens, e a construção de grafos topológicos, são capazes de superar a verificação de similaridade fundamentada em texto. A ferramenta I-Sim1, entretanto, devido à fragilidade da métrica baseada em sobreposição de cores, incorre em muitos casos de falsos positivos, como o discutido na Subseção 4.1.2.

Apesar de ser possível fazer ajustes no Sim-1, usando, por exemplo, cores dependentes dos níveis de intrusão, a técnica foi descontinuada neste trabalho por ser uma abordagem de custo computacional elevado. Entretanto, outras perspectivas de uso podem evoluir dessa ideia inicial, incluindo a extração de características independentes de cor das regiões formadas por uma *treemap*.

De maneira a estender a análise do desempenho das ferramentas e técnicas desenvolvidas comparativamente ao grupo considerado como referência para detecção de plágio entre códigos-fonte, foram efetuadas experimentações em um conjunto de códigos significativamente maior. As experiências e os resultados são discutidos na Seção 4.2.

4.2 Código-fonte com similaridade não conhecida a *priori*

A viabilidade de aplicação do método da precisão e revocação tradicionais para um grande volume de códigos em que não se conheça a *priori* a similaridade é limitada, pois exigiria um grande esforço de classificação manual antecipada, além de ser um trabalho que envolve grande subjetividade, e que pode incorrer em possíveis inconsistências.

Essa limitação levou à utilização de um método diferenciado, capaz de comparar a qualidade de uma ferramenta de maneira relativa, considerando o resultado apresentado por outras ferramentas que vêm sendo e são avaliadas como robustas em muitos trabalhos acadêmicos (GREEN *et al*., 2012; HAGE; RADEMAKER; VUGT, 2010; RAGKHITWETSAGUL; KRINKE; CLARK, 2018; WEBER-WULFF; KÖHLER; CHRISTOPHER, 2012). Empregou-se, assim, o método de conformidade (FRANCA *et al*., 2018), apresentado na Subseção 2.5.2.

4.2.1 Cenário de Experimentação

Esta forma de experimentação, em que não se conhece antecipadamente a classificação quanto à similaridade entre os pares, foi realizada com códigos gerados ao longo dos anos de 2012 e 2013 em uma turma anual de programação para um curso de Engenharia, ministrada para 97 alunos. As turmas foram divididas em 4 grupos devido à restrição do número de máquinas nos laboratórios disponíveis. A cada semana, eram propostos 2 problemas para cada grupo de laboratório.

Nos experimentos realizados, foram considerados 22 problemas de cada turma de laboratório por cada ano. Para esse cenário específico, contabilizou-se um total de 176 problemas ao longo dos dois anos. Para cada problema, todos os códigos, uma vez submetidos pelos alunos, foram comparados entre si. Ao todo, foram considerados 12767 pares de códigos, conforme a Tabela 4.2 e a Tabela 4.3. Esse número é obtido pelo somatório da combinação 2 a 2 do total de submissões para cada problema.

Tabela 4.2 - Relação quantitativa dos códigos analisados no ano de 2012.

	Total de Exercícios Submetidos	Total de Pares Comparados
	2012	
Turma A	332	2405
Turma B	250	1325
Turma C	200	877
Turma D	299	1917
SubTotal	1081	6524

Fonte: Próprio autor.

Tabela 4.3 - Relação quantitativa dos códigos analisados no ano de 2013.

	Total de Exercícios Submetidos	Total de Pares Comparados
	2013	
Turma A	255	1368
Turma B	281	1712
Turma C	260	1406
Turma D	283	1757
SubTotal	1079	6243

Fonte: Próprio autor.

Os códigos gerados foram submetidos à análise de similaridade utilizando as ferramentas I-Sim1, I-Sim2, Sherlock N-overlap com normalizações 4 e 5, SIM, MOSS e JPlag. A porcentagem de similaridade retornada por cada uma foi agrupada em uma única tabela, como visto na Tabela 4.1. A precisão, a revocação e a média harmônica foram calculadas a partir dessa tabela.

4.2.2 Utilizando o método de conformidade para a comparação entre ferramentas

O processo desenvolvido é semelhante aos passos do método empregado no experimento descrito na Subseção 4.1.2. No entanto, algumas alterações são necessárias, uma vez que os cálculos do método de conformidade são relativos.

As etapas do processo são:

1) Execução dos algoritmos e obtenção das tabelas com as similaridades calculadas por cada algoritmo (conforme o exemplo da Tabela 4.1).
2) Contagem das ocorrências de similaridade, ocorrências isoladas e ausências isoladas.
3) Cálculo da precisão e revocação de conformidade e da média harmônica (conforme o exemplo da Tabela 4.4).
4) Geração dos gráficos da média harmônica (conforme os exemplos das Figura 4.15, Figura 4.16 e Figura 4.17).

No caso do cenário proposto, em que temos uma média de vinte atividades por turma, decidiu-se somar as médias harmônicas, obtidas para cada atividade por turma de

acordo com um determinado limiar, como forma de obter o resultado acumulado com o desempenho geral.

No método de precisão e revocação usado no primeiro cenário, 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, uma vez que é feito um somatório das atividades por turma.

A Tabela 4.4 exemplifica como o valor da média harmônica permite relacionar o desempenho entre as ferramentas estudadas com o uso do método de conformidade.

No exemplo, para um limiar de 60%, o Sherlock N-Overlap com normalização 5 registrou a maior quantidade de ocorrências, não tendo apresentado ausências ou ocorrências isoladas. Observando-se o cálculo da média harmônica, tem-se que o Sherlock N-Overlap com normalização 5 obteve o melhor resultado, com o valor de média harmônica máximo igual a 1, seguido pelo SIM, JPlag e MOSS.

Tabela 4.4 - Relatório completo de quantificação de ocorrências.

		Sherlock N-Overlap N5	SIM	MOSS	JPlag
Limiar: 60%	Total de Ocorrências	7	6	4	6
	Ocorrência Isolada	0	0	0	1
	Ausência Isolada	0	0	1	0
	Total Indicativo de Ocorrências	7	6	4	5
	Precisão	1	0,8571	0,5714	0,7143
	Revocação	1	1	0,8	1
	Média Harmônica	1,0000	0,9231	0,6667	0,8333

Fonte: Próprio autor.

No exemplo, mesmo contabilizando 6 ocorrências como o JPlag, o SIM se destacou por não registrar nenhuma ocorrência isolada, obtendo média igual a 0,92.

