UM MÉTODO DE DETECÇÃO DE PLÁGIO EM CÓDIGOS-FONTE PARA DISCIPLINAS INICIAIS DE PROGRAMAÇÃO

ADRIA MENEZES DE OLIVEIRA

UM MÉTODO DE DETECÇÃO DE PLÁGIO EM CÓDIGOS-FONTE PARA DISCIPLINAS INICIAIS DE PROGRAMAÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: David Fernandes de Oliveira

Manaus

Dezembro de 2016

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Oliveira, Adria Menezes de
O48u Um método de detecção de plágio em códigos-fonte para disciplinas iniciais de programação / Adria Menezes de Oliveira. 2016
64 f.: il. color; 31 cm.

Orientador: David Braga Fernandes de Oliveira
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Plágio. 2. Detecção de Plágio. 3. Códigos-fonte. 4. Disciplinas iniciais de programação. I. Oliveira, David Braga Fernandes de II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Um Método de Detecção de Plágio em Códigos-Fonte para Disciplinas Iniciais de Programação"

ADRIA MENEZES DE OLIVEIRA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Marco Antonio Pinheiro de Cristo - PRESIDENTE

Prof. David Braga Fernandes de Oliveira - MEMBRO EXTERNO

Prof. André Luiz da Costa Carvalho - MEMBRO EXTERNO

Prof. Thierson Couto Rosa - MEMBRO EXTERNO

Manaus, 22 de Dezembro de 2016

Dedicado àqueles que sempre me ensinaram a acreditar em meus sonhos e que trabalharam muito para que eu pudesse realizá-los, meus pais, Leopoldo e Rosane, e minha avó Elina.

Agradecimentos

Agradeço primeiramente a Deus, por ter me iluminado e me abençoado nessa caminhada, sendo meu refúgio e fortaleza nos momentos mais difíceis. A Ele, minha eterna gratidão.

Agradeço especialmente aos meus pais, Leopoldo e Rosane, e a minha avó Elina, por sempre acreditarem em minha capacidade e me acharem a melhor de todas. Isso deu força para que eu me tornasse a melhor que poderia ser. Vocês são pessoas fundamentais e eu os amo muito. Obrigada por existirem em minha vida!

Agradeço àquele que, com toda paciência e dedicação, conduziu-me durante esta jornada. Aquele que, mesmo sem me conhecer, acreditou em meu potencial e se tornou um dos responsáveis diretos por esta conquista, meu querido orientador, Prof. David Fernandes de Oliveira. A você, ofereço minha eterna gratidão, respeito e admiração.

Agradeço ao meu parceiro de vida, melhor amigo e companheiro de todas as horas, Dorivan Almeida, por todo amor, carinho, paciência e compreensão que tem me dedicado.

Um agradecimento muito especial aos professores do IFAM, Francisco Mendes, Jucimar Brito e Marcelo Chamy que sempre me inspiraram e que são verdadeiros exemplos de dedicação e competência. Certamente, nada disso teria sido possível sem o incentivo de vocês. Obrigada, de coração!

Agradeço também a todos os professores do PPGI pelo conhecimento adquirido e os laços de amizade que formamos. A todos os meus colegas de mestrado e, de forma especial, aos amigos que fiz nesta jornada: Adriana Rodrigues, Caio Gregoratto, Márcio Palheta, Michel Yvano, Pablo Elleres e Thais Almeida, pelo companheirismo nas horas mais difíceis e pela atenção dedicada ao compartilharem seus conhecimentos. Aos amigos Grenny Souza e Jordan Queiroz, que tanto contribuíram, de forma direta ou indireta, deixo os meus sinceros agradecimentos e reconhecimento.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo auxílio financeiro concedido.

A todos vocês que me ajudaram, quero que sintam-se parte desta conquista, pois

é desta forma que os vejo. E, mais uma vez, muito obrigada!

"Always pass on what you have learned."
(Mestre Yoda)

Resumo

A prática do plágio é um problema grave e crescente no meio acadêmico, que interfere na qualidade da produção científica, compromete a credibilidade do processo de autoria e ameaça a reputação de instituições de ensino e pesquisa. Esta pesquisa se contextualiza no problema da detecção de plágio entre códigos-fonte desenvolvidos por turmas iniciais de programação. Com o aumento do número de alunos nas instituições, a identificação manual de irregularidades nas avaliações dessas turmas se tornou uma prática que sobrecarrega os professores/monitores. Apesar da ampla quantidade de ferramentas disponíveis para a detecção de plágio, poucas são capazes de identificar, de maneira eficaz, o plágio em códigos desenvolvidos por alunos iniciantes em programação, uma vez que esse códigos solucionam questões simples, e, portanto, geram pequenas soluções. Diante disso, esta dissertação apresenta um método para detectar plágios em atividades das disciplinas iniciais dos cursos de programação, com o intuito de favorecer a confiabilidade dos resultados no processo de ensino-aprendizagem. O método proposto apresenta resultados satisfatórios em comparação aos métodos disponíveis na literatura relacionados ao tema.

Palavras-chave: Plágio, Detecção de Plágio, Códigos-fonte, Disciplinas iniciais de programação.

Abstract

The practice of plagiarism is a growing and serious problem in academic environment, which interferes on the quality of scientific production, affects the credibility of authoring process and threats the reputation of research institutions. This research focus on the problem of plagiarism detection in source codes developed in the initial programming classes. With the growing number of college students, the manual identification of inconsistency in the evaluations of these classes has become a practice that overcome the teachers/monitors. Despite the wide variety of available tools for plagiarism detection, only few tools are able to effectively identify plagiarism on codes developed by novice programmers, since, in most cases, these codes solve simple questions, and thus generate small solutions. Therefore, this dissertation presents a method for detecting plagiarism in activities in the initial disciplines of programming courses, in order to promote the reliability of results in the teaching-learning process. The proposed method provides satisfactory results in comparison to the methods available in the literature related to the topic.

Keywords: Plagiarism, Plagiarism Detection, Source Codes, Initial Programming Disciplines.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O plágio pode ser definido como o ato de furtar ou se apropriar de modo ilícito do conteúdo existente em uma obra científica ou literária, sem dar os devidos créditos ao autor [Gil et al., 2014]. Esse ato é frequente nas diversas áreas do conhecimento como educação, artes, fotografia, música, publicidade e etc, muitas vezes pelo fato do indivíduo desconhecer que tal ato é crime [Ali et al., 2011; Gil et al., 2014].

O dicionário Aurélio [Ferreira, 2009] define o plágio como "1. A apresentação de obras alheias como sendo de sua própria autoria. 2. A reprodução de trabalho alheio". Plágios podem ser feitos com diferentes intenções. No ambiente acadêmico, a prática do plágio é um problema conhecido e difundido [Vamplew & Dermoudy, 2005]. Estudos revelam que cerca de 90% dos alunos vivenciaram experiências relacionadas com o plágio [Sheard et al., 2002], o que faz do plágio um tópico importante a ser estudado e muito comum nos dias de hoje.

O plágio não ocorre somente em documentos textuais, imagens e letras musicais, mas também em códigos-fonte. De acordo com Vogts [2009], os cursos de Introdução à Programação não estão livres de plágio. Em exercícios de programação é comum encontrar entre as respostas dos alunos, cópias idênticas ou minimamente alteradas na tentativa de diferenciá-los dos originais, com o intuito de dificultar a percepção da ação por parte de professores e/ou monitores.

Joy & Luck [1999], Sheard et al. [2002] e Vamplew & Dermoudy [2005] apontam os principais motivos que levam os alunos iniciantes em programação a plagiar códigos-fonte: (i) trabalhos práticos que exigem um nível de abstração maior; (ii) tempo de entrega insuficiente; (iii) medo de fracassar; e (iv) recursos de software e hardware inadequados.

Alguns métodos como JPlag [Prechelt et al., 2002], GPlag [Liu et al., 2006] e AuDeNTES [Mariani & Micucci, 2012] têm surgido com o objetivo de automatizar o

processo de detecção de plágio, indicando o grau de similaridade entre pares de código-fonte entregues por diferentes pessoas. Caso a similaridade seja considerada alta, é necessário uma checagem manual, por um especialista, para a confirmação da prática de plágio.

Embora esses métodos apresentem bons resultados em grandes sentenças de código, seu desempenho em códigos menores, como programas desenvolvidos por iniciantes em programação, tende a ser insatisfatório. Nestes casos, os códigos tendem a ser muitos parecidos, mesmo quando codificados por alunos diferentes, o que reduz a quantidade de informações necessárias ao cálculo de similaridade e, por consequência, a eficácia dos métodos propostos até então pela literatura.

Desta maneira, esta dissertação propõe um método que permita aos professores e/ou monitores detectar quais questões possuem maior incidência de similaridade e possam ser possíveis plágios. É importante salientar que essas questões são resolvidas por alunos das disciplinas introdutórias de programação.

A prática do plágio pode ser classificada em dois tipos: (i) intra-corpal, quando dois indivíduos devem realizar a mesma tarefa e um deles copia o trabalho do outro; e (i) extra-corpal, quando o indivíduo copia as informações de obras já concluídas através de fontes externas, tal como a Internet. Esta dissertação se contextualiza no problema de detecção de plágio entre os trabalhos dos próprios alunos, isto é, análise intra-corpal.

O método proposto nesta pesquisa foi comparado com JPlag [Prechelt et al., 2002], AuDeNTES [Mariani & Micucci, 2012] e CodeCompare [Tao et al., 2013], métodos conhecidos e destacados na literatura sobre detecção de plágio. Conforme será apresentado no Capítulo 5, os resultados obtidos nos experimentos mostram que o método proposto apresenta resultados satisfatórios em comparação a tais métodos.

1.1 Definição do Problema

Considere um cenário em que um professor de Introdução a Programação tenha elaborado uma lista com questões a serem resolvidas individualmente pelos alunos de sua turma. É necessário considerar também que, por ser uma disciplina introdutória, a maioria dessas questões levam a soluções muito simples, que podem ser desenvolvidas com poucas linhas de código.

Supondo que o professor dessa hipotética turma já tenha passado por problemas com trocas de códigos-fonte entre alunos de turmas anteriores, ele resolveu recorrer a uma das inúmeras técnicas de detecção de plágio disponíveis na literatura. No entanto, a maioria das técnicas propostas na literatura adota alguma métrica de similaridade

para identificar as ocorrências de plágio. Em virtude disso, essas técnicas enfrentam dois tipos de problemas: (i) como os códigos desenvolvidos pelos alunos tendem a ser muito simples e possuir poucas linhas de código, as técnicas baseadas em similaridade possuem poucas evidências para o cálculo de similaridade, e (ii) como os códigos desenvolvidos pelos alunos para a mesma questão tendem a ser muito similares, essas técnicas possuem poucas evidências para discriminar um código de outro.

O método descrito nesta dissertação propõe a detecção de plágio neste tipo de cenário, onde existe uma lista com questões propostas por um professor e cada aluno da turma deverá resolver as questões desta lista. É importante destacar que o método proposto usa informações de todos os códigos-fonte desenvolvidos por um aluno para as questões propostas, a fim de tentar identificar se algum desses códigos foi plagiado de outro aluno. Portanto, o método usa mais informações que os demais métodos existentes na literatura, visto que ele usa os códigos elaborados pelos alunos para todas as questões, e não apenas as informações de um único código-fonte. Além disso, esse método tenta identificar os estilos comuns à determinada questão, com o intuito de não considerar como plágio fragmentos inerentes a essa questão.

1.2 Hipótese e Questão de Pesquisa

Q1 - Será que as informações de estilo aluno, questão e professor podem ser utilizadas para melhorar o processo de detecção de plágio entre trabalhos de alunos iniciantes em programação?

H1 - A hipótese é que informações de estilo de escrita de código-fonte podem ser utilizadas para melhorar a estimativa da importância dos termos na coleção. De maneira mais específica, os estilos comuns entre essas classes devem ser menos evidenciados.

1.3 Objetivos

1.3.1 Objetivo Geral

Propor e validar um método para detectar possíveis ocorrências de plágios entre questões resolvidas por alunos das disciplinas iniciais dos cursos de computação, a fim de proporcionar ao professor uma melhor tomada de decisão quanto à avaliação destas questões.

1.3.2 Objetivos Específicos

1. Propor e validar um método para detectar possíveis plágios de forma automática e eficaz em ambientes acadêmicos;

2. Desenvolver um método para detectar possíveis plágios em programas desenvolvidos por iniciantes em programação, onde a solução dos problemas tende a ser muito parecida;

3. Criar e disponibilizar uma coleção de códigos-fonte com dados reais de programadores iniciantes para futuras comparações com outros métodos de detecção de plágio.

1.4 Contribuições Esperadas

Esta dissertação pretende contribuir com a criação de um método para detecção de plágio em códigos-fonte, e com a criação de uma coleção de códigos-fonte com dados reais de alunos iniciantes em programação.

1.5 Organização da Dissertação

Esta dissertação está organizada em seis Capítulos. O Capítulo 2 apresenta os conceitos fundamentais para a compreensão do tema desta pesquisa. No Capítulo 3 é apresentada a revisão da literatura acadêmica sobre as técnicas de detecção de plágio e são apresentados os principais trabalhos relacionados.

O Capítulo 4 apresenta o método proposto nesta dissertação. O processo de criação das coleções de códigos, os experimentos realizados e os resultados obtidos são explicados no Capítulo 5. Por fim no Capítulo 6, são realizadas as considerações finais, e as perspectivas de trabalhos para a continuidade desta pesquisa.

Capítulo 2

Noções Preliminares

Este capítulo apresenta os conceitos e terminologias fundamentais para a compreensão do tema desta pesquisa. Nessa perspectiva, são abordados o conceito de plágio, plágio em códigos-fonte, os fundamentos de recuperação de informação e dois modelos clássicos: o Vetorial e o Probabilístico BM-25. Além disso, é demonstrada uma visão sobre detecção de plágio baseada em recuperação de informação.

2.1 Definição de Plágio

O plágio vai além de uma simples cópia de códigos ou informações, e em virtude disso são apresentadas diferentes visões encontradas na literatura. Um plágio ocorre quando você usa palavras e ideias de alguém sem lhe dar o devido crédito, e considera o trabalho como seu [Park, 2004]. De acordo com Michaelis [2008], o plágio se assemelha ao ato de furtar. Se o furto de um carro, telefone celular e outros pertences é punível por lei, o furto de ideias, palavras ou pensamentos também pode ser considerado ilícito. Considerando o meio acadêmico, isso não significa que alunos e pesquisadores não possam ver obras de outras fontes ou referências. Park [2004] cita que aumentar o conhecimento vendo outras opiniões e ideias de terceiros pode ser um fato positivo, porém é necessário se certificar que as referências estão sendo citadas devidamente.

Para Moraes [2004], a imitação de uma obra pode ser reconhecida como plágio, o que é considerado um ato ilícito. A reutilização dos próprios materiais também caracteriza uma violação aos direitos autorais e é uma prática conhecida como auto-plágio. A apresentação de dois trabalhos semelhantes também pode caracterizar plágio, mesmo no caso em que seus autores coproduziram o trabalho [Badge & Scott, 2009].

Existem casos nos quais sérias ações judiciais foram iniciadas pelo fato de se ter comprovado o plágio de trabalhos. Como exemplo, em 2011, o Ministro da Defesa da

Alemanha, Karl-Theodor zu, renunciou ao cargo em função de uma denúncia de que ele havia cometido plágio em sua tese de Doutorado [Pithan & Vidal, 2013]. No entanto, a maneira de penalizar o plágio obedece às regras de cada país. No Brasil, o problema do plágio e da cópia é tratado em conformidade com a lei 9.610, de 19 de fevereiro de 1998, que dispõe sobre os direitos autorais.

No meio acadêmico, a prática do plágio é um problema grave que interfere na qualidade da produção científica, compromete a credibilidade do processo de autoria e ameaça a reputação de instituições de pesquisa. Silva [2008] cita que o plágio em trabalhos acadêmicos é crime e a penalidade pode ser a não obtenção do grau acadêmico.

Ali et al. [2011] apresenta uma taxonomia que divide o plágio em dois tipos: (i) plágio em documentos de texto: comumente utilizado por estudantes ou pesquisadores em ambientes acadêmicos, onde os documentos são copiados na íntegra ou parcialmente a partir de relatórios, redações e trabalhos científicos, e (ii) plágio em códigos-fonte: frequentemente utilizado por estudantes em universidades, onde esses fazem uso parcial ou total do código-fonte de outra pessoa, como sendo de sua própria autoria.

A seguir, será abordado sobre plágio em códigos-fonte, que é o foco desta dissertação.

2.1.1 Plágio em códigos-fonte

Há algumas diferenças entre plágios de conteúdos textuais e plágios de códigos-fontes de linguagens de programação. Detectar plágio em linguagens naturais é um processo mais difícil, tendo em vista que a linguagem natural é muito mais complexa e ambígua, enquanto a gramática da linguagem de programação pode ser definida e especificada. Cosma & Joy [2006] citam que, quando o assunto é código-fonte, são consideradas como plágio as seguintes atitudes: (i) reutilização de código fonte sem fornecer referência adequada; (ii) conversão parcial ou total do código-fonte de um autor para outra linguagem de programação; (iii) geração automática de código-fonte por um programa especificado para isso, se esta prática não for permitida; e (iv) o pagamento para outra pessoa desenvolver o código.

Detectar plágio em códigos-fonte não é uma tarefa trivial, devido ao fato do grande número de formas de se "camuflar" as similaridades. No meio acadêmico, alunos podem gerar modificações simples, como alterar nomes das variáveis, ou mais complexas, como a alteração da estrutura do código em si, modificando também a sintaxe[1]. Se o plágio se constituísse apenas de duplicação de código-fonte não modificado,

[1] Conjunto de regras que regem a escrita de uma linguagem de programação.

a detecção seria extremamente simples. Faidhi & Robinson [1987] foram os primeiros autores a estudar essas modificações, ilustrando-as conforme a Figura 2.1.

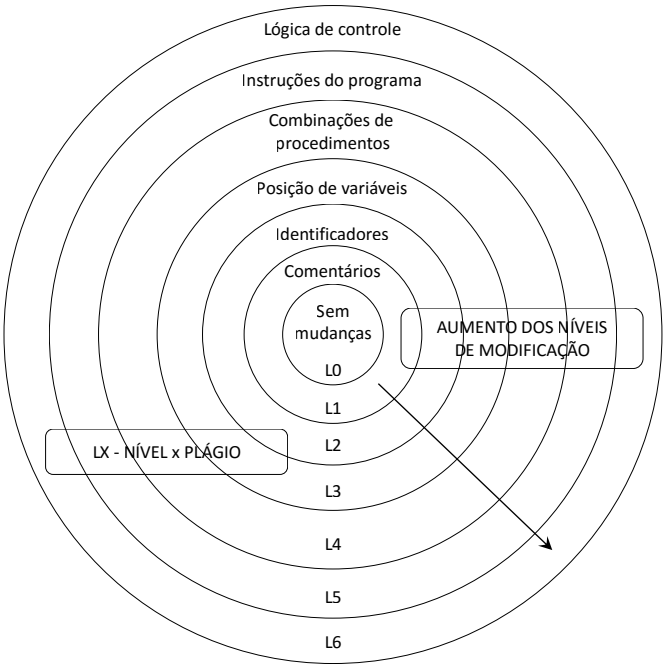

Figura 2.1: Componentes da complexidade na detecção de plágio. Fonte: Adaptado de Faidhi & Robinson [1987].

A Figura 2.1 apresenta, de dentro para fora, os níveis de plágio de acordo com o tipo de modificação empregada: $L0$: corresponde a códigos sem modificação; $L1$: corresponde a modificação de comentários no código; $L2$: corresponde a alteração de identificadores (por exemplo, os nomes das variáveis); $L3$: corresponde a mudança das posições de variáveis (por exemplo, tornar uma variável global em local); $L4$: equivale a modificações de combinação de procedimentos (por exemplo, trocar trechos de código por funções); e $L5$: equivale a alterações nas instruções (por exemplo, substituir um operador por outro similar).

Whale [1990] identificou doze técnicas de disfarce que devem ser consideradas primordiais pelos sistemas de detecção de plágio: (1) mudar os comentários ou a formatação; (2) mudar os identificadores; (3) mudar a ordem dos operandos nas expressões; (4) mudar tipos de dados (por exemplo, *integer* para *float*); (5) substituir as expressões por expressões equivalentes (por exemplo, *while* (1) para *while* (true)); (6) adicionar variáveis ou instruções redundantes; (7) mudar a ordem das funções ou procedimentos; (8) mudar as estruturas das instruções de iteração (por exemplo, *while* para *for* ou *repeat* para *while*); (9) mudar as estruturas das instruções de seleção (por exemplo, *if* para *case*); (10) excluir chamadas a procedimentos e utilizá-los diretamente no corpo

do código-fonte; (11) introduzir novas instruções não estruturadas; e (12) combinar fragmentos do programa copiado com o original.

Whale [1990] agrupou esses disfarces nas seguintes categorias: (i) alteração nos comentários e na formatação do código; (ii) alteração nos identificadores ou tipos de dados; (iii) substituição da estrutura das instruções de iteração ou de seleção, trocando as chamadas de procedimento por instruções no corpo do procedimento; (iv) alteração na ordem dos operandos; e (v) inclusão de variáveis ou instruções redundantes.

Apesar das técnicas de disfarce apresentadas na tentativa de dissimular o plágio, ele pode ser descoberto por meio da análise do professor. Entretanto, a detecção manual somente é viável em pequenas turmas de alunos. No caso de uma análise de plágios de um único exercício de programação proposto para uma turma de 40 alunos, seria necessário comparar 780 pares de códigos para que todos os arquivos fossem verificados entre si. Segundo Mariani & Micucci [2012], o cálculo para o número de comparações necessárias para um único exercício de programação é dado por:

$$\frac{n \times (n-1)}{2}$$

onde n é o número total de alunos da turma.

Através desse exemplo é possível perceber que a automatização do processo de detecção de plágio é bastante útil para o professor, visto que a análise manual requer muito esforço e tempo disponível.

2.2 Recuperação de Informação

Segundo Baeza-Yates & Ribeiro-Neto [2013], Recuperação de Informação (RI) é uma área da Ciência da Computação que lida com processos de representação, armazenamento e busca de informações, e que tem por principal finalidade facilitar o acesso a informações relevantes às necessidades dos usuários. Um processo de RI inicia quando o usuário de um sistema fornece uma consulta (*query*) formulada a partir de uma necessidade de informação. O sistema, por sua vez, compara a consulta com os dados de uma coleção (*corpus*) e sua tarefa é retornar documentos que satisfaçam a necessidade de informação expressa pela consulta. Um dos conceitos mais abordados na área de RI é a noção de similaridade. Nesta pesquisa, são utilizadas técnicas de cálculo de similaridade entre códigos-fonte para detectar plágios. Nas próximas seções, são apresentados os principais componentes de um sistema de RI.

2.2.1 Máquinas de Busca

Segundo Croft et al. [2010], uma máquina de busca consiste na aplicação prática de técnicas de recuperação de informação em grandes coleções de texto. As máquinas de busca mais conhecidas são as da *web*. Essas máquinas estão presentes em várias aplicações, tais como: *smartphones*, e-mails, aplicações comerciais, etc. Conforme ilustrado na Figura 2.2, a arquitetura de uma máquina de busca para a *web* é composta pelos seguintes componentes:

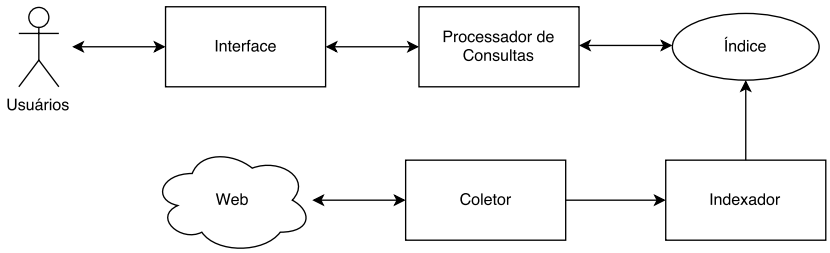

Figura 2.2: Arquitetura de um sistema de recuperação de informação para a *web*. Adaptado de Baeza-Yates & Ribeiro-Neto [2013]

(i) Interface: é o componente responsável por receber as requisições solicitadas pelos usuários. O usuário informa ao sistema uma consulta em forma de texto, que representa a informação da qual ele necessita, e como resposta é retornado um conjunto de documentos potencialmente relevantes à consulta realizada.

(ii) Processador de consultas: é o componente responsável por identificar e recuperar documentos na coleção como resposta a determinada consulta. Esses documentos são ordenados conforme uma estimativa que tenta predizer a relevância de cada documento em relação à consulta submetida. Segundo Arasu et al. [2001], esse componente retorna à interface um conjunto ordenado de documentos, denominado *ranking*, no qual o documento no topo desse *ranking* é o que tem maior chance de relevância, e o último é o que tem menor chance de relevância em relação a consulta informada pelo usuário.

(iii) Índice: é o componente encarregado por armazenar e possibilitar o acesso aos indicadores de ocorrências de termos na coleção de documentos [Baeza-Yates & Ribeiro-Neto, 2013]. De modo geral, o índice consiste em um *arquivo invertido* contendo, para cada termo da coleção, uma lista de documentos nos quais o termo ocorre, bem como a sua frequência.

(iv) Indexador: é o componente encarregado por realizar a análise sintática dos documentos, extrair os termos e suas respectivas frequências e criar o *arquivo invertido*. A função do indexador, segundo Baeza-Yates & Ribeiro-Neto [2013], é organizar os dados extraídos, de modo a garantir o acesso rápido a eles.

(v) Coletor: o coletor de páginas *web* (*crawler*, *spider* ou *robot*) é o componente responsável por encontrar e coletar páginas da *web* de maneira automática. Essa coleta é feita através do envio de requisições a servidores remotos e, em seguida, o conteúdo das páginas é retornado.

Os processos de indexação e processamento de consultas são explicados detalhadamente nas próximas seções.

2.2.1.1 Indexação

De acordo com Baeza-Yates & Ribeiro-Neto [2013], no processo de indexação é gerada a estrutura fundamental de uma máquina de busca, frequentemente chamada de *índice* ou *arquivo invertido*. O componente indexador processa cada documento coletado pelo módulo coletor, extrai os termos (palavras) e suas frequências na coleção de documentos.

O *índice* gerado durante o processo de indexação consiste em dois componentes principais: o vocabulário e as listas invertidas. O vocabulário contém todos os termos (palavras) distintos da coleção de documentos e as listas invertidas mantêm a frequência com que cada termo t, do vocabulário, ocorre em cada documento da coleção.

A fim de ilustrar, considere a coleção de documentos apresentada na Tabela 2.1. Para esse exemplo, a coluna ID representa a chave dos documentos, a coluna Documento representa o nome do documento, e a coluna Termos representa os termos contidos em cada documento.