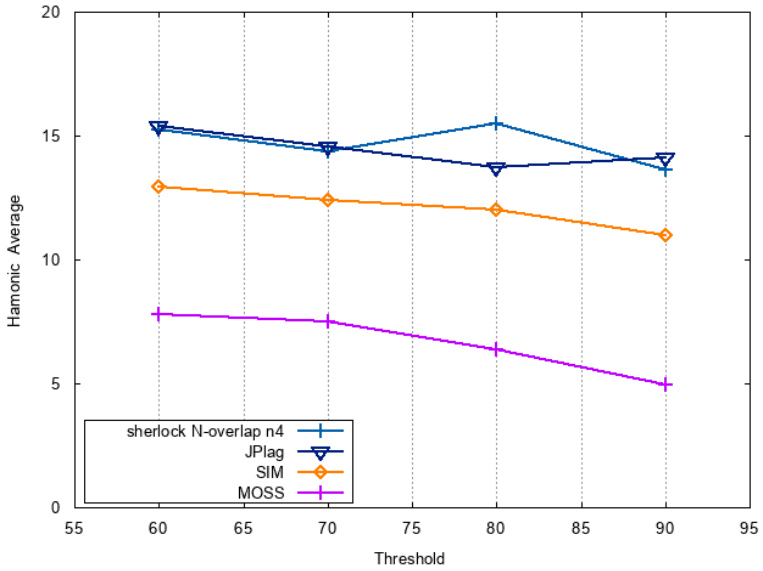
O JPlag registrou uma ocorrência isolada e obteve média igual a 0,83, ficando atrás do SIM e à frente do MOSS, que obteve média igual a 0,67, uma vez que contabilizou menos ocorrências que as demais ferramentas e apresentou uma ausência isolada. Dessa forma, pode-se ver como a média harmônica permite relacionar o desempenho entre as ferramentas analisadas.

4.2.3 Discussão dos Resultados

Para cada problema, executou-se o JPlag, MOSS e SIM, comparando-osprimeiramente com o Sherlock N-Overlap com normalização 4, seguindo-se da Normalização 5 e da ferramenta I-Sim2.

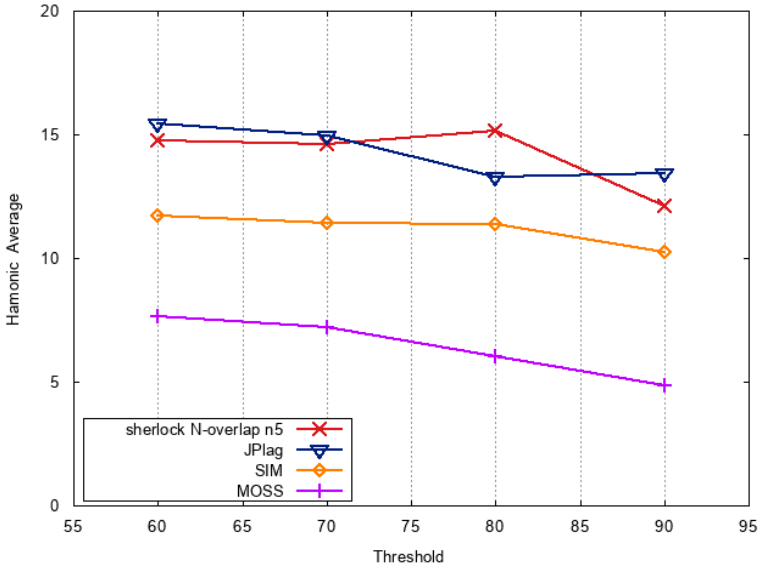
Com os resultados de cada verificação de similaridade, foi calculada a média harmônica para quatro faixas de limiares: 60%, 70%, 80% e 90%. Para cada limiar e para cada ferramenta, para efeito de comparação, os valores das médias harmônicas foram somados e apresentados conforme ilustrado nos gráficos da Figura 4.15, Figura 4.16 e Figura 4.17.

Figura 4.15 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – Sherlock N-overlap - normalização 4 – sob análise.



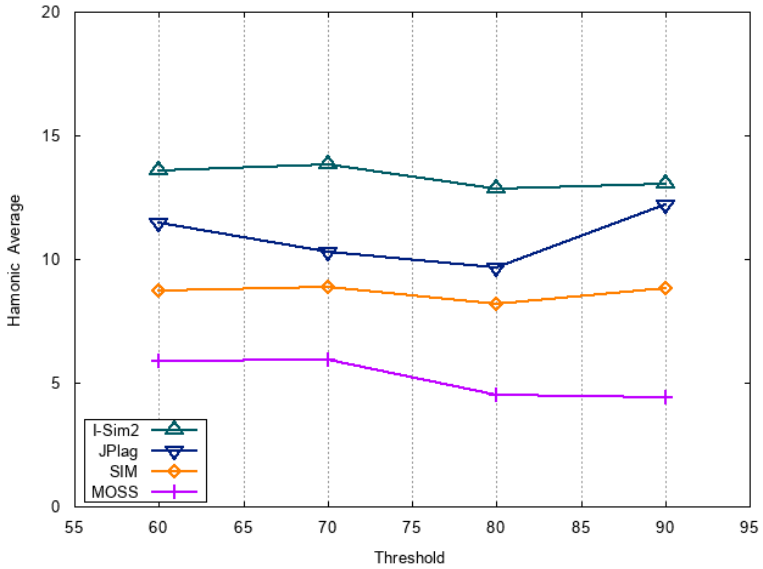
Fonte: Próprio autor.

Figura 4.16 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – Sherlock N-overlap - normalização 5 – sob análise.



Fonte: Próprio autor.

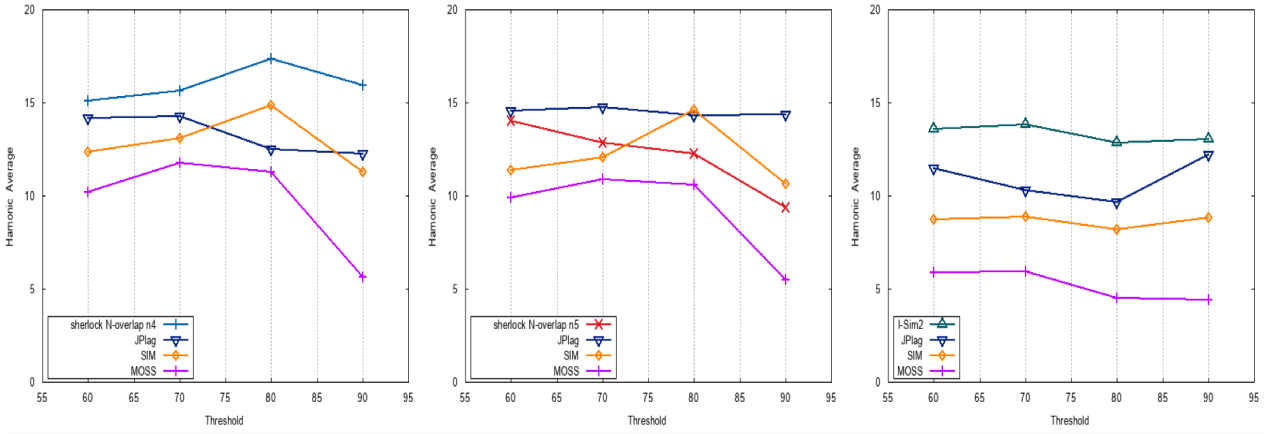
Figura 4.17 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe A (2012) por limiar de similaridade – I-Sim2 – sob análise.



Fonte: Próprio autor.

É possível observar, por meio dos gráficos obtidos pelo somatório das médias harmônicas, que a diferença entre a normalização 4 e 5 não é tão significativa. Diferentemente do observado na Seção 4.2.2, na maioria das situações deste cenário, a normalização 4 se sobrepõe à normalização 5, sendo a diferença de desempenho muito pequena. Isso é confirmado nos gráficos apresentados nas Figuras 4.18, 4.19, 4.20, 4.21, 4.22, 4.23 e 4.24.