Tabela 2.1: Exemplo de coleção de documentos.

ID	Documento	Termos
1	D1	maçã maçã maçã laranja
2	D2	maçã maçã uva
3	D3	maçã maçã
4	D4	laranja laranja

A Tabela 2.2 apresenta o *índice* gerado a partir da coleção de documentos mostrada na Tabela 2.1. O vocabulário é constituído por três termos: maçã, laranja e uva. Cada item de uma lista invertida é uma tupla ordenada do tipo *d-tf*, onde d indica o *id* do documento e *tf* indica a frequência do termo t no documento d. Dessa forma, verifica-se que o termo "maçã" ocorre 3 vezes no documento de *id* 1 ($D1$), 2 vezes no documento de *id* 2 ($D2$), e 2 vezes no documento de *id* 3 ($D3$).

De acordo com Witten et al. [1999] e Trotman [2003], essa técnica tradicional de indexação torna as máquinas de busca altamente escaláveis para grandes coleções de dados. Cada termo é armazenado uma única vez no vocabulário e as listas invertidas

Tabela 2.2: Exemplo de um índice para um vocabulário de três termos.

Vocabulário	Listas Invertidas
maçã	(1,3) (2,2) (3,2)
laranja	(1,1) (4,2)
uva	(2,1)

são compostas de números inteiros, que podem ser processados rapidamente. Além disso, as listas são altamente compressíveis.

Um dos principais desafios na construção das listas invertidas é o seu tamanho. De acordo com Anh & Moffat [2005], termos que ocorrem em muitos documentos da coleção apresentam listas invertidas muito longas, o que acarreta em um alto custo ao processar consultas com esses termos. Portanto, alguns sistemas de RI adotam estratégias para reduzir o número de termos distintos. Algumas estratégias comumente utilizadas são: transformação de letras maiúsculas em minúsculas, *stemming*[2] e remoção de *stopwords*[3].

2.2.1.2 Processamento de Consultas

Com o vocabulário e as listas invertidas construídas, o sistema de RI é capaz de receber e responder às consultas dos usuários. Assim, o processamento de consultas se inicia através da necessidade do usuário. Ele submete uma consulta à máquina de busca, contendo uma sequência de termos que representam o seu interesse. Esse processamento é realizado em duas etapas: na primeira, são identificados e recuperados os documentos que contêm os termos da consulta; e na segunda é gerado um *ranking*, onde os documentos retornados são ordenados de acordo com a sua probabilidade de relevância, por meio de uma função de similaridade.

Uma função de similaridade pode levar em conta diversos fatores. Alguns desses fatores dependem da consulta, como por exemplo a frequência de ocorrência dos termos da consulta no documento. Enquanto outros são independentes, como o *PageRank*, que estima a popularidade do documento da coleção [Page et al., 1999]. Baeza-Yates & Ribeiro-Neto [2013] citam que normalmente alguns documentos da coleção podem ter mais termos relacionados à consulta que outros e que determinado termo pode ser mais importante que outro na coleção, e isso faz com que as funções de similaridade utilizem combinações de informações, tais como: (i) frequência do termo no documento (tf), se o termo t ocorrer várias vezes em um documento d, isso implica que o termo t

[2] *Stemming* é o processo de extração do radical das variantes de uma mesma palavra. Desta forma, as variantes passam a ser representadas por um mesmo termo.

[3] *Stopwords* são termos que frequentemente ocorrem em grande parte dos documentos e são comumente constituídos por artigos, preposições e conjunções.

é importante para o documento d; e (ii) frequência do termo na coleção (idf), quanto mais raro um termo for na coleção de documentos, maior é a sua importância.

No decorrer dos anos, foram desenvolvidos vários modelos de RI que as máquinas de busca podem utilizar para tentar predizer a relevância dos documentos para determinada consulta. Dentre os modelos existentes, podem ser citados como os mais importantes: o modelo booleano, proposto por Wartik [1992], o modelo probabilístico, elaborado por Robertson & Jones [1976], o modelo vetorial, produzido por Salton & Lesk [1968] e Salton & Yang [1973], que é considerado um modelo clássico para recuperação de informação. É interessante destacar também o algoritmo BM-25 [Robertson et al., 2000], também da família dos probabilísticos, por ter obtido bons resultados em várias coleções testadas.

A seguir, são descritos os modelos de RI utilizados nesta pesquisa, o modelo vetorial e o modelo probabilístico BM-25.

2.2.1.3 Modelo Vetorial

Considerado um dos modelos clássicos em RI, o modelo vetorial visa recuperar informação de forma simples e eficiente. Nesse modelo é possível obter documentos que respondam parcialmente a uma expressão de busca.

De acordo com Baeza-Yates & Ribeiro-Neto [2013], no modelo vetorial o documento e a consulta são representados por vetores, onde cada coordenada consiste na importância de um termo do vocabulário da coleção para o documento ou consulta em questão. Dessa forma, as representações dos vetores para o documento d_j e a consulta q são, respectivamente, $\vec{d_j} = (w_{1j}, w_{2j}, \ldots, w_{Tj})$ e $\vec{q} = (w_{1q}, w_{2q}, \ldots, w_{Tq})$, onde T é o número de termos no vocabulário da coleção, w_{ij} é o peso do termo t_i no documento d_j, e w_{iq} o peso do termo t_i na consulta q. Os cálculos dos pesos w_{ij} e w_{iq} podem ser definidos da seguinte forma:

$$w_{ij} = tf_{ij} \times idf_i$$
$$w_{iq} = tf_{iq} \times idf_i$$

onde tf_{ij} e tf_{iq} (*term frequency*), representam a frequência com que o termo t_i ocorre no documento d_j e consulta q, respectivamente. O idf_i (*inverse document frequency*) representa a importância do termo t_i para a coleção de documentos e pode ser calculado por:

$$idf_i = \log\left(\frac{N}{n_i}\right)$$

onde N é o número de documentos da coleção e n_i é o número de documentos onde ocorre o termo t_i. O idf estima a raridade do termo t_i na coleção de documentos.

Quanto menor o número de documentos contiverem t_i, maior será seu *idf*, ou seja, t_i é raro na coleção. Entretanto, se t_i ocorrer em todos os documentos da coleção, seu *idf* é igual a zero, ou seja t_i é frequente na coleção.

No modelo vetorial, o cálculo de similaridade entre o documento d_j e a consulta q é feito por meio da correlação entre os vetores $\vec{d_j}$ e $\vec{q}$. Essa correlação pode ser calculada utilizando qualquer medida de relação entre vetores. A medida de relação comumente adotada é a similaridade entre cossenos (*cosine similarity*), isto é, o valor de cosseno do ângulo formado entre eles. Desta forma, a similaridade entre o documento d_j e a consulta q é calculada da seguinte maneira:

$$sim(d_j, q) = \frac{\vec{d_j} \bullet \vec{q}}{|\vec{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{T} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{T} w_{ij}^2} \times \sqrt{\sum_{i=1}^{T} w_{iq}^2}}$$

Segundo Baeza-Yates & Ribeiro-Neto [2013], o modelo vetorial, apesar de ser um modelo simples, obtém bons resultados devido ao esquema de ponderação de termos. Além disso, sua normalização é feita por meio do tamanho dos documentos. Baeza-Yates & Ribeiro-Neto [2013] ainda destacam as principais vantagens do modelo vetorial: (i) o esquema de ponderação de termos melhora a qualidade das respostas; (ii) a estratégia de casamento (*matching*) parcial permite que documentos que são próximos à expressão de busca informada possam ser retornados; e (iii) a fórmula de *ranking* pelo cosseno permite que os documentos sejam ordenados de acordo com seus graus de similaridade com a consulta.

2.2.1.4 Modelo Probabilístico Okapi BM-25

A *Okapi* BM-25 [Robertson et al., 2000] é uma função de ordenação de documentos baseada na teoria da probabilidade, considerada uma das funções mais bem sucedidas na literatura. O índice de similaridade de cada documento, dada uma consulta q, é calculado com base na seguinte fórmula:

$$sim(d_j, q) = \sum_{t_i \in Q} \log\left(\frac{N + 0.5}{n_i + 0.5}\right) \times \frac{(k_1 + 1) \times tf_{ij}}{K + tf_{ij}} \times \frac{(k_3 + 1) \times tf_{iq}}{k_3 + tf_{iq}}$$

onde tf_{ij} é o número de vezes que o termo t_i ocorre no documento d_j, tf_{iq} é o número de vezes que o termo t_i ocorre na consulta q, N é o número de documentos na coleção, n_i é o número de documentos da coleção em que t_i ocorre. O valor de K é definido por: $K = k_1 \times \left((1 - b) + \frac{|d_j|}{d_{avg}}\right)$, onde $|d_j|$ é o tamanho do documento d_j, d_{avg} é o tamanho médio de documentos da coleção e, k_1, k_3 e b são parâmetros que podem ser

ajustados por meio de experimentos. Em linhas gerais, o primeiro termo da função BM-25 corresponde à frequência inversa do documento (idf), enquanto o segundo e o terceiro termo correspondem à frequência do termo no documento (tf_{ij}) e a frequência do termo na consulta (tf_{iq}), respectivamente.

Embora seja possível definir os melhores valores para os parâmetros k_1, k_3 e b experimentalmente, nesta pesquisa são utilizados valores recomendados por Robertson & Walker [1999]: $k_1 = 1.2$, $k_3 = 1000$ e $b = 0.75$. Os autores ainda afirmam que o BM-25 tem sido utilizado com muito sucesso em uma diversidade de tarefas de busca e em diferentes coleções de documentos.

A equação original para o cálculo do idf no BM-25 é $\left(\frac{N-n_i+0.5}{n_i+0.5}\right)$. Entretanto, essa equação obtém pesos de idf negativo sempre que $n_i > N/2$, neste caso termos negativos são introduzidos no cálculo [Baeza-Yates & Ribeiro-Neto, 2013]. Para evitar esse tipo de comportamento, nesta dissertação, foi adotada a estratégia de cálculo do idf: $\left(\frac{N+0.5}{n_i+0.5}\right)$, sugerida por Robertson & Walker [1997] e Robertson [2004] apud Baeza-Yates & Ribeiro-Neto [2013].

2.3 Considerações Finais

Neste capítulo, foram apresentados o conceitos fundamentais para a compreensão desta pesquisa. Foram elucidados os principais conceitos relacionados ao problema do plágio e fundamentos de RI com destaque para os modelos: Vetorial e Probabilístico BM-25. A partir dos conceitos apresentados neste capítulo é possível identificar algumas características presentes nos métodos de detecção de plágio em códigos-fonte. Essas características serão apresentadas detalhadamente no próximo capítulo.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta uma revisão da literatura acadêmica e descreve os principais trabalhos relacionados ao tema detecção de plágio. Tais obras são apresentadas em forma de taxonomia.

3.1 Taxonomia das técnicas de detecção

De acordo com Roy & Cordy [2007], as técnicas de detecção podem pertencer às seguintes classes: baseadas em textos, baseadas em *tokens*, baseadas em árvores, baseadas em grafos, baseadas em métricas e híbridas. As seções a seguir apresentam uma breve descrição de cada uma dessas classes, bem como o estado da arte dos métodos existentes dentro de cada classe.

3.1.1 Técnicas baseadas em textos

Essas técnicas são baseadas puramente em textos, também conhecidas como métodos léxicos, e são consideradas uma análise sintática. Nas técnicas baseadas em textos o código-fonte é considerado como uma sequência de linhas, onde cada linha é uma sequência de caracteres [Roy & Cordy, 2007]. Tais técnicas comparam dois fragmentos de códigos-fonte, a fim de encontrar similaridades entre eles. De acordo com Roy & Cordy [2007], algumas alterações são comumente aplicadas como: remoção de qualquer tipo de comentário do código-fonte, remoção dos espaços em branco e tabulações, e a aplicação de normalizações básicas, como remoções de literais. Na maioria das vezes, o próprio código-fonte é utilizado para comparação.

Baker [1992] propõe o método Dup para detecção de plágio em códigos-fonte. Nesse método o código-fonte é transformado em uma sequência de linhas e a compara-

ção é feita linha-a-linha. Antes de realizar a comparação são removidos do código-fonte: tabulações, espaços em branco e comentários. Identificadores de funções, variáveis e tipos de variáveis são substituídos. O algoritmo de detecção realiza o casamento de palavras individuais das linhas (*string matching*). Entretanto, o desempenho do Dup é baixo, visto que não é possível encontrar as similaridades quando os nomes das variáveis são diferentes, ou quando há mudança no estilo de codificação [Gitchell & Tran, 1999]. Foram utilizadas duas coleções nos experimentos, a primeira com partes de um Sistema Windows (menos algumas inicializações de tabela), e a segunda com parte de um software AT&T.

Outro método de detecção baseado somente em texto, é proposto por Johnson [1993]. Nesse método, assinaturas de substrings das linhas são calculadas e comparadas com objetivo de identificar similaridades. O algoritmo *Karp-Rabin* [Karp & Rabin, 1987] é usado para calcular as assinaturas de todas as substrings. Embora seja possível identificar mudanças estruturais nos códigos-fonte, as transformações utilizadas pelo método e a estratégia de comparação produzem muitos falsos positivos, o que gera resultados não satisfatórios. Nos experimentos foram usados códigos-fonte do GNU C Compiler v. 2.3.3.

Ducasse et al. [1999] propõe o método Duploc, que pouco depende da linguagem de programação. O pré-processamento realizado nesse método consiste na transformação do código-fonte em sequências de linhas e na remoção de espaços em branco. As linhas são comparadas inteiramente e o cálculo de similaridade é feito por meio de um algoritmo quadrático baseado em *strings* denominado *Dynamic Pattern Matching* (DPM). Além disso, a detecção só é possível se os códigos forem idênticos, visto que qualquer mudança no nome da variável, inserção de códigos ou códigos reestruturados faz com que o método perca sua eficácia. As coleções utilizadas nos experimentos são: arquivos de implementação da fonte GNU gcc (escrito em C), um quadro de mensagens baseado na web (Python), partes de um aplicativo de folha de pagamento (Cobol) e um servidor de banco de dados (Smalltalk).

Marcus & Maletic [2001] propõem um método que aplica uma técnica de recuperação de informação chamada indexação semântica latente [Dumais, 1994] para detecção de plágio em códigos-fonte. Nesse método, o código não é comparado por inteiro. O objetivo é encontrar correspondências entre os fragmentos de identificadores e comentários. Entretanto, a principal desvantagem dessa técnica é que funções que possuem as mesmas estruturas, que realizam as mesmas operações, mas sem comentários e com identificadores completamente diferentes, não são detectadas como plágio.

Para os experimentos foi utilizada a coleção Mosaic[1] Source Code v2.7b5.

3.1.2 Técnicas baseadas em *tokens*

Nas técnicas baseadas em *tokens*, o código-fonte é transformado em uma série de *tokens* que podem ser palavras, frases, símbolos, operadores matemáticos, etc. A detecção de plágio é feita por meio das comparações entre as sequências de *tokens* obtidas a fim de encontrar a maior subsequência comum. Segundo Roy & Cordy [2007], essa técnica normalmente é mais robusta do que as técnicas baseadas em texto. De acordo com Cornic [2008], a principal vantagem dessa técnica é a sua generalidade, pois através dela é possível descartar informações desnecessárias como nomes de variáveis ou métodos. Sendo assim, nessa técnica, alterações como "procurar e substituir" não são significativas.

Prechelt et al. [2002] desenvolveram um método denominado JPlag [2] que detecta plágios por meio de uma versão modificada do algoritmo *Running Karp-Rabin Greedy String Tiling* (RKR-GST) [Wise, 1993]. Esse método funciona da seguinte forma: primeiro, os códigos-fonte são submetidos a um analisador que interpreta as estruturas da linguagem e gera os *tokens*. Em seguida, esses *tokens* são comparados par a par. Na sequência, é calculado um índice de similaridade por par. O JPlag atualmente oferece suporte a várias linguagens de programação, tais como: C, C++, C# e Java. Além disso, também é possível comparar arquivos de texto em linguagem natural. Foram usados quatro tipos diferentes de programas nos experimentos (chamados i12, i27, i51 e j5). Três foram exercícios de programação de um curso de informática de segundo semestre, e um era de um curso de programação avançada de pós-graduação. Uma implementação desse método é usada como um *baseline* para o método proposto nesta dissertação.

Mariani & Micucci [2012] propõem um método para detecção de plágio em ambientes acadêmicos, chamado AuDeNTES. Esse método requer duas informações: a coleção de códigos-fonte e uma solução de referência para a questão, fornecida pelo professor. Para Mariani & Micucci [2012], a solução de referência é uma solução completa para a questão proposta aos alunos. O método realiza a detecção de plágio em códigos-fonte Java por meio de três fases: conversão, filtragem e cálculo de similaridade. Na fase de conversão, o código é transformado em sequências de *tokens*, por

[1] ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/

[2] Uma implementação para o método pode ser encontrada em http://jplag.ipd.kit.edu/.

meio das ferramentas JavaCC[3] [Java.net, 2014] e JTB[4] [Palsberg, 2004]. Na fase de filtragem, os *tokens* iguais à solução de referência são removidos por meio do algoritmo *Running Karp-Rabin Greedy String Tiling* (RKR-GST) [Wise, 1993]. Por fim, na fase do cálculo de similaridade é calculado um valor de similaridade entre os *tokens* com uma variação do RKR-GST. AuDeNTES foi avaliado usando cinco conjuntos de dados com submissões reais de alunos de um curso de programação. Esse trabalho é usado, com modificações (cf. Seção 5.2), como um *baseline* para o método proposto nesta dissertação.

Yuan & Guo [2012] apresentam um método baseado em *tokens* que incorpora um conceito denominado matriz de características. Essa matriz é construída com base nas características das variáveis do código-fonte. Os autores empregam uma série de evidências sobre a forma como as variáveis são usadas ao longo do código-fonte, tais como: se uma variável é usada dentro de um *if* ou de um *loop*; ou se a variável é usada em uma adição ou subtração. O cálculo de similaridade entre os pares é feito por meio do modelo vetorial, comumente adotado em sistemas de recuperação de informação. Para a avaliação da eficácia do método foram usados códigos-fonte do Java SE Development Kit 7 (7.492 arquivos, 2,260,946 LoC) e do kernel Linux 2.6.38.6 (35.856 arquivos, 10.068.963 LoC).

3.1.3 Técnicas baseadas em árvores

Técnicas baseadas em árvores, também chamadas de *Abstract Syntax Trees* (ASTs), são representações hierárquicas da estrutura sintática do código-fonte, formada pelas representações abstratas de cada elemento [Roy & Cordy, 2007]. Assim, uma AST é uma árvore de *tokens* que segue um conjunto de regras específicas de determinada linguagem de programação.

A detecção de plágio em uma estratégia AST, consiste em encontrar as subárvores comuns aos pares de códigos-fonte por meio de algoritmos de casamento (*matching*) em árvores. A Figura 3.1 apresenta um exemplo de representação em AST, onde à esquerda é exibido um exemplo de código-fonte em linguagem C e à direita é apresentada a estrutura da árvore AST desse código.

De acordo com Cornic [2008], para que métodos baseados nessa estratégia se tornem mais robustos na detecção, basta eliminar nomes de variáveis e constantes.

[3] *Java Compiler Compiler* (JavaCC) é uma ferramenta geradora de parser utilizada para ler uma gramática e converter em um programa Java que faz o reconhecimento desta gramática.

[4] *Java Tree Builder* (JTB) é uma ferramenta que usa a gramática JavaCC e gera um conjunto de classes de árvores de sintaxe, referente à gramática, dois visitantes e uma gramática JavaCC com anotações.

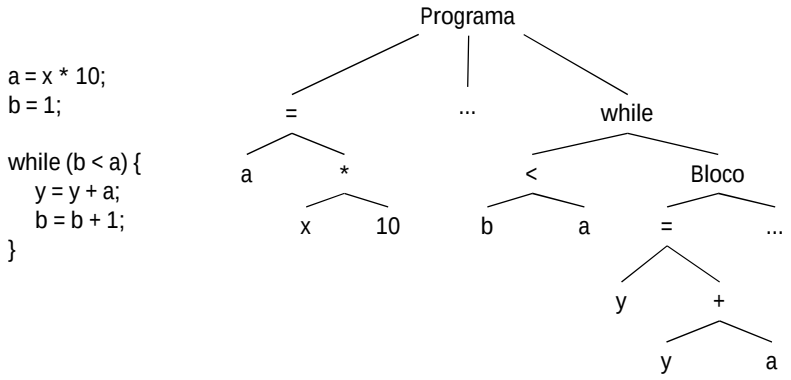

Figura 3.1: A esquerda o código-fonte em linguagem C e a direita sua respectiva representação em árvore (AST).

Contudo, Cornic [2008] afirma que ela é vulnerável à inserção de código ou códigos reestruturados. Para procurar uma subárvore comum entre duas árvores é necessária a comparação entre as subárvores, o que a torna complexa é inviável para grandes conjuntos de dados.

O método CloneDR, desenvolvido por Baxter et al. [1998], é um dos pioneiros em detecção de plágio que usa uma estratégia baseada em AST. Nesse método os códigos são transformados em uma representação AST e as subárvores são comparadas através de métricas baseadas em funções *hash*. Os fragmentos de subárvores similares são considerados como plágio. De acordo com Baxter et al. [1998], o CloneDR suporta reordenação de instruções. Foi usado nos experimentos, códigos-fonte de sistema de controle de processos com aproximadamente 400KSLOC de código-fonte em C.

Zhang & Liu [2013] apresentam um método baseado em AST para detectar plágio em códigos-fonte em cursos de programação. Essa detecção é feita da seguinte forma: primeiro, os códigos são transformados em uma representação AST por meio da ferramenta ANTLR[5] [Parr, 2007] e o algoritmo Smith Waterman [Smith & Waterman, 1981] é usado para calcular a similaridade entre os códigos. Em seguida, os fragmentos de árvores similares são identificados, extraídos e são obtidos os vetores com as características. Por fim, esses vetores são agrupados e o algoritmo *k-medoids* [Jing-Bin et al., 2011] é usado para detectar a ocorrência de plágios. Os experimentos foram feitos com 12 grupos de programas em linguagem C.

Tao et al. [2013] propõem outro método baseado em AST, denominado *CodeCompare*, para detecção de plágio em códigos na Linguagem C/C++. Antes de realizar a detecção é necessário executar um pré-processamento nos códigos, onde os espaços, as linhas em branco e referências a arquivos externos são removidos e as instruções

[5] *ANother Tool for Language Recognition* (ANTLR) é um gerador de analisador sintático que com base em uma gramática especifica gera um analisador que pode construir e andar em arvores sintáticas.

define e *typedef* são substituídas. Para realizar a detecção, esse método transforma o código em uma representação AST por meio das ferramentas Lex[6] [Levine et al., 1992] e Yacc[7] [Levine et al., 1992]. Em seguida, é calculado um valor *hash* para cada nó da árvore. Por fim, o cálculo de similaridade é feito com base nesses valores. Nós da árvore que possuem valores *hash* iguais e possuem a mesma quantidade de subnós são detectados como plágio. De acordo com os autores, o método é robusto a alterações nas estruturas condicionais. Para os experimentos foram usados códigos-fonte de snort-1.7, FileZilla_3.2.8.1 e Azureus/Vuze_4.2.0.8. Esse trabalho é usado como um *baseline* para o método proposto nesta dissertação.

3.1.4 Técnicas baseadas em grafos

O *Program Dependence Graph* (PDG), também chamado de técnica baseada em grafo, é uma representação em grafo do código-fonte onde as declarações são representadas por vértices, e as dependências de dados e de controle entre as declarações, através de arestas. De acordo com Roy & Cordy [2007], PDG descreve a estrutura significativa dos programas, pois contém as informações de fluxo de controle e fluxo de dados e, portanto, carrega a informação semântica do código-fonte analisado.

Diferente das outras representações utilizadas na detecção de plágio, o PDG não armazena informação sintática do código-fonte e é considerado uma medida de similaridade semântica. Comumente, os plagiadores realizam modificações como renomeação de variáveis e reordenação de declarações. Essas variações não alteram ou modificam um PDG. A Figura 3.2 apresenta um exemplo de representação PDG, sendo que à esquerda é exibido um exemplo de código-fonte em linguagem C e à direita é apresentada a estrutura PDG desse código.

Liu et al. [2006] propõem um método de detecção de plágio baseado em grafos denominado GPlag. Nesse método, os códigos-fonte são transformados em uma representação PDG e, em seguida, os subgrafos isomorfos entre os pares de códigos são detectados como plágios. De acordo com Cornic [2008], os desenvolvedores asseguram que ele é capaz de detectar os mais complexos tipos de plágios, entretanto, a documentação e os testes que comprovem sua eficácia são escassos. Embora o GPlag se mostre robusto diante das técnicas baseadas em *tokens*, visto que alterações de nomes de variáveis ou reordenação de funções não modificam um PDG, Ohmann [2013] relata que o método tende a esperar que o código-fonte contenha chamadas a funções

[6] É uma ferramenta para geração de analisadores léxicos na linguagem C. O Lex divide o código-fonte em *tokens* de acordo com a gramática especificada.

[7] É uma ferramenta para geração de analisadores sintáticos em código C. Através dos *tokens* obtidos com o Lex, o Yacc forma as regras de produção e a respectiva estrutura hierárquica do código-fonte.

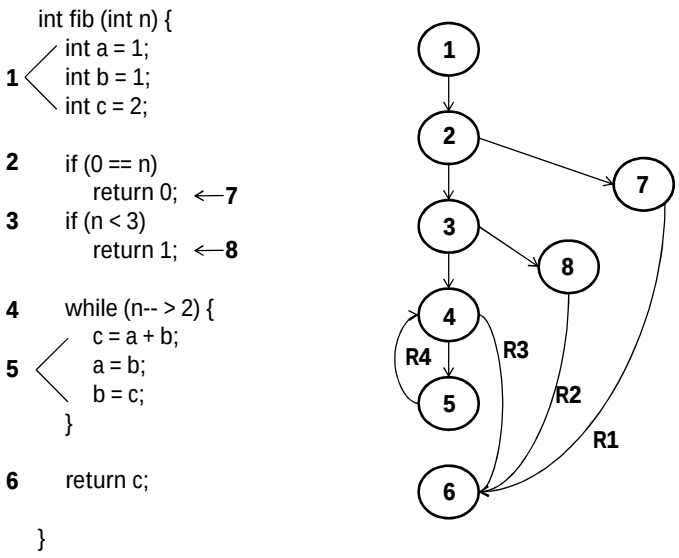

Figura 3.2: A esquerda o código-fonte em linguagem C e a direita sua respectiva representação em grafo (PDG).

ou procedimentos, e essas funcionalidades não são comuns em cursos introdutórios de programação. Para os experimentos, utilizou códigos-fonte de quatro programas Linux: join (programa que une textos), bc (calculadora), less (visualizador de texto) e tar (programa de arquivos).