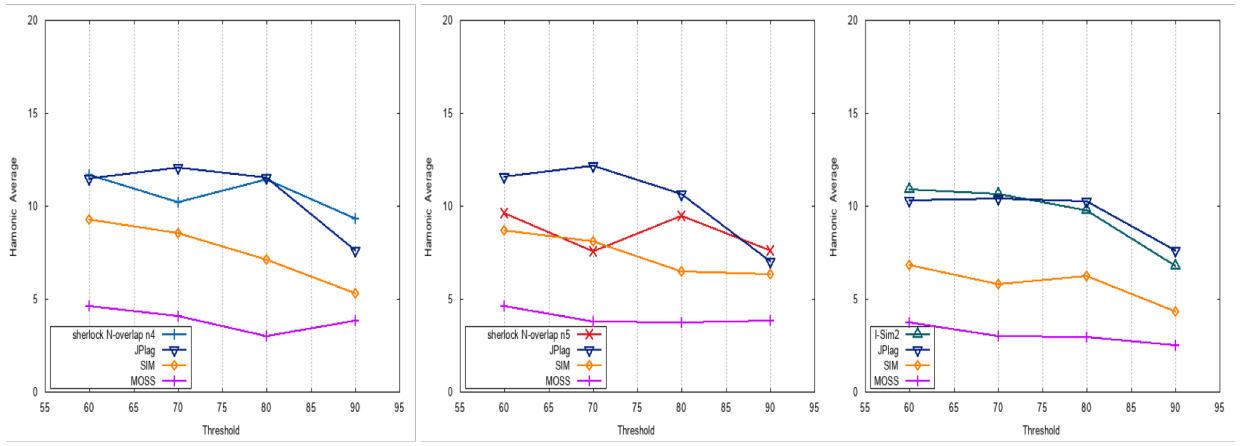
Figura 4.18 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe B (2012) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

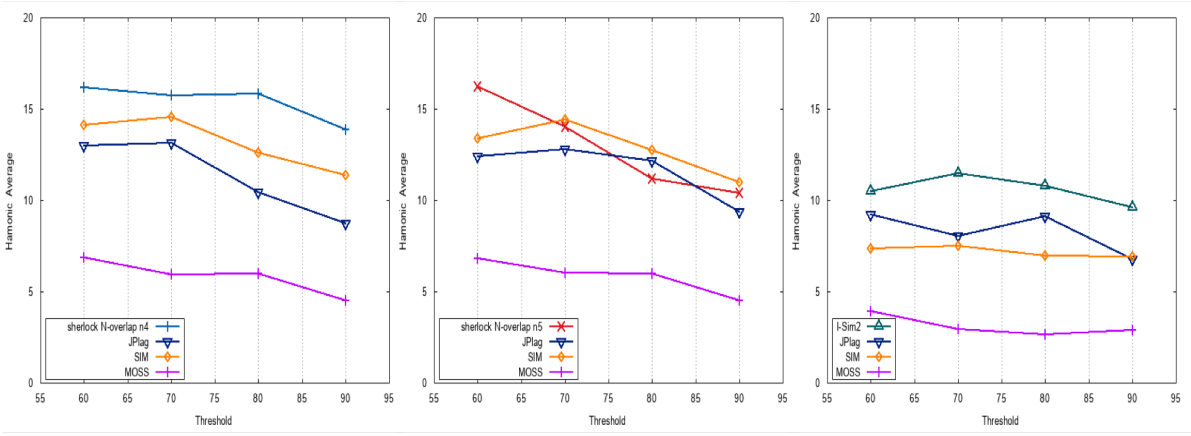
Figura 4.19 - Gráfico da soma das médias harmônicas dos códigos desenvolvidos na classe C (2012) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

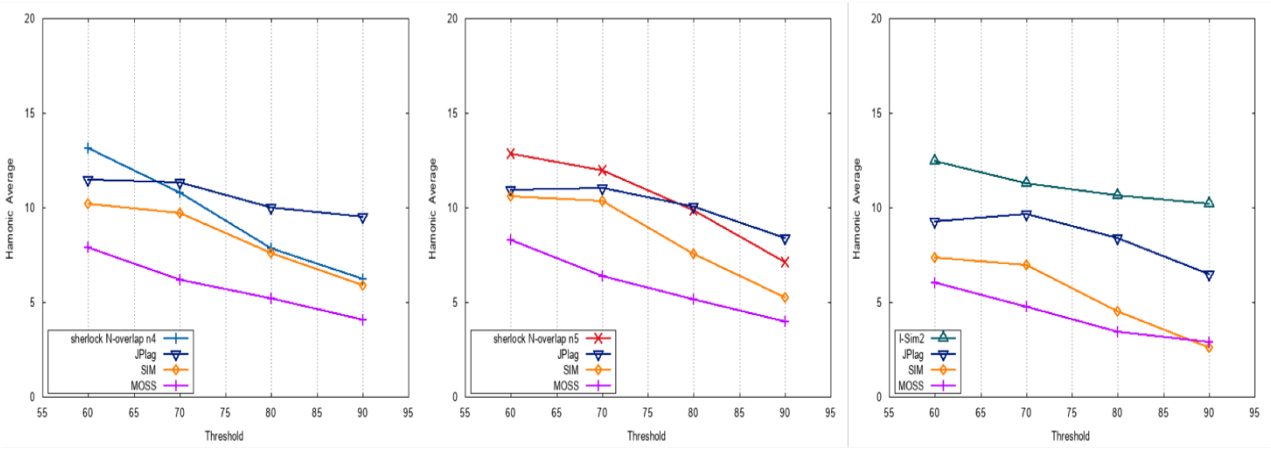
Figura 4.20 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe D (2012) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

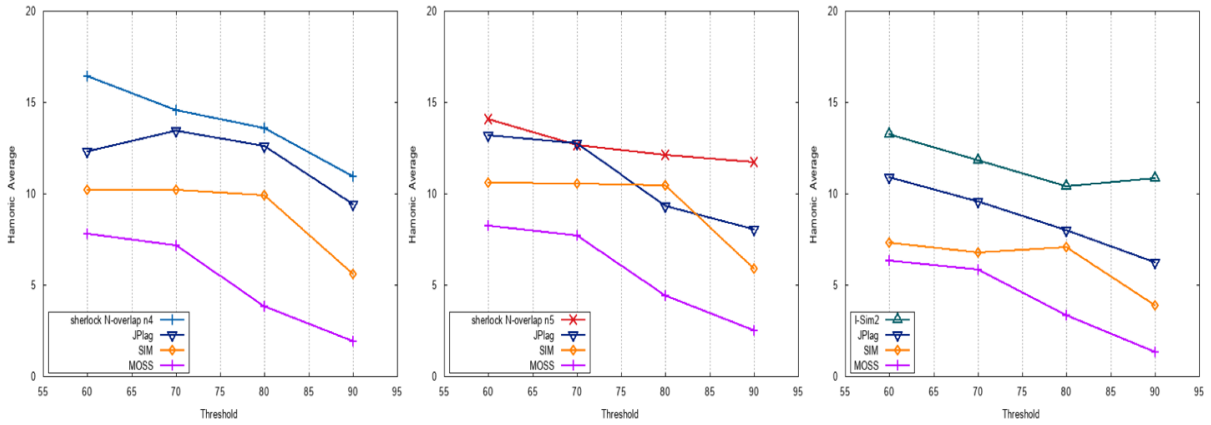
Figura 4.21 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe A (2013) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