Li & Ernst [2012] apresentam o método *Cloned Buggy Code Detector* (CBCD), que tem como propósito de encontrar plágios em códigos-fonte com erros. O CBCD executa a detecção por meio de três fases: na primeira fase, os códigos e os erros são transformados em PDGs. Na segunda fase, os PDGs obtidos na fase anterior são transformados em vários subgrafos para reduzir o número de comparações a serem feitas. Na terceira e última fase, esses subgrafos são comparados a fim de verificar se o PDG de erro é um subgrafo PDG do código-fonte. As coleções usadas nos experimentos foram o SCM do kernel Linux, Git e PostgreSQL, e o sistema de relatório de bugs de uma linha de produtos de software comercial.

3.1.5 Técnicas baseadas em métricas

Técnicas baseadas em métricas empregam métricas sobre o código-fonte, tais como: a frequência de *keywords*, número de chamadas a funções, quantidade de *loops* usadas, etc, que são utilizadas nos algoritmos de comparação. Segundo Roy & Cordy [2007], na maioria dos casos, o código-fonte é transformado em uma representação AST ou PDG para o cálculo dessas métricas. Desta forma, o alvo da detecção são as métricas calculadas e não o código-fonte em si, como acontece nas técnicas baseadas em textos,

tokens, árvores e grafos.

Mayrand et al. [1996] propõem um método baseado em métricas para detectar plágio em códigos-fonte. Essas métricas incluem número de linhas, número de chamadas de funções, número de arestas ou relacionamentos, número de declarações, número de variáveis locais e não locais, entre outros. As métricas são extraídas dos códigos-fonte para se obter um grau de similaridade por meio da comparação dos valores obtidos. Fragmentos de código com valores de métricas similares são detectados como plágio. Para os experimentos foram criados dois projetos com códigos-fonte em linguagem C++.

Outro método baseado em métricas, denominado *Student Submissions Integrity Diagnosis* (SSID), é proposto por Poon et al. [2012]. Esse método realiza a detecção da seguinte forma: primeiro, os códigos-fonte são transformados em *tokens* N-gramas. Em seguida, esses *tokens* são comparados por meio de uma versão modificada do algoritmo *Greedy-String-Tiling* [Wise, 1993]. Por fim, os *tokens* similares são agrupados por meio do DBScan [Ester et al., 1996], e para fins de classificação dos códigos entre plágios e não plágios, utiliza-se um limiar de similaridade. Para os experimentos, 28 estudantes participaram da confecção da coleção de códigos, gerando uma versão plagiada de códigos de exemplo.

3.1.6 Técnicas híbridas

Técnicas híbridas combinam diferentes representações do código-fonte ou diferentes algoritmos de comparação. Vários métodos usam esse tipo de técnica, entretanto, tais métodos também podem ser classificados nas categorias anteriores.

Koschke et al. [2006] propõem um método que usa árvores de sufixo[8] para detecção de plágio. Nesse método, o código-fonte é transformado em uma representação AST e são criadas árvores de sufixo a partir dos nós dessa AST. As sequências de nós AST mais longas são retiradas da árvore de acordo com a região sintática, e permanecem na árvore apenas os nós sintaticamente próximos. Para comparar esses nós é usado um algoritmo baseado em árvores de sufixo. De acordo com os autores, os experimentos realizados indicam que o método apresenta bons resultados e ainda oferece maior escalabilidade que técnicas baseadas em *tokens*. Nos experimentos foram usados códigos-fonte de quatro programas: bison 1.32, wget 1.5.3, SNNS 4.2 e postgreSQL 7.2.

Basit et al. [2007] apresentam o método *Repeated Tokens Finder* (RTF) para detecção de plágio em códigos-fonte. Esse método se destaca por uma detecção de plágio

[8] É uma estrutura de dados que forma um índice de todos os sufixos de uma palavra, através desse índice é possível obter de forma rápida informações sobre sua estrutura.

em menor tempo computacional e por uma economia de memória. Seu processo de detecção é feito da seguinte maneira: em primeiro lugar, os códigos-fonte são transformados em *tokens* AST. Em seguida, o RTF usa um vetor de sufixos[9] baseado em um algoritmo de complexidade linear ao invés de árvore de sufixos, onde é possível remover os *tokens* desnecessários de modo a reduzir a falsa detecção. A avaliação da eficácia do método foi feita usando códigos-fonte do kernel do Linux 2.6.14.

Jiang et al. [2007] propõem o método DECKARD, que detecta plágio em códigos-fonte independentemente da linguagem de programação. DECKARD utiliza vetores de características que armazenam informações estruturais aproximadas de subárvores AST. Os vetores similares são agrupados com a utilização de funções *Locality Sensitive Hashing* (LHS). As subárvores que estiverem no mesmo agrupamento serão consideradas similares, isto é, são possíveis plágios. Para os experimentos foram usados códigos-fonte de dois programas: JDK 1.4.2 (8,534 arquivos java, 2,418,767 LoC) e kernel Linux 2.6.16 (7,988 arquivos C, 5,287,090 LoC).

3.1.7 Análise Comparativa das Técnicas de Detecção

Como apresentado nas seções anteriores, a literatura propõe diferentes métodos para detecção de plágio em códigos-fonte. Através desse estudo é possível perceber que as normalizações realizadas nos códigos estão diretamente relacionadas à forma de representação do código. A maioria dos métodos apresentados remove comentários, espaços e linhas em branco.

Na Tabela 3.1 é apresentado um resumo dos métodos estudados nesta dissertação. Nessa tabela, a primeira coluna representa o nome dos autores, e a segunda coluna refere-se à representação usada nos métodos. A terceira diz respeito ao tipo de pré-processamento realizado. Por fim, na quarta coluna é apresentada a estratégia de comparação usada pelos métodos estudados. O método proposto é o último listado.

Dentre os métodos encontrados na literatura, apenas o de Mariani & Micucci [2012] compara com outro código que resolve o mesmo problema, a fim de estabelecer um padrão pessoal do indivíduo. O método proposto além de tentar estabelecer o padrão do indivíduo, como contribuição, tenta estabelecer também o do professor e o da questão, conforme será explicado com mais detalhes no Capítulo 4. Com isso, novos métodos de dectecção ainda mais eficientes podem surgir a partir dos resultados obtidos.

[9] É uma estrutura de dados semelhante à árvore de sufixos. Entretanto, é mais eficiente em termos de uso de memória, além de ter menor complexidade para codificação.

O método proposto será comparado aos métodos de Prechelt et al. [2002], Mariani & Micucci [2012] e Tao et al. [2013], abordagens destacadas na literatura. A técnica de detecção adotada para nortear esta pesquisa é a baseada em *tokens*, visto que nesta técnica é possível manter espaços e linhas em branco, que são descartadas na fase de pré-processamento das técnicas baseadas em árvores e grafos.

3.2 Considerações Finais

Neste capítulo foram apresentadas as principais técnicas de detecção de plágio existentes na literatura e alguns trabalhos relacionados ao tema desta pesquisa. A técnica para o desenvolvimento do método proposto e os trabalhos que servirão de base de comparação foram definidos. No próximo capítulo é apresentado o método proposto por esta dissertação para o problema de detecção de plágio.

Tabela 3.1: Compilação dos métodos descritos quanto ao tipo de formas de representação, pré-processamento e estratégia de comparação.

Autor	Forma de Representação	Pré-processamento	Estratégia de Comparação
Baker [1992]	Sequência de linhas	Remoção de comentários e espaços em brancos	*String Matching*
Johnson [1993]	Assinatura de substrings	Remoção de espaços em brancos	*Karp-Rabin*
Ducasse et al. [1999]	Sequência de linhas	Remoção de espaços em brancos	*Dynamic Pattern Matching* (DPM)
Marcus & Maletic [2001]	Texto	Remoção de espaços em brancos	*Latent Semantic Indexing*
Prechelt et al. [2002]	*Tokens*	Remoção de comentários e espaços em brancos	*Running Karp-Rabin Greedy String Tiling* (RKR-GST)
Mariani & Micucci [2012]	*Tokens*	Remoção de comentários, espaços em brancos e partes comuns com o RKR-GST	*Running Karp-Rabin Greedy String Tiling* (RKR-GST)
Yuan & Guo [2012]	*Tokens*	Remoção de comentários e espaços em brancos	Modelo Vetorial
Baxter et al. [1998]	AST	Transformações para AST	*Tree Matching*
Zhang & Liu [2013]	AST (ANTLR)	Transformações para AST	*Smith Waterman*
Tao et al. [2013]	AST (LEX e YACC)	Remoção de linhas e espaços em branco	*Matching* de valores *hash*
Liu et al. [2006]	PDG	CodeSurfer para PDG	*Matching* de subgrafos isomorfos
Li & Ernst [2012]	PDG	Transformações para PDG	*Matching* de subgrafos
Mayrand et al. [1996]	Métricas obtidas com Datrix	Transformações diversas	*Matching* de métricas
Poon et al. [2012]	*Tokens*	Remoção de comentários e espaços em brancos	*Greedy String Tiling*
Koschke et al. [2006]	AST	Transformações para AST	Árvores de sufixo
Basit et al. [2007]	*Tokens*	transformações diversas	Vetores de sufixo
Jiang et al. [2007]	Vetores de características	extração de métrica	*Locality Sensitive Hashing* (LSH)
Método Proposto	*Tokens*	Remoção de comentários e literais	*Okapi* BM-25

Capítulo 4

Detecção de Plágio baseada em Estilos de Programação

Este capítulo apresenta a solução proposta para problema de detecção de plágio em códigos-fonte. São descritas as etapas do método, bem como suas particularidades.

4.1 Estilos de Programação

O método proposto assume que cada fragmento de código-fonte possui um *estilo de programação*, que representa uma forma particular de codificar instruções em determinada linguagem. Para exemplificar, na linguagem C os programadores podem desenvolver seus códigos com diferentes estruturas de repetição. Isso pode ser observado nos Códigos 4.1 e 4.2, onde o programador do primeiro código optou por usar a estrutura *for*, e o programador do segundo código optou por usar a estrutura *while*.

Código 4.1: Exemplo de código em C com a estrutura de repetição *for*.

```
#include <stdio.h>

int main(void) {

    int var;
    for (var=0; var<5; var++) {
        printf("%d\n", var);
    }
    return 0;

}
```

Código 4.2: Exemplo de código em C com a estrutura de repetição *while*.

```
#include <stdio.h>

void main()
{
    int var=0;
    while (var<5)
    {
        printf("%d\n", var);
        var+=1;
    }
}
```

As duas estruturas estão corretas segundo a sintaxe da linguagem C e a escolha

de uma delas caracteriza um *estilo de programação*. As linguagens de programação possibilitam uma série de outras opções de estilo, com outros tipos de estruturas e instruções. Esta dissertação considera a existência de três classes de *estilos*:

1) *Estilo do aluno*: durante o processo de aprendizagem, o aluno desenvolve um estilo próprio de programação, que tende a estar presente em todos os seus códigos. A exemplo disso, o aluno pode aprender uma maneira de usar uma estrutura condicional e usa apenas essa estrutura em todos os códigos desenvolvidos;

2) *Estilo da questão*: é considerado o estilo específico de cada questão (ou problema a ser solucionado) e, portanto tende a estar presente em todos os códigos desenvolvidos para solucionar essa questão. Um exemplo desse tipo de estilo é o caso de uma questão requerer o uso de dois laços de repetição para ser solucionada;

3) *Estilo do professor*: é considerado o estilo que tende a estar presente em todos os códigos desenvolvidos por alunos de determinado professor. O método assume que os professores tendem a influenciar o estilo de alunos iniciantes em programação. Por exemplo, alguns professores preferem o uso do *#define* para definir valores constantes de uma aplicação, ao invés de usar variáveis para esse mesmo fim.

Para ilustrar a existência dessas classes de estilos, considere a Tabela 4.1, que apresenta uma hipotética lista de exercícios com M questões solucionadas por uma turma com P alunos. Nessa tabela, cada campo representa uma tentativa de solução de determinada questão por determinado aluno. A coluna Aluno 01, representa os códigos desenvolvidos pelo primeiro aluno para cada questão da lista. Uma das hipóteses do método proposto é que cada código-fonte desta coluna tenderá a conter fragmentos com o estilo do Aluno 01 e a não conter fragmentos com os estilos dos demais alunos. O estilo de programação de um aluno tende a não estar presente em códigos plagiados por esse aluno, e por isso esse estilo pode ser uma importante evidência para as tarefas de detecção de plágio.

Tabela 4.1: Lista de exercícios com M questões oferecida para uma turma de P alunos. Nessa tabela, $S_{x,y}$ corresponde a uma solução de um aluno x para uma questão y.

Questão	Aluno 01	Aluno 02	Aluno 03	...	Aluno P
01	$S_{01,01}$	$S_{02,01}$	$S_{03,01}$	...	$S_{P,01}$
02	$S_{01,02}$	$S_{02,02}$	$S_{03,02}$	...	$S_{P,02}$
03	$S_{01,03}$	$S_{02,03}$	$S_{03,03}$	...	$S_{P,03}$
⋮	⋮	⋮	⋮	⋮	⋮
M	$S_{01,M}$	$S_{02,M}$	$S_{03,M}$	...	$S_{P,M}$

Na mesma linha de raciocínio, cada linha da Tabela 4.1 representa as tentativas de soluções dos P alunos para uma determinada questão. Por exemplo, a primeira linha representa as tentativas de soluções de todos os alunos para a Questão 01. É necessário observar que os códigos desenvolvidos por diferentes alunos para uma mesma questão podem possuir fragmentos de códigos similares, mesmo que tais códigos não tenham sido plagiados.

O método proposto assume que o professor tende a influenciar o estilo de programação dos alunos iniciantes. Em virtude disso, nesta dissertação assume-se que o estilo comum a toda lista de exercícios, isto é, a todos os campos da Tabela 4.1, são definidos como sendo do professor. A hipótese é que essa influência tende a ser mais forte nas turmas mais básicas de programação, portanto, esse tipo de estilo também representa uma importante evidência explorada pelo método proposto.

4.2 Método Proposto

Seja uma lista de exercícios com M questões de programação solucionadas por P alunos, tal como no exemplo da seção anterior. O método proposto executa, na coleção de $M \times P$ códigos desenvolvidos para essa lista, o seguinte conjunto de passos:

1) *Pré-processamento*: onde os fragmentos não úteis ao processo de detecção de plágio são removidos dos códigos da coleção;

2) *Tokenização*: passo responsável por transformar cada um dos $M \times N$ códigos em sequências de *tokens*, por meio do algoritmo *janela deslizante* descrito na Seção 4.2.2;

3) *Indexação*: passo responsável por construir o *vocabulário* e as *listas invertidas* a partir de todos os códigos-fonte da coleção;

4) *Cálculo de similaridade*: onde os *tokens* de determinado código-fonte são submetidos como consulta ao índice por meio de um modelo de recuperação de informação.

A Figura 4.1 apresenta uma visão geral do método proposto. A seguir, são descritos detalhadamente os quatro passos do método proposto nesta dissertação de mestrado.

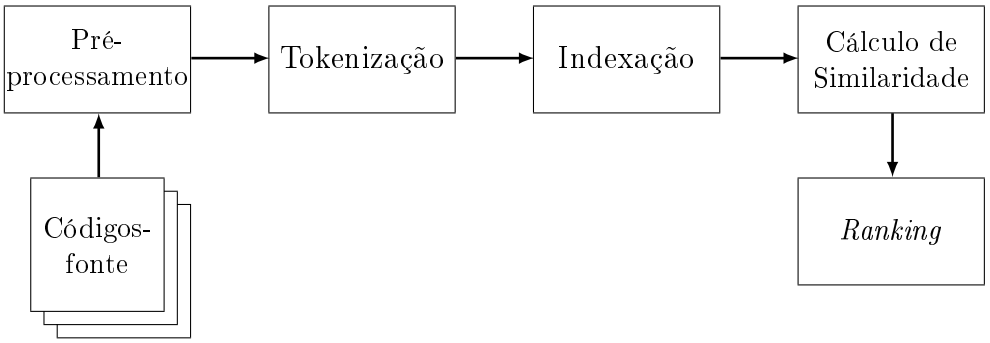

Figura 4.1: Visão geral do método proposto.

4.2.1 Pré-processamento

De maneira análoga aos trabalhos de Burrows & Tahaghoghi [2007] e Palheta [2013], todos os comentários, referências à arquivos externos (bibliotecas) e caracteres entre aspas (“ ” e ’ ’) são removidos dos códigos-fonte, para fins de anonimização. Tais fragmentos são removidos para que o método possa focar somente na codificação lógica. De acordo com Burrows et al. [2007], essas estruturas são boas para o reconhecimento humano, em métodos automatizados seu uso não é bom. O Código 4.3 destaca em tom de cinza os comentários, referências às bibliotecas e caracteres removidos neste passo.

Código 4.3: Comentários, referências à bibliotecas e caracteres removidos no passo de pré-processamento.

```
1 #include <stdio.h>
2 int main(void) { /*Simples programa em C*/
3     int var;
4     for (var=0; var<5; var++) {
5         printf("%d\n", var);
6     }
7     return 0;
8 }
```

Fonte: Adaptado de Burrows et al. [2007].

4.2.2 Tokenização

Após o pré-processamento, os códigos-fonte da coleção são transformados em sequências de *tokens*. O processo de tokenização consiste em dividir o código-fonte em cada fragmento que o compõe. De acordo com Burrows et al. [2007], a tokenização é usada para atribuir a cada fragmento do código-fonte um símbolo que o represente. Nesse processo, devem ser mantidas as estruturas mais significativas como tipos de dados utilizados (por exemplo, *int* e *float*), instruções de seleção (por exemplo, *if* e *switch case*), e instruções de iteração (por exemplo, *for* e *while*) [Burrows et al., 2007]. Observe

que os tokens gerados neste passo mantêm os estilos de aluno, de questão e de professor dos códigos-fonte originais. Essa é a principal razão da escolha pelo desenvolvimento de uma técnica de detecção de plágio baseada em *tokens*, uma vez que nos demais tipos de técnicas (baseadas em árvore, grafos, etc) os espaços, linhas em branco e outras evidências de estilo são removidas na etapa de pré-processamento.

A lista completa dos operadores, palavras reservadas e caracteres de espaço considerados durante o processo de tokenização podem ser vistos no Anexo A. A Tabela A.1 apresenta os operadores, as Tabelas A.2 e A.3 apresentam as palavras-chave e, a Tabela A.4 exibe os caracteres de espaços em branco. Para ilustrar este processo, o Código 4.4 apresenta as palavras reservadas destacadas em azul e operadores destacados em amarelo mantidos neste passo.

Código 4.4: Palavras reservadas e operadores da Linguagem C considerados durante o processo de tokenização.

```
1 int main ( void ) {
2     int var;
3     for ( var = 0; var < 5; var ++) {
4         printf ( "" , var ) ;
5     }
6     return 0;
7 }
```

Fonte: Adaptado de Burrows et al. [2007].

O Código 4.5 destaca as sequências de espaços em branco mantidos durante o processo de tokenização. O caractere "•" representa um espaço em branco, o "⟶" simboliza uma tabulação de tamanho 4 e o "↩" corresponde a uma quebra de linha.

Código 4.5: Caracteres de espaços e quebras de linha considerados durante o processo de tokenização.

```
1 int•main(void)•{↩
2 ⟶int•var;↩
3 ⟶for•(var=0;•var<5;•var++)•{↩
4 ⟶⟶printf("",•var);↩
5 ⟶}↩
6 ⟶return•0;↩
7 }
```

Fonte: Adaptado de Burrows et al. [2007].

A Tabela 4.2 apresenta os fragmentos do Código 4.4 e do Código 4.5 mantidos neste passo e seus respectivos símbolos, de acordo com os símbolos do Anexo A. Após essa conversão, os símbolos dessa tabela são concatenados para gerar a seguinte cadeia

Tabela 4.2: Fragmentos de códigos-fonte e seus respectivos símbolos.

No.	Fragmento	Símbolo	No.	Fragmento	Símbolo
1	int	tb	20	++	ga
2	espaço	de	21	)	ba
3	(	aa	22	espaço	de
4	void	sb	23	quebra	ge
5	)	ba	24	tab	fe
6	espaço	de	25	tab	fe
7	quebra	ge	26	printf	ed
8	tab	fe	27	(	aa
9	int	tb	28	vírgula	ob
10	espaço	de	29	espaço	de
11	quebra	ge	30	)	ba
12	tab	fe	31	quebra	ge
13	for	uc	32	tab	fe
14	espaço	de	33	quebra	ge
15	(	aa	34	tab	fe
16	=	db	35	return	tc
17	espaço	de	36	espaço	de
18	<	ra	37	quebra	ge
19	espaço	de			

Fonte: Adaptado de Burrows et al. [2007].

de caracteres (S): tbdeaasbbadegefetbdegefeucdeaadbderadegabadegefefeedaaobdebagefegefetcdege.

Para a criação dos N-gramas é utilizada a técnica de *janela deslizante*, que gera N-gramas movendo uma "janela" de tamanho N em todas as partes da *string* da esquerda para a direita. Diferentes trabalhos utilizaram diferentes tamanhos de N para gerar os N-gramas das coleções. Por exemplo, o trabalho de Burrows et al. [2007] usa o valor N=4 e, o trabalho de Palheta [2013] usa o valor N=6. O valor de N varia de coleção para coleção e se faz necessário a realização de experimentos para ajudar a definir o tamanho em cada caso. Para essa dissertação, foram realizados experimentos com valores de $N = \{4,5,6\}$, tomando-se por base os valores de N mencionados pela literatura. Com base nesse conjunto de experimentos, foi adotado o valor de N = 5, pois foi o que obteve um resultado satisfatório para a coleção de códigos utilizada.

A Tabela 4.3 ilustra o processo de geração de 5-gramas da string S por meio do algoritmo de *janela deslizante*. Nessa tabela, as sequências geradas representam os termos do vocabulário de um sistema de RI.

Como resultado, obtêm-se os seguintes termos: tbaasbbatb, aasbbatbuc, sbbatbucaa, batbucaadb, tbucaadbra, ucaadbraga, aadbragaba, e etc. Por meio desses termos é possível construir o índice do sistema de RI.

Tabela 4.3: Sequências de 5-gramas obtidas por meio do algoritmo *janela deslizante*.

	tb	aa	sb	ba	tb	uc	aa	db	ra	ga	ba	ed	aa	ba	tc
								⇓							
1	tb	aa	sb	ba	tb										
2		aa	sb	ba	tb	uc									
3			sb	ba	tb	uc	aa								
4				ba	tb	uc	aa	db							
5					tb	uc	aa	db	ra						
6						uc	aa	db	ra	ga					
7							aa	db	ra	ga	ba				
8								db	ra	ga	ba	ed			
9									ra	ga	ba	ed	aa		
10										ga	ba	ed	aa	ba	
11											ba	ed	aa	ba	tc

4.2.3 Indexação

O índice gerado neste passo é diferente do índice tradicional (cf. Seção 2.2.1.1), visto que o método proposto neste projeto requer tipos de dados adicionais. Cada item da lista invertida é representado por uma 4-tupla (tf, a, q, p), onde tf é a frequência do termo, a é o identificador do aluno, q é o identificador da questão, e p é o identificador do professor. A Tabela 4.4 apresenta um exemplo de um índice gerado a partir dos termos mencionados anteriormente. Nessa tabela, o termo "tbdeaasbba", ocorre 3 vezes no código-fonte do aluno 1 e ocorre 5 vezes no código-fonte do aluno 2, ambos submetidos como resposta para a questão 2 da lista de exercícios do professor 1.

Tabela 4.4: Exemplo de um índice para um vocabulário de três termos.

Vocabulário	Listas Invertidas
tbaasbbatb	(3,1,2,1) (5,2,2,1)
aasbbatbuc	(1,1,1,1) (2,2,2,1) (3,2,3,1) (4,3,2,1)
sbbatbucaa	(2,3,3,1) (3,4,3,1) (7,3,2,1)
...	.....

4.2.4 Cálculo de Similaridade

Com o *vocabulário* e as *listas invertidas* construídos é possível utilizar técnicas da área de RI a fim de verificar o quão similar é cada par de códigos-fonte desenvolvido por alunos de um professor. Para o cálculo do grau de similaridade, serão avaliadas duas funções de *ranking*, o modelo Vetorial e o modelo Probabilístico BM-25, com o objetivo de escolher que função apresenta melhor resultado.

O método parte da hipótese de que o processo de detecção de plágio pode ser melhorado com o uso das três classes de estilo mencionadas anteriormente: estilos de

alunos, estilos de professor e estilos de questão. Para testar essa hipótese, o modelo Vetorial e o modelo Probabilístico BM-25, apresentados no Capítulo 2, foram estendidos a fim de incorporar as informações descritas e verificar seu impacto no processo de detecção de plágio. A seguir, são descritas as adaptações feitas nos modelos tradicionais.

4.2.4.1 Adaptação do Modelo Vetorial e do Modelo Probabilístico BM-25

Nos modelos Vetorial e Probabilístico BM-25, a importância de um termo no documento é dada pela frequência do termo no documento (*term frequency, tf*) e por sua raridade na coleção (*inverse document frequency, idf*). Em especial, a métrica *idf* proposta em 1972 por Karen Sparck Jones explora o conceito de especificidade/raridade de termos [Sparck Jones, 1972]. O *idf* de um termo é uma medida da raridade desse termo em determinada coleção, e sua fórmula é descrita na Seção 2.2.1.3. Para entender a ideia dessa métrica, é necessário considerar uma coleção de códigos-fonte tokenizados através da técnica descrita anteriormente. Pode-se notar que, se um token t ocorre em muitos documentos da coleção, então a ocorrência de t em dois códigos-fonte distintos não constitui uma evidência forte de plágio. Por outro lado, se t é raramente usado, então a ocorrência de t em dois códigos-fontes distintos constitui uma evidência de plágio mais forte.

Em sua fórmula original, o *idf* explora o conceito de raridade em coleções completas de documentos ou códigos-fonte. No entanto, as classes de estilo possibilitam a exploração desse conceito nos seguintes tipos de coleções adicionais: (1) *coleções de códigos de alunos*: cada aluno gera uma coleção formada pelos códigos-fonte desenvolvidos por esse aluno; (2) *coleções de códigos de professores*: cada professor gera uma coleção formada pelos códigos-fonte desenvolvidos por alunos desse professor; (3) *coleções de códigos de questões*: cada questão gera uma coleção formada pelos códigos-fonte desenvolvidos para essa questão.