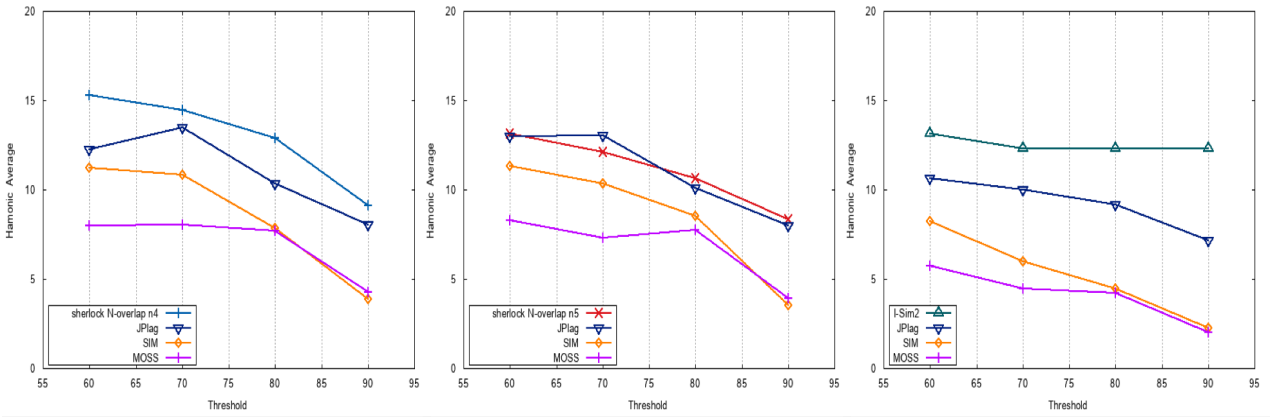
Figura 4.22 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe B (2013) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

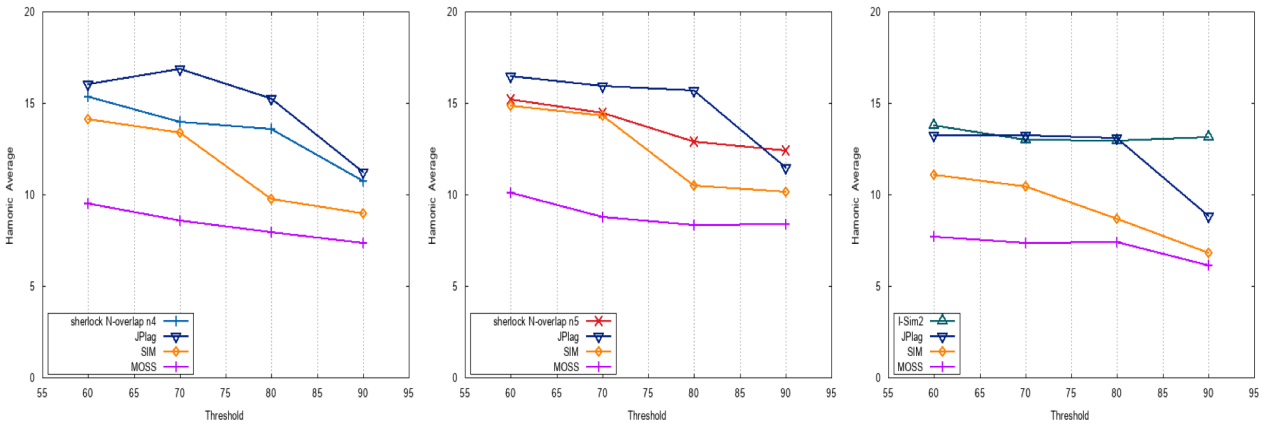
Figura 4.23 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe C (2013) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

Figura 4.24 - Gráficos da soma das médias harmônicas dos códigos desenvolvidos na classe D (2013) por limiar de similaridade



(a) (b) (c)

Legenda: (a) Sherlock N-overlap - normalização 4 sob análise; (b) Sherlock N-overlap - normalização 5 sob análise; e (c) I-Sim2 sob análise.
Fonte: Próprio autor.

Portanto, constata-se que a maior legibilidade, para o ser humano, do código pré-processado devido à conservação de palavras reservadas da linguagem específica não representa, em todas as situações, maior potencial de identificação de similaridades com o uso de ferramentas.

De toda maneira, pela análise dos dados registrados e pela observação dos gráficos gerados, percebe-se que, na maioria das situações, o Sherlock N-Overlap, seja com normalizações 4 ou 5, apresenta resultados superiores às ferramentas JPlag, SIM e MOSS, usadas como oráculos.

Ressalte-se que, na maioria das situações, o MOSS apresenta notória distância das demais ferramentas. Também é possível observar que a variação de limiar não altera de forma significativa o desempenho comparativo em relação aos algoritmos analisados.

Pelos resultados obtidos, destaca-se, por fim, que a ferramenta I-Sim2 apresenta os melhores resultados na maioria dos cenários testados.

4.3 Limitação do método de conformidade

O método de conformidade (FRANCA *et al.*, 2018) foi concebido como alternativa à indisponibilidade de conjuntos de códigos-fonte que contivessem um volume significativo de pares com similaridade previamente conhecida, a fim de que fosse possível avaliar ferramentas como as que são propostas nesta tese.

É preciso considerar que o método se constitui em uma heurística, visto que se apoia na qualidade de um conjunto de ferramentas (oráculos), mas que estas podem, eventualmente, falhar todas em conjunto.

Um exemplo é quando uma ocorrência isolada é, de fato, um verdadeiro positivo não capturado por nenhuma das ferramentas oráculo. Nos experimentos realizados no segundo cenário, foram encontrados alguns casos em que a ferramenta I-Sim2 foi a única a indicar similaridade de maneira correta. Um exemplo pode ser visto na Tabela 4.5.

Pelo método de conformidade, esse caso é caracterizado como uma ocorrência isolada, levando o I-Sim2 a ser penalizado conforme equação da precisão de conformidade (Equação 2.8).

Entretanto, ao analisarmos os códigos com maior cuidado, observa-se que existe boa similaridade entre os códigos comparados, como se pode ver na Figura 4.25.

Tabela 4.5 - Tabela contendo os percentuais de similaridade encontrados pelas ferramentas analisadas.

	596_exe45a.c			
Algoritmo	**I-Sim2**	**JPlag**	**SIM**	**MOSS**
normalização	**-**	**-**	**-**	**-**
519_exercicio45_orig.c	81,82%	0,00%	0,00%	0,00%

Fonte: Próprio autor.

Esta limitação do método, entretanto, não o invalida, principalmente para grandes volumes de dados em que se espera certa regularidade nas indicações de semelhança.

Uma sugestão deixada para trabalhos futuros, entretanto, é tornar o método de conformidade iterativo, de maneira que se possa analisar a *posteriori* se as ocorrências isoladas correspondem, de fato, a uma similaridade real. Neste caso, procede-se à retroalimentação do sistema, registrando os casos de similaridade efetiva e realizando-se novos cálculos.