Desta forma, neste trabalho presume-se que a exploração do conceito de raridade nas coleções mencionadas anteriormente pode influenciar de forma positiva os métodos de recuperação de informação aplicados ao problema de detecção de plágio. Com base nessa hipótese, foram geradas as variações de *idf* listadas na Tabela 4.5.

A variante $idf_q(t, q)$ é uma medida da raridade de t nos códigos desenvolvidos para solucionar a questão q. Observa-se que *tokens* raramente usados em uma questão (alto valor de $idf_q(t, q)$) são mais úteis para detecção de plágios nas soluções dessa questão. Por outro lado, *tokens* típicos dessa questão, isto é, *tokens* que fazem parte do estilo da questão, não são úteis para detecção de plágio, e invariavelmente possuem

Tabela 4.5: Variações da métrica *Inverse Document Frequency* (idf) consideradas pelo método proposto.

	Variação	Descrição
1	$idf_q(t,q)$	Raridade de t nos códigos desenvolvidos para a questão q.
2	$idf_p(t,p)$	Raridade de t nos códigos desenvolvidos por alunos do professor p.
3	$idf_a(t,a)$	Raridade de t nos códigos desenvolvidos pelo aluno a.

Fonte: Adaptado de Palheta [2013].

baixo valor de $idf_q(t,q)$. Um ponto importante a ser observado é que um termo t pode possuir um baixo valor de $idf_q(t,q)$, sendo portanto pouco útil para detectar plágios em q, e mesmo assim possuir um alto valor de $idf(t)$.

Na mesma linha de raciocínio, a variante $idf_p(t,p)$ é uma medida da raridade de t nos códigos desenvolvidos por alunos do professor p. O uso desta métrica parte da hipótese de que o professor influencia o estilo de seus alunos. Desta forma, termos com baixo valor de $idf_p(t,p)$ e alto valor de $idf(t)$ podem ser resultado desta influência, e tipicamente não são úteis para a detecção de plágio em códigos desenvolvidos por alunos de p.

A métrica $idf_a(t,a)$ representa a raridade com que o aluno a usa o token t. Essa métrica é diferente das demais, visto que ela é usada para identificar termos t que fazem parte do estilo de programação do aluno a, e isso ocorre apenas quando $idf_a(t,a)$ é baixo. Nota-se que, se um token t possui um baixo valor de $idf_a(t,a)$ e um alto valor de $idf(t)$, isto é, se o termo é muito usado por a e pouco usado pelos demais alunos, então t pode ser bastante útil para diferenciar códigos de a dos códigos dos demais alunos.

A Tabela 4.6 apresenta outras variações da métrica idf, criadas através da combinação das métricas descritas anteriormente. Das linhas 1 a 4, utilizou-se o produto para combinar tais métricas, e das linhas 5 a 7 foi utilizada a soma para esse mesmo fim. Tais combinações de produto e soma não envolvem a métrica $idf_a(t,a)$, visto que essa métrica é usada para identificar termos que são típicos do estilo de um aluno, e isso ocorre quando o valor de $idf_a(t,a)$ é pequeno. Desta forma, a métrica $idf_a(t,a)$ é usada para gerar outras variações de idf, listadas das linhas 8 a 12 da Tabela 4.6. Essas métricas usam a operação de divisão para combinar $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$ com a métrica $idf_a(t,a)$.

No total, são avaliadas 17 variações da métrica idf nos dois modelos estudados, Vetorial e Probabilístico BM-25, com o intuito de determinar se as informações de estilos de programação podem ser úteis, para melhorar o processo de detecção de plágio. O resultado desse passo gera uma lista (*ranking*) de códigos-fonte, ordenados pelo grau de similaridade.

Tabela 4.6: Outras variações do *idf*, criadas pela combinação das métricas anteriores.

	Variação	Descrição
1	$idf(t) \times idf_p(t,p)$	Produto de $idf(t)$ e $idf_p(t,p)$
2	$idf(t) \times idf_q(t,q)$	Produto de $idf(t)$ e $idf_q(t,q)$
3	$idf_p(t,p) \times idf_q(t,q)$	Produto de $idf_p(t,p)$ e $idf_q(t,q)$
4	$idf(t) \times idf_p(t,p) \times idf_q(t,q)$	Produto de $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$
5	$idf(t) + idf_p(t,p)$	Soma de $idf(t)$ e $idf_p(t,p)$
6	$idf(t) + idf_q(t,q)$	Soma de $idf(t)$ e $idf_q(t,q)$
7	$idf_p(t,p) + idf_q(t,q)$	Soma de $idf_p(t,p)$ e $idf_q(t,q)$
8	$idf(t) + idf_p(t,p) + idf_q(t,q)$	Soma de $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$
9	$idf(t) / idf_a(t,a)$	Divisão de $idf(t)$ e $idf_a(t,a)$
10	$idf_p(t,p) / idf_a(t,a)$	Divisão de $idf_p(t,p)$ e $idf_a(t,a)$
11	$idf_q(t,q) / idf_a(t,a)$	Divisão de $idf_q(t,q)$ e $idf_a(t,a)$
12	$idf(t) \times idf_p(t,p) \times idf_q(t,q) / idf_a(t,a)$	Divisão de $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$ por $idf_a(t,a)$
13	$idf(t) + idf_p(t,p) + idf_q(t,q) / idf_a(t,a)$	Divisão de $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$ por $idf_a(t,a)$

Fonte: Adaptado de Palheta [2013].

4.3 Análise para Classificar Plágios e Não-Plágios

A maioria dos autores [Prechelt et al., 2002; Burrows et al., 2007; Mariani & Micucci, 2012; Đurić & Gašević, 2012] considera o problema do plágio como um problema de classificação. Dessa forma, é necessário que os métodos avaliados nesta dissertação sejam capazes de realizar tal classificação. Nesse caso, como classificar em plágios e não-plágios os valores de similaridade obtidos entre os pares de códigos não só pelo método proposto, mas também pelos *baselines*?

Nesta pesquisa, optou-se por usar algoritmos de agrupamento de dados para classificação dos códigos entre plágios e não-plágios. De acordo com Bussab [1990], agrupamento de dados (*clustering*) é uma técnica que identifica associações ou correlações entre objetos. Esse agrupamento tem a finalidade de formar grupos de objetos com alta similaridade e baixa similaridade em relação a objetos de outros grupos. Normalmente, o conceito de similaridade está associado à distância entre os objetos.

A estratégia adotada pauta-se no cálculo das distâncias entre os pontos adjacentes e são escolhidas as maiores distâncias para realizar o agrupamento. É importante que essa estratégia seja capaz de determinar agrupamentos que preservem proximidade entre elementos de um mesmo agrupamento e excluam de um mesmo grupo elementos com grandes distâncias entre si. Essa estratégia é explicada detalhadamente a seguir:

1. Dado um conjunto de valores ordenados $S = \{s_1, s_2, s_3, \ldots, s_n\}$ separados em um espaço unidimensional, e seja $D = \{(d_1, 1), (d_2, 2), (d_3, 3), \ldots, (d_{n-1}, n-1)\}$ o conjunto das distâncias entre os pontos adjacentes de S, onde d_i representa a distância entre os pontos e i representa posição dos pontos adjacentes;

2. É necessário ordenar o conjunto D por as distâncias d_i, desta forma as últimas

posições de D representam as maiores distâncias entre os pontos do conjunto S. É importante salientar que por meio da dupla (d_i, i) é possível obter a distância e o ponto para segmentação.

3. Encontre os k pontos para segmentação do conjunto S em k+1 conjuntos menores. Esses pontos estão entre os pontos adjacentes mais distantes entre si ainda não separados. Por exemplo, para $k = 1$, S é segmentado em dois conjuntos menores S^1 e S^2, e o ponto para segmentação está entre s_i e s_{i+1}, com i definido pela última posição do conjunto ordenado D;

4. O agrupamento é feito com base nos k pontos selecionados, resultando em $k + 1$ agrupamentos. Assim, para esta dissertação, apenas o último agrupamento é considerado, visto que ele contém o valor do topo do *ranking*.

Para exemplificar, considere o conjunto de valores S =$\{0.14, 0.20, 0.55, 0.56, 0.89, 0.99, 0.99\}$ e D =$\{(0.06, 1), (0.35, 2), (0.01, 3), (0.33, 4), (0.1, 5), (0.0, 6)\}$ o conjunto das distâncias entre os pontos adjacentes de S.

Primeiro, deve-se ordenar o conjunto D por os valores de similaridade, desta forma D =$\{(0.0, 6), (0.01, 3), (0.06, 1), (0.1, 5), (0.33, 4), (0.35, 2)\}$. Em seguida deve-se escolher os k pontos para segmentação. Foram realizados experimentos com valores de $k = \{1, 2, 3, 4$ e $5\}$. Com base nesse conjunto de experimentos, foi adotado o valor de $k = 2$, pois foi o que obteve um resultado satisfatório. Os 2 (dois) pontos com as maiores distâncias são os últimos apresentados em D, ou seja, os pontos 4 e 2.

Portanto, o conjunto S é dividido em três partes a partir dos dois pontos de segmentação escolhidos. O primeiro agrupamento é representado por $[0.14, 0.20]$, o segundo é representado por $[0.55, 0.56]$, e o terceiro e último agrupamento é representado por $[0.89, 0.99, 0.99]$. Para validação dos métodos será usado apenas o terceiro agrupamento, como mencionado anteriormente.

Por meio do agrupamento, os códigos são classificados entre plágios e não-plágios. Dessa forma, é possível realizar a validação do método proposto e *baselines*. No geral, essa validação é feita através de medidas de precisão e revocação, conforme encontrado nos trabalhos de Prechelt et al. [2002], Burrows et al. [2007], Mariani & Micucci [2012] e Đurić & Gašević [2012].

Antes de definir essas medidas, é necessário entender os quatro tipos possíveis de resultados de um método de classificação. Esses resultados, quando contabilizados, determinam a eficácia de um método com base na precisão e na revocação.

Considerando os casos identificados como plágio, estes casos podem ser compostos por os que são de fato plágios, chamados de verdadeiros positivos (VP) e por aqueles

que não são, chamados de falsos positivos (FP). Em contrapartida, os casos que não foram identificados como plágio, podem existir os que de fato não são plágios, chamados de verdadeiros negativos (VN) e os que são plágios, mas não foram identificados dessa forma pelo método, chamados de falsos negativos (FN).

A precisão, denominada por P, é a proporção de pares de códigos-fonte classificados como plágio que realmente são, isto é:

$$P = \frac{Total\ de\ Pares\ de\ Plágios\ Recuperados\ (VP)}{Total\ de\ Pares\ Recuperados\ (VP{+}FP)} \tag{4.1}$$

Desse modo, o valor de P é 1 (um) quando não há falso positivo.

A revocação, denominada por R, é a proporção de pares de códigos-fonte plagiados que foram corretamente classificados como plágio, isto é:

$$R = \frac{Total\ de\ Pares\ de\ Plágios\ Recuperados\ (VP)}{Total\ de\ Pares\ de\ Plágio\ (VP{+}FN)} \tag{4.2}$$

Dessa maneira, o valor de R é 1 (um) quando não há falso negativo.

Diante do exposto é possível concluir que se um método recupera todos os pares de plágios, isso ocorre quando $R = 1$, não significa que o método é ideal, visto que podem existir falsos positivos (FP). Por outro lado, se um método recupera apenas os pares de plágios, o que ocorre quando $P = 1$, da mesma forma não significa que o método é ideal, visto que podem existir falsos negativos (FN).

Conforme mencionado anteriormente, os falsos negativos (FN) são os casos de plágio que o método classifica de maneira errada como não sendo plágios, e os falsos positivos (FP) são os casos que não são plágios e o método classifica-os como plágios. É importante observar que para um método de detecção de plágio, um FN não é tão grave quanto um FP. O motivo dessa observação pode ser explicado pelo fato de que os falsos positivos (FP) são críticos no cenário de detecção de plágio, visto que podem levar o professor a cometer injustiças com os alunos. Em virtude disso, para evitar ao máximo acusações sem fundamentos, esta pesquisa considera que a precisão é mais importante do que a revocação.

De forma a estimar a confiabilidade estatística dos resultados, são consideradas a precisão e revocação de cada método avaliado usando a estratégia mencionada anteriormente. Dessa forma, dado o resultado de dois métodos, são calculadas as medidas de precisão e revocação média de cada um e é calculado o Teste-t, de *Student*, pareado e bi-caudal [Witten et al., 1999].

A escolha pelo Teste-t se deve ao fato de que os experimentos possuem um número razoável de consultas, 50 códigos para cada coleção avaliada. O teste pareado foi

adotado porque a comparação dos métodos é dependente da consulta. O teste bi-caudal refere-se à estimativa de confiabilidade da diferença entre os métodos. Portanto, nesta dissertação são considerados confiáveis os resultados cujo o valor p do Teste-t for menor ou igual a 5% (grau de confiança de, pelo menos, 95%).

4.4 Considerações Finais

Neste capítulo, foi apresentado o método proposto para detecção de plágio em cursos iniciais de programação o qual considera informações de estilos de programação, e a métrica de avaliação adotada nesta dissertação. Além disso, o método propõe que a detecção pode ser realizada a partir da coleção de códigos (modelo tradicional) ou das coleções *idf*, idf_a, idf_p e idf_q, onde os códigos são agrupados por coleção, aluno, professor e questão, respectivamente. Dessa forma, são propostas variações no cálculo da métrica *idf*, para os modelos Vetorial e Probabilístico BM-25, a fim de avaliar a importância de informações de estilos de programação para o processo de detecção de plágio em códigos-fonte. No próximo capítulo são explicitados os experimentos realizados e os resultados obtidos nesta pesquisa.