Figura 4.25 - Comparação entre dois códigos semelhantes.

```c
int main()
{
        int a=0,  isso=0,  c=0,  d=0,  e=0;
        printf( " progama,  ex 45a " );
        printf( " digite o valor de termos desejados " );
        scanf( " % d",  &a );
        for( isso=1; isso <=a; isso+=1 ) {
                c=( isso / 3 ) + 1;
                d=( isso / 3 ) + 4;
                e=( isso / 3 ) + 3;
                if(( isso % 3 ) ==1 )
                {
                        printf( " % i",  c );
                }
                else {
                        if(( isso % 3 ) ==2 )
                        {
                                printf( " % i",  d );
                        }
                        else {
                                printf( " % i",  e );
                        }
                }
        }
}
```

```c
int main()
{
  int a,  b,  x,  y,  z;
  printf( "digite o numero de termos " );
  scanf( " % d",  &b );
  printf( " " );
  for( a=1; a <=b; a++ ){
    if( a % 3 ==1 )
      printf( " % d,  ", ( a / 3 ) +1 );
    else
      if( a % 3 ==2 )
        printf( " % d,  ", ( a / 3 ) +4 );
      else
        printf( " % d,  ", ( a / 3 ) +3 );
  }
  return 0;
}
```

Fonte: Próprio autor.

5 CONCLUSÃO

Contribuindo para a análise de similaridade entre códigos-fonte, nesta tese foram concebidas, desenvolvidas e apresentadas técnicas que recodificam ou transformam os códigos originais para outro domínio de representação antes do processo de comparação.

Os resultados alcançados colocam a principal técnica proposta nesta tese como superior a um conjunto de ferramentas frequentemente referenciadas na bibliografia atual relativa à detecção de plágio, apresentando, ainda, perspectivas de melhoria.

Inicialmente, o algoritmo de comparação Sherlock foi alvo de estudo aprofundado devido às suas características intrínsecas e ao fato de ser um algoritmo de código aberto, de simples entendimento e de fácil modificação.

Diferentemente de outras ferramentas, como o JPlag e o MOSS, muito empregadas para a detecção de plágio em código-fonte, o Sherlock, utilizado como base para o Sherlock N-overlap, não utiliza nenhuma técnica interna de pré-processamento dos documentos de entrada a serem comparados, tampouco pressupõe que regras sintáticas ou léxicas específicas sejam usadas em tais documentos. Essas características viabilizam, portanto, a concepção e utilização de regras próprias de pré-processamento.

Isso permite a transformação ou a recodificação dos documentos de entrada com o objetivo de produzir melhores condições efetivas para a geração de assinaturas e a comparação realizadas pelo Sherlock N-overlap.

A transformação dos documentos a serem comparados pelo Sherlock N-overlap pode ser bastante invasiva, como demonstrado pela normalização 4, que traduz o documento original em outro praticamente ininteligível ao ser humano, mas capaz de preservar rastros significativamente relevantes, os quais permitem manter a identidade do código-original (o código dentro do código-fonte).

Para verificar se um nível de transformação com menor descaracterização do código original é capaz de permitir melhores recursos e condições de comparação pelo Sherlock N-overlap (possibilitando, desse modo, a leitura pelo ser humano), propôs-se, nesta tese, a normalização 5.

Esta nova normalização, ainda que imponha um conjunto rígido de regras, resulta em um código final que conserva estruturas de maior significado para o homem. Entretanto, os testes mostraram que, embora a normalização 5 apresente resultados bons quando

comparados à maioria das ferramentas de referência utilizadas no trabalho, não é evidente a sua superioridade sobre a normalização 4 em todas as situações analisadas.

Os experimentos realizados sobre as duas bases de código utilizadas mostraram uma inversão entre si quanto aos resultados alcançados por essas duas técnicas. Além disso, os resultados da comparação alcançados pelas normalizações 4 e 5 no primeiro cenário são, em geral, muito próximos, se distanciando no segundo cenário. Os resultados sugerem, portanto, que a ilegibilidade pelo homem não implica necessariamente em menor qualidade da comparação por ferramentas.

Com inspiração na técnica proposta pelo Sherlock N-overlap, e continuando as investigações sobre transformações dos elementos originais em outras formas de representação, foi proposta uma nova abordagem: a transformação de códigos-fonte em imagens. Construídas (codificadas) segundo regras específicas e em forma de *treemaps*, as imagens geradas conservam representações das características relevantes dos códigos originais. A ferramenta implementada com a nova abordagem foi denominada I-Sim.

Para a construção da imagem e a atribuição das cores foi proposta uma tabela, na qual funções da linguagem que sejam consideradas semelhantes (funcionalmente) pelo avaliador podem ser mapeadas manualmente e substituídas automaticamente por suas palavras de agrupamento, evitando que estruturas funcionais iguais, como, por exemplo, *if/else* e *switch/case,* recebam cores distintas o que comprometeria a comparação.

Porém, caso essa tabela não seja corretamente configurada pelo usuário, pode levar a erros na comparação, o que irá diminuir o percentual de similaridade entre os códigos comparados.

Para a comparação das imagens geradas, foram propostas duas métricas de similaridade. A primeira métrica de comparação é baseada no cálculo da superfície de recobrimento, com sobreposição *pixel* a *pixel* das imagens comparadas. O I-Sim1, nome atribuído à nova ferramenta que usa essa métrica, mostrou-se limitado por implicar em grande nível de redundância de áreas de sobreposição, requerendo um maior aprofundamento na definição de regras de utilização de cores. Além dessa limitação, o I-Sim1 possui um custo computacional elevado, o que levou à decisão de reconduzir os experimentos para uma segunda estratégia de comparação.

A segunda métrica proposta é baseada no uso de descritores *Color Spectrum*. Nessa técnica, primeiramente é gerado o grafo topológico da imagem. Esse grafo é, então, processado para que seja calculada sua matriz de adjacência. Dessa matriz são, em seguida, extraídos os autovalores que, depois de ordenados, compõem o respectivo descritor.

Cada descritor é usado como uma assinatura, sendo a similaridade entre os códigos determinada pela coincidência de assinaturas, de forma semelhante à usada pelo Sherlock N-overlap. O I-Sim2, nome atribuído à ferramenta com a métrica baseada em extração dos descritores, apresentou resultados, em sua maioria, superiores a todas as demais ferramentas utilizadas e citadas nesta tese.

Assim, considera-se demonstrada a hipótese de ser possível identificar o nível de similaridade entre códigos-fonte originais a partir de sua transformação em outro tipo de código, incluindo a transformação para o domínio das imagens, como proposto para o I-Sim2.

Para comprovação dos resultados, todas as técnicas propostas foram submetidas a testes rigorosos, sendo os resultados comparados aos obtidos pelo JPlag, MOSS e o SIM. Os experimentos e os resultados foram apresentados e discutidos detalhadamente no Capítulo 4.

Como discutido ao final do capítulo acima referido, o método de conformidade fornece uma alternativa importante para validação de ferramentas quando não se tem o conhecimento a *priori* da similaridade entre códigos analisados. Entretanto, sendo baseado na comparação com os resultados de outras ferramentas (oráculos), esse método pode interpretar como um falso positivo uma eventual situação de eficácia superior da ferramenta em avaliação, quando relacionado a todos os oráculos.

Essa situação foi identificada em casos registrados como ocorrência isolada pelo I-Sim2, o que penalizou os seus índices com o uso do método de conformidade. Isso indica que os resultados alcançados pelo I-Sim2, mesmo colocando-o como a melhor entre as ferramentas avaliadas, podem ser ainda superiores aos que foram apresentados.

Tendo em vista as possibilidades de evolução no escopo em que este trabalho se insere, algumas perspectivas de continuidade incluem:

- Aprofundar a investigação sobre as normalizações propostas no contexto do Sherlock N-overlap, propondo novas formas de configuração de palavras de agrupamento, como utilizados na normalização 5, e proposição de formação de novas subestruturas acessórias à recodificação proposta pela normalização 4;
- Investigar novas formulações ou propor mecanismos de retroalimentação para o método de conformidade, visando lidar com o problema da consideração de todas as ocorrências isoladas como falso positivo, mesmo quando se tratam de verdadeiros positivos;

- Investigar outras formas de recodificação de código-fonte em outros tipos de imagens, incluindo as do tipo 3D e a utilização de *beamtrees* (HAM; WIJK, 2003);
- Investigar outras métricas para comparação das imagens referentes às *treemaps*.
- Criar mecanismo para criação automatizada da tabela de agrupamento utilizada no I-Sim2.
- Executar testes com as ferramentas desenvolvidas em um ambiente educacional, observando seu uso através dos agentes envolvidos (professores, tutores, alunos, etc).

REFERÊNCIAS

ALI, A. M. E. T.; DAHWA, H. M. A.; SNÁSEL, V. Overview and comparison of plagiarism detection tools. **CEUR Workshop Proceedings**, Aachen, v. 706, p. 161–172, 2011.

BAKER, B. S. A program for identifying duplicated code. *In:* Computer Science and Statistics: Proceedings of the 24th Symposium on the Interface, 24th, Fairfax Station. **Anais…** Fairfax Station, VA: Interface Foundation of North America., 1993. p. 49–49.

BOWYER, K. W.; HALL, L. O. **Experience using "MOSS" to detect cheating on programming assignments.** Tampa, Flórida: Stripes Publishing L.L.C, 1999. Disponível em: <10.1109/FIE.1999.840376>. Acesso em: 03 ago. 2019.

BRETAG, T. **Handbook of Academic Integrity**. Singapore: Springer Singapore, 2016.

BURROWS, S.; TAHAGHOGHI, S. M. M.; ZOBEL, J. Efficient plagiarism detection for large code repositories. **Software: Practice and Experience**, New York, v. 37, n. 2, p. 151–175, 11 fev. 2007.

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B. (Eds.). **Readings in Information Visualization: Using Vision to Think**. San Francisco,: Morgan Kaufmann Publishers Inc., 1999.

CARNEIRO, G. de F. **SourceMiner:** Um Ambiente Integrado para Visualização Multi-Perspectiva de Software. 2011. 230 f. Tese (Doutorado em Ciências da Computação) – Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Salvador e Universidade Estadual de Feira de Santana, Bahia, 2011.

CARROLL, J. **A Handbook for Deterring Plagiarism in Higher Education.** 2. ed.. Oxford: Oxford Centre for Staff and Learning Development, Oxford Brookes University, 2007.

CHERNOFF, H. The use of faces to represent points in k-dimensional space graphically. **Journal of the American Statistical Association**, Alexandria, v. 68, n. 342, p. 361–368, 1973.

CHUDA, D. *et al.* The Issue of (Software) Plagiarism: A Student View. **IEEE Transactions on Education**, London, v. 55, n. 1, p. 22–28, fev. 2012.

CHUNG, F. R. K. **Spectral Graph Theory**. Providence: American Mathematical Society, 1997.

CLOUGH, P. **Plagiarism in natural and programming languages: an overview of current tools and technologies**. London: Department of Computer Science, 2000.

CORNIC, P. **Software Plagiarism Detection Using Model-Driven Software Development in Eclipse Platform**. Manchester: University of Manchester, 2008.

CORREIA, M.; SANTOS, R. Game-based learning: The use of kahoot in teacher education. *In:* International Symposium on Computers in Education**, SIIE** 2017, Lisboa. **Fórum…Lisboa: Polytechnic Institute of Lisbon,**2018, p. 1–4, 2018.

COSMA, G.; JOY, M. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. **IEEE Transactions on Computers**, Washington, v. 61, n. 3, p. 379–394, mar. 2012.

DE LA TORRE, L. *et al*. The ball and beam system: A case study of virtual and remote lab enhancement with Moodle. **IEEE Transactions on Industrial Informatics**, Taipé, v. 11, n. 4, p. 934–945, 2015.

DIPTI, P. *et al*. Product Review Analysis Tool. **International Journal on Recent and Innovation Trends in Computing and Communication,** Bikaner, v. 4, n. 4, p. 960–963, 2007.

DUCASSE, S.; RIEGER, M.; DEMEYER, S. A language independent approach for detecting duplicated code. *In:* **Proceedings IEEE International Conference on Software Maintenance (ICSM'99).**, 1999, Oxford. **Anais… Oxford: Keble College**, 1999, p. 109–118.

DURIC, Z.; GASEVIC, D. A Source Code Similarity System for Plagiarism Detection. **The Computer Journal**, Oxford, v. 56, n. 1, p. 70–86, jan. 2013.

EVERING, L. C.; MOORMAN, G. Rethinking Plagiarism in the Digital Age. **Journal of Adolescent & Adult Literacy**, Newark, v. 56, n. 1, p. 35–44, set. 2012.

FAIDHI, J. A. W.; ROBINSON, S. K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. **Computers and Education**, Amsterdam, v. 11, n. 1, p. 11–19, 1987.

FEIST, M. D. *et al*. Visualizing Project Evolution through Abstract Syntax Tree Analysis. *In:* **Proceedings IEEE Working Conference on Software Visualization, VISSOFT 2016**, Raleigh. **Anais…** Raleigh: The Institute of Electrical and Electronics Engineers, Inc. 2016. p. 11–20.

FRANCA, A. B. *et al*. Sherlock N-overlap: invasive normalization and overlap coefficient for the similarity analysis between source code. **IEEE Transactions on Computers**, Washington, v. 9340, n. c, p. 1–1, 2018.

FRANÇA, A. B.; MACIEL, D. L.; SOARES, J. M. 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**, Porto Alegre, v. 21, n. 01, p. 91–105, 2013.

GIL, Y. R.; PALMA, Y. DEL C. T.; LAHENS, E. F. Determination of writing styles to detect similarities in digital documents. **International Journal of Educational Technology in Higher Education**, New York, v. 11, n. 1, p. 128–141, jan. 2014.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing. (3. ed**. Upper Saddle River: Prentice-Hall, Inc., 2006.

GREEN, P. *et al*. Same Difference: Detecting Collusion by Finding Unusual Shared Elements.

In: International Plagiarism Conference, 5th, 2012 Newcastle. **Anais**... Newcastle: Science & Technology Research Institute 2012. Disponível em: <http://hdl.handle.net/2299/18509>. Acesso em: 03 ago. 2019.

GRUNE, D. **The software and text similarity tester SIM**. Home Page. Disponível em: <https://dickgrune.com/Programs/similarity_tester/>. Acesso em: 16 nov. 2018.

_____. **The software and text similarity tester SIM**. Manual de Instruções. Disponível em: <https://dickgrune.com/Programs/similarity_tester/TechnReport>. Acesso em: 16 nov. 2018b.

GRUNE, D.; HUNTJENS, M. **Detecting copied submissions in computer science workshops**. Amsterdam: Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit, 1989.

HAGE, J.; RADEMAKER, P.; VAN VUGT, N.. **A comparison of plagiarism detection tools**. Utrecht: Department of Information and Computing Sciences, Utrecht University. Disponível em: <http://www.cs.uu.nl/research/techreps/repo/CS-2010/2010-015.pdf>. Acesso em: 20 dez. 2017.

JOHNSON, J. H. Identifying Redundancy in Source Code using Fingerprints. *In:* Conference of the Centre for Advanced Studies on Collaborative Research, 1993, Toronto. **Anais…** Toronto: Centre for Advanced Studies on Collaborative research, 1993. p. 171–183.

JONYER, I.; APIRATIKUL, P.; THOMAS, J. Source code fingerprinting using graph grammar induction. *In:* Eighteenth International Florida Artificial Intelligence Research Society Conference **(**FLAIRS 2005)**,** 18th, 2005, Florida. **Anais…** Florida: The AAAI Press, 2005. p. 468–473.

JOY, M. *et al*. Source Code Plagiarism—A Student Perspective. **IEEE Transactions on Education**, London, v. 54, n. 1, p. 125–132, fev. 2011.

JOY, M.; LUCK, M. Plagiarism in programming assignments. **IEEE Transactions on Education**, London, v. 42, n. 2, p. 129–133, 1999.

KEIM, D. A. Designing Pixel-Oriented Visualization Techniques: Theory and Applications. **IEEE Transactions on Visualization and Computer Graphics,** New York, v. 6, n. 1, p. 59–735, 2000.

KEIM, D. A.; KRIEGEL, H. Visualization Techniques for Mining Large Databases: A Comparison. **Knowledge Creation Diffusion Utilization**, Sydney, v. 8, n. 6, 1996.

KROKOSCZ, M. Abordagem do plágio nas três melhores universidades de cada um dos cinco continentes e do Brasil. **Revista Brasileira de Educação**, Rio de Janeiro, v. 16, n. 48, p. 745–768, 2011.

LANCASTER, T. **Effective and Efficient Plagiarism Detection**. London,: School of Computing, Information Systems and Mathematics South Bank University, 2003.

LE, T. *et al*. Educating computer programming students about plagiarism through use of a code similarity detection tool. *In:* Learning and Teaching in Computing and Engineering

(LaTiCE 2013), 2013, Macau. **Anais…** Macau: The Institute of Electrical and Electronics Engineers, 2013. p. 98–105.

LEVY, P.; COSTA, C. I. da. **As Tecnologias da Inteligência:** o Futuro do Pensamento na Era da informática**.** São Paulo: Editora 34, 1993.

LI, D. *et al*. One pass preprocessing for token-based source code clone detection. *In:* 2014 IEEE International Conference on Awareness Science and Technology (iCAST 2014), 6th, 2014, Paris. **Anais…** Paris: The Institute of Electrical and Electronics Engineers, 2014.

LUQUINI, E.; OMAR, N. Programming plagiarism as a social phenomenon. *In:* 2011 IEEE Global Engineering Education Conference (EDUCON), 2011, Amman. **Anais**... Amman: The Institute of Electrical and Electronics Engineers, 2011.

MACIEL, D. L. **Sherlock N-Overlap:** Normalização invasiva e coeficiente de sobreposição para análise de similaridade entre códigos-fonte em disciplinas de programação. 2014. 105 f. Dissertação (Mestrado em Engenharia de Teleinformática) − Departamento de Engenharia de Teleinformática, Universidade Federal do Ceará, 2014.

MANOHARAN, S. Personalized assessment as a means to mitigate plagiarism. **IEEE Transactions on Education**, London, v. 60, n. 2, p. 112–119, 2017.

MOZGOVOY, M. Desktop Tools for Offline Plagiarism Detection in Computer Programs. **Informatics in education**, Vilnius, v. 5, n. 1, p. 97–112, jan. 2006.

MOZGOVOY, M.; KARAKOVSKIYZ, S.; KLYUEV, V. Fast and reliable plagiarism detection system. Annual Frontiers In Education Conference, 37th, 2007, Milwaukee. **Anais**... Milwaukee: The Institute of Electrical and Electronics Engineers, out. 2007. Disponível em: <http://ieeexplore.ieee.org/ielx5/4417794/4417795/04417860.pdf?tp=&arnumber=4417860&isnumber=4417795>. Acesso em: 03 ago. 2019.

MUBARAK ALI, A. F.; SULAIMAN, S. A hybrid technique in pre-processing and transformation process for code clone detection. *In:* 2014 Malaysian Software Engineering Conference(MySEC 2014), 8th, 2014, Malaysia. **Anais…** Malaysia: Software Engineering Research Group, 2014. p. 102–107.

NOVAK, M. Review of source-code plagiarism detection in academia. *In:* International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2016), 39th, 2016, Croatia. **Anais…** Croatia: MIPRO Croatian Society, 2016. -, p. 796–801.

NOZAL, C. L. *et al.* An innovative moodle final project management module for bachelor and master's studies. **Revista Iberoamericana de Tecnologias del Aprendizaje**, Santiago de Compostela, v. 8, n. 3, p. 103–110, 2013.

OHMANN, A. Efficient Clustering-based Plagiarism Detection using IPPDC. New York: Saint John's University, 2013.

OLIVEIRA, A. B. de. **Descritor de Forma 2D Baseado em Redes Complexas e Teoria**

Espectral de Grafos. 2016. 76 f. Tese (Doutorado em Ciência da Computação) − Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2016.

OLIVEIRA M. C. R.; MINGHIM, R. Uma Introdução à Visualização Computacional. , Jornada de Atualização em Informática (JAI97), JAI97, XVI, Congresso da SBC, XVII, 1997, Brasília. **Anais...** Brasília: Sociedade Brasileira da Computação, 1997. p. 85–131.

ORTEGO, R. G. *et al.* Fingerprint verification system in tests in moodle. **Revista Iberoamericana de Tecnologias del Aprendizaje**, Santiago de Compostela, v. 7, n. 1, p. 23–30, 2012.

PERNOMIAN, V. A. **Visualização exploratória de dados do desempenho na aprendizagem em um ambiente adaptável**. São Carlos: Universidade de São Paulo, 2008.

PIKE, R.; LOKI. **The Sherlock Plagiarism Detector**. Home Page. Disponível em: < https://github.com/diogocabral/sherlock>. Acesso em: 04 ago. 2019.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. Finding Plagiarisms among a Set of Programs with JPlag. **Journal Of Universal Computer Science**, Graz, v. 8, n. 11, p. 1016–1038, 2002.

RAGKHITWETSAGUL, C.; KRINKE, J.; CLARK, D. A comparison of code similarity analysers. **Empirical Software Engineering**, New York, v. 23, n. 4, p. 2464–2519, 25 ago. 2018.

RODRIGUES, D. B. **Teoria Espectral e o Problema de Isomorfismo de Grafos Regulares**. 2011. 84 f. Dissertação (Mestrado em Iformática) − Centro Tecnológico, Unversidade Federal do Espírito Santo, Vitória 2011.

ROY, C. K.; CORDY, J. R. A Survey on Software Clone Detection Research. **Queen's University School Of Computing** , Ontario, v. 115, TR n. 2007-541, 2007.

SABI, Y.; HIGO, Y.; KUSUMOTO, S. Rearranging the order of program statements for code clone detection. *In:* IEEE International Workshop on Software Clones, co-located with SANER 2017 (IWSC 2017) 11th, 2017,Klagenfurt. **Anais…** Klagenfurt: Springer Journal Empirical Software Engineering, 2017. p. 15–21.

SANTOS, D. F. D. D. *et al*. Combining color and topology for partial matching. *In:* International Conference on Tools with Artificial Intelligence (ICTAI 2012), 24th, 2012, Athens. **Anais…** Athens: The Institute of Electrical and Electronics Engineers, 2012. v. 1, p. 770–777.

SANTOS, D. F. D. dos. **Recuperação de imagens:** similaridade parcial baseada em espectro de grafo e cor. 2012. 88 f. Dissertação (Mestrado em Ciência da Computação) − Faculdade de Computação, Universidade Federal de Uberlândia, Uberlândia, 2012.

SANTOS, P. L. F. dos. **Teoria Espectral de Grafos Aplicada ao Problema de Isomorfismo de Grafos**. 2010. 71 f. Dissertação (Mestrado em Informática) − Departamento de Informática, Universidade Federal do Espírito Santo, Vitória, 2010.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: Local Algorithms for Document Fingerprinting. *In:* ACM SIGMOD international conference on on Management of data (SIGMOD '03), 2003, San Diego. **Anais…** San Digo: Association for Computing Machinery, 2003. p. 76–85.

SHNEIDERMAN, B. Tree visualization with tree-maps. **ACM Transactions on Graphics**, New York, v. 11, n. 1, p. 92–99, 1992.

SONEGO, A. H. S. *et al.* Use of moodle as a tool for collaborative learning: A study focused on wiki. **Revista Iberoamericana de Tecnologias del Aprendizaje**, Santiago de Compostela v. 9, n. 1, p. 17–21, 2014.

STAMATATOS, E. Plagiarism detection based on structural information. *In:* ACM international conference on Information and knowledge management (CIKM '11), 20th, 2011, Glasgow. **Anais…** Glasgow: University of Glasgow, 2011. p. 1221.

THURMOND, B. H. **Student plagiarism and the use of a plagiarism detection tool by community college faculty**. Indiana: Proquest, Umi Dissertation Publishing, 2011.

VAN HAM, F.; VAN WIJK, J. J.. Beamtrees: compact visualization of large hierarchies. **Information Visualization**, Thousand Oaks, v. 2, n. 1, p. 31, 2003.

VOGTS, D. Plagiarising of source code by novice programmers a "cry for help"? *In:* 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - (SAICSIT '09), 2009, Johannesburg. **Anais…** Johannesburg: Association for Computing Machinery, 2009. n. October, p. 141–149.

WEBER-WULFF, D. **False Feathers**. Berlin: Springer Berlin Heidelberg, 2014.

WEBER-WULFF, D.; KÖHLER, K.; CHRISTOPHER, M. Collusion Detection System Test Report. **Plagiarism Portal**,Berlin, 09 nov. 2012. Disponível em: <http://plagiat.htw-berlin.de/collusion-test-2012/>. Acesso em: 20 jul. 2018.

WHALE, G. Identification of program similarity in large populations. **The Computer Journal**, Oxford, v. 33, n. 2, p. 140–146, 1990.

WISE, M. J. **YAP3:** improved detection of similarities in computer program and other texts. *In:* twenty-Seventh SIGCSE Technical Symposium On Computer Science Education, 27th, 1996, Philadelphia. **Anais**... Philadelphia: Association for Computing Machinery, 1996.

ZHAO, J. *et al*. An AST-based Code Plagiarism Detection Algorithm. *In:* International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA 2015), 10th, 2015, Krakow. **Anais…** Krakow: Pedagogical University of Cracow, 2015. p. 178–182.

APÊNDICE A – TABELA COM AS PALAVRAS DE LINGUAGEM E SUAS CORES CORRESPONDENTES

Tabela A.1 - Palavras de linguagem e suas cores correspondentes

NÓ	RGB	COR	NÓ	RGB	COR
call	255, 130, 071		while	255, 255, 0	
call_io	191, 191, 191		cond	0, 255, 0	
call_math	127, 127, 0		if	0, 255, 0	
call_string	0, 127, 127		switch	0, 255, 0	
call_file	255, 0, 255		struct	127, 0, 0	
call_array	255, 218, 185		else	0, 127, 0	
call_system	000, 238, 238		function	0, 255, 255	
call_threads	125, 038, 205		return	0, 0, 255	
stmnts	127, 127, 127		typedef	127, 0, 127	
stmts	127, 127, 127		for	0, 0, 127	
arg	100, 0, 0		goto	238, 238, 0	
do	255, 255, 0		main	0, 0, 0	

Fonte: Próprio autor.

APÊNDICE B – ENUNCIADOS DOS PROBLEMAS PLAGIADOS PROPOSITALMENTE

No primeiro cenário, a análise foi feita em um conjunto de códigos controlados que foram propositadamente plagiados utilizando critérios pré-estabelecidos explicitados na Subseção 4.1.1. Este cenário, em que a situação de cada par de códigos era previamente conhecida, foi utilizado para avaliar as ferramentas, utilizando o método de precisão e revocação.

Para este experimento, três programadores desenvolveram um conjunto de códigos na linguagem C. Cada programador recebeu duas tarefas de programação pré-programadas diferentes para as quais ele teria que desenvolver seu código.

Os códigos e as questões foram separados em dois grupos: A e B. O grupo A possui as questões mais simples que resultam em códigos pequenos, compreendendo entre 10 e 20 linhas. O grupo B tem as outras questões, que foram um pouco mais difíceis e que resultaram em códigos contendo entre 60 e 70 linhas.

As questões são descritas abaixo:

1. Questões utilizadas pelo programador 1:
    a. Questão 1 (Grupo A): Crie um jogo simples de “adivinhe o número”. O programa irá gerar um número aleatório. 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.
    b. Questão 2 (Grupo B): Desenvolver um programa que realize as seguintes operações em uma estrutura representando 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 expõe os recursos acima.

2. Questões utilizadas pelo programador 2:

a. Questão 1 (Grupo A): Crie um programa que leia dez números inteiros e imprima o maior e o segundo maior número da lista.
b. Questão 2 (Grupo B): Construa um programa para calcular a média de valores pares e ímpares de 10 números que serão digitados pelo usuário. No final, o programa deve mostrar essas duas médias. O programa também deve mostrar o número par mais alto e o menor número ímpar digitado. Esses dados devem ser armazenados em um vetor. Além disso, os valores pares ainda maiores que a média devem ser impressos, assim como valores ímpares menores que a média ímpar. Além disso, os números pares maiores que a média dos números pares e os valores ímpares menores que a média dos números ímpares devem ser impressos.

3. Questões utilizadas pelo programador 3:
    a. Questão 1 (Grupo A): Cria um programa que lê uma string via console e imprime a mesma string toda em maiúscula.
    b. Questão 2 (Grupo B): Cria um programa que exibe e atualiza um relógio de software que mostra horas, minutos e segundos. Neste programa, o atraso para alterar estados deve ser definido a *priori*, mas não precisa manter uma relação direta com o tempo real.