Capítulo 5

Experimentos e Resultados

Este capítulo apresenta a avaliação do método proposto no Capítulo 4. São apresentadas as coleções sobre as quais são realizados os experimentos, os *baselines* adotados e as métricas de avaliação. Por fim, são expostos e discutidos os resultados obtidos a partir da comparação entre os *baselines* e a solução proposta nesta dissertação para o problema de detecção de plágio em códigos-fonte.

5.1 Coleções de Códigos-Fonte

As coleções de códigos-fonte utilizadas nos experimentos descritos neste capítulo advêm do sistema CodeBench, e são compostas por códigos desenvolvidos na linguagem C por alunos do Instituto de Computação da Universidade Federal do Amazonas (UFAM) como resposta às listas de exercícios elaboradas por seus professores. A escolha da linguagem C se deve ao fato dessa linguagem ser o padrão utilizado na instituição para o ensino de programação nos períodos iniciais.

No momento da confecção dessa coleção, no primeiro semestre de 2015, o sistema CodeBench possuía 2638 códigos-fonte desenvolvidos na linguagem C. Esses códigos foram desenvolvidos por alunos de disciplinas de programação presentes nos três primeiros semestres de seus respectivos cursos.

A coleção está organizada de acordo com a estrutura apresentada na Figura 5.1. Nessa figura, o primeiro nível é composto por professores que ministram as disciplinas de programação, o segundo nível representa as questões propostas por esses professores, e o terceiro e último nível representa os códigos-fontes desenvolvidos pelos alunos como respostas às questões propostas por um professor. Desta forma, cada código da coleção representa uma tentativa de um aluno de solucionar uma questão proposta por um professor.

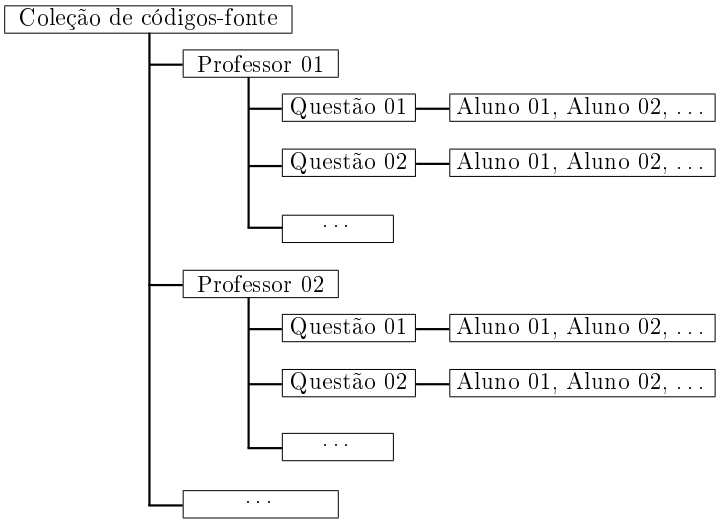

Figura 5.1: Estrutura da coleção de códigos-fonte do CodeBench.

Para avaliação da eficácia do método proposto nesta dissertação foram criadas duas coleções a partir dos 2638 códigos oriundos do CodeBench. A primeira, denominada *coleção natural*, é formada por casos de plágios reais e a segunda, denominada *coleção sintética*, é composta por casos de plágios simulados. A seguir, são descritos os processos de criação das duas coleções.

5.1.1 Coleção Natural

A partir dos códigos do CodeBench, foi selecionado um subconjunto aleatório de 50 códigos para compor a *coleção natural*. De agora em diante, será usada a terminologia *código de consulta* para referenciar cada um desses códigos. É importante salientar que cada código de consulta representa a tentativa de solução de um aluno para um exercício de programação proposto por seu professor. No total, os 50 códigos de consulta foram desenvolvidos para 38 questões distintas. Além dos 50 códigos de consulta, também foram incluídos na coleção todos os códigos desenvolvidos para cada uma dessas 38 questões, totalizando 466 códigos.

Dado um código de consulta C desenvolvido para uma questão Q, e dada a lista L_C de códigos-fonte desenvolvidos para a mesma questão Q, o método proposto nesta pesquisa será avaliado de acordo com sua capacidade de identificar em L_C os eventuais casos de plágio que envolvem o código de consulta C.

A coleção foi analisada manualmente para que fossem identificados os casos de plágio que envolvem cada um dos códigos de consulta. Essa análise contou com a ajuda de 3 especialistas, de forma que cada especialista detectou os casos de plágio em

cerca de 16 códigos de consulta. De posse de um código de consulta C e sua respectiva lista L_C, o especialista foi convidado a identificar manualmente em L_C as tentativas de soluções que foram plagiadas ou que serviram de fonte de plágio para C.

Através dessa análise foi possível identificar que a maioria das questões apresentam soluções menores, com cerca de 5-20 linhas. Entretanto, algumas questões apresentaram soluções maiores, com cerca de 70-80 linhas. É interessante observar que isso indica que os alunos das disciplinas iniciais solucionam as questões propostas por o professor com mais ou menos essa quantidade de linhas de código, o que caracteriza que a coleção é representada por códigos feitos por alunos iniciantes em programação.

Ao término da análise, os especialistas identificaram 93 pares com casos de plágios reais nessa coleção. Em virtude da quantidade de casos de plágios encontrados, é necessária a criação de um método que seja capaz de realizar a detecção nos códigos dessa coleção.

5.1.2 Coleção Sintética

A *coleção sintética* foi elaborada com o intuito de complementar os experimentos, visto que a coleção apresentada na seção anterior apresentou apenas 50 consultas com casos de plágios identificados, o que denota uma quantidade pequena de consultas para este tipo de experimento.

Nas pesquisas realizadas, foi encontrado o artigo de Liu et al. [2006], que propõe um método para simular plágios em códigos-fonte. De acordo com Liu et al. [2006], as principais modificações realizadas pelos alunos para dissimular a cópia de código-fonte são:

1. Alteração de nome de variáveis;
2. Reordenação de funções, se apresentarem mais de duas funções;
3. Substituição de laços de repetição *while* por *for* e vice-versa. O *for* é substituído por um laço *while(1)* com interrupção através do *break*;
4. Substituição do *if* (exp) {1}, por algumas das modificações abaixo:
   a) *if* (!(!exp)) {1};
   b) *if* (exp==1) {1};
   c) *if* (!(!exp==1)== 1) {1}.
5. Substituição do *if*(exp) {1} else {0} por *if*(!(exp)) {0} else {1}.

Para a criação dessa coleção, foi selecionado um subconjunto aleatório de 50 códigos de consulta, a partir dos 2638 códigos do CodeBench. Os 50 códigos de consulta foram desenvolvidos para 38 questões distintas. Conforme mencionado anteriormente, além dos 50 códigos de consulta, foram incluídos também todos os códigos desenvolvidos para cada uma dessas 38 questões, totalizando 487 códigos.

Os *plágios sintéticos* são gerados através dos seguintes passos: (i) inicialmente, é feita uma cópia do código de consulta; (ii) são aplicadas todas as modificações propostas por Liu et al. [2006] na cópia gerada; (iii) essa cópia é atribuída a um aluno fictício; e depois (iv) incluída na coleção de códigos de forma a simular um plágio.

Ao término do processo de criação dos *plágios sintéticos* foi utilizado o compilador *gcc* para verificar se a gramática do código permanecia correta. Somando os *plágios sintéticos* à quantidade códigos que a coleção possuía, a *coleção sintética* tem, ao todo, 537 códigos-fonte.

Com os códigos devidamente rotulados entre plágios e não-plágios é possível avaliar a eficácia dos métodos experimentados nesta dissertação. A Tabela 5.1 sumariza as informações mais importantes sobre cada coleção. Nessa tabela, a coluna Coleção especifica o nome das coleções, a coluna Questões especifica a quantidade de questões distintas na coleção, e a coluna Consulta indica a quantidade de consultas para cada coleção. A quantidade de pares de códigos a serem analisados pelos métodos experimentados pode ser vista na coluna Pares. Por fim, a quantidade de pares que são e não são plágios são descritas nas colunas Plágio e Não-Plágio, respectivamente.

Tabela 5.1: Informações sobre as coleções de códigos.

Coleção	Questões	Consultas	Códigos	Pares	Plágio	Não-Plágio
Natural	38	50	466	3094	93	3001
Sintética	38	50	537	3921	50	3871

5.2 Baselines

O método proposto foi comparado com três *baselines*, um baseado em *Abstract Syntax Trees* (ASTs) e dois baseados em *tokens*, são eles: CodeCompare, AuDeNTES e JPlag. Conforme explicado na Seção 3.1.3, o CodeCompare [Tao et al., 2013] é um método baseado em árvore (ASTs). Para geração da árvore AST e manipulação dos dados foram utilizadas duas ferramentas clássicas na área de compiladores: *Lex* e *Yacc*, os quais são geradores de analisadores léxicos e sintáticos, respectivamente [Levine et al., 1992].

Outro *baseline* adotado é o método AuDeNTES proposto por Mariani & Micucci [2012], onde somente é possível analisar códigos na linguagem Java. Conforme mencionado na Seção 3.1.2, o AuDeNTES usa as ferramentas JavaCC e JTB para transformar o código em *tokens* ASTs. De acordo com Mariani & Micucci [2012], para que o método seja adaptado para outras linguagens, é necessário modificar a gramática JavaCC, fornecida como entrada para a JTB, e reimplementar alguns métodos. Desse modo, essas adaptações foram realizadas na implementação[1] disponibilizada pelos autores, para que o método pudesse analisar os códigos desenvolvidos na linguagem C.

Conforme descrito na Seção 3.1.2, o método AuDeNTES requer um código de referência para cada questão. O código de referência é um exemplo de solução fornecido pelo professor da turma, e é usado para diferenciar os estilos da questão dos estilos dos alunos. Como as coleções descritas nas seções anteriores não possuem esses códigos, foi realizado um conjunto de experimentos para identificar os códigos que poderiam ser adotados como referências. Para cada código de consulta C, foi a avaliada a precisão do método AuDeNTES ao adotar cada código $c \in L_C$ como código de referência. Nesse conjunto de experimentos, foram adotados como referências os códigos $c \in L_C$ que retornaram maior precisão.

É importante ressaltar que esta pesquisa supõe que o código-fonte de um aluno pode simular a existência do código de referência. Esse pressuposto resulta do fato de que a solução de referência é uma das possíveis soluções para a questão, e tudo o que for semelhante a essa referência estará correto como resposta a tal questão.

O último *baseline* adotado é o método JPlag [Prechelt et al., 2002], implementado pela mesma ferramenta do AuDeNTES. De acordo com Mariani & Micucci [2012], esta implementação difere da original ao permitir que os códigos sejam analisados pelo mesmo conjunto de tokens do AuDeNTES, gerando resultados similares aos obtidos pela versão disponível na Web.

5.3 Configuração dos Experimentos

Antes de executar os experimentos foi realizado um pré-processamento nos códigos da coleção. Conforme mencionado nos Capítulos 3 e 4, os métodos adotados como *baseline* e o método proposto realizam essa etapa de pré-processamento.

Para o método proposto foram removidos comentários, referências aos arquivos externos (bibliotecas) e todos os caracteres situados entre aspas (“ ” e ’ ’). Nos métodos CodeCompare, AuDeNTES e JPlag foram removidos comentários, linhas em branco

[1] A implementação do método pode ser obtida em http://www.sal.disco.unimib.it/technologies/audentes/.

e referências aos arquivos externos (bibliotecas). Além disso, no CodeCompare foram substituídas as referências "*define*" e "*typedef*". É importante salientar, que os métodos AuDeNTES e JPlag já possuem um pré-processamento específico.

Após o passo de pré-processamento, os *baselines* já podem realizar o processo de comparação para verificação de possíveis plágios. No entanto, para o método proposto, ainda é necessário realizar os passos de tokenização (cf. Seção 4.2.2) e indexação (cf. Seção 4.2.3). Assim, é possível realizar as consultas e realizar as comparações na busca de possíveis plágios.

5.4 Experimentos e Resultados

Nesta seção, é apresentada a avaliação comparativa entre os resultados de todos os métodos estudados.

5.4.1 Comparação entre os Modelos Clássicos

Os resultados apresentados nesta seção são usados para selecionar uma dentre as estratégias de cálculo de similaridade propostas na Seção 4.2.4.1. Foram realizados experimentos com os modelos Vetorial e Probabilístico BM-25. Nesta comparação, cada modelo foi avaliado individualmente para perceber como é o seu funcionamento no processo de detecção de plágio. A comparação é feita com base nos 100 códigos de consulta mencionados na Seção 5.1, no qual 50 são para a *coleção natural* e 50 são para a *coleção sintética*.

A Tabela 5.2 apresenta os níveis de precisão e revocação obtidos ao aplicar os modelos Vetorial e Probabilístico BM-25 nas coleções descritas na Seção 5.1. Nessa tabela, as linhas representam os modelos comparados e as colunas os níveis de precisão (P) e revocação (R) dos modelos para cada coleção.

Tabela 5.2: Níveis de precisão (P) e revocação (R) para os modelos Vetorial e Probabilístico BM-25 nas coleções natural e sintética.

	Coleção de Códigos			
	Natural		Sintética	
Modelo	P	R	P	R
Vetorial	0,904	0,902	0,960	0,980
Probabilístico BM-25	**0,915**	0,878	**0,977**	1,000

Antes de discorrer sobre os resultados, é importante salientar que esta pesquisa considera que a precisão é mais importante do que a revocação. No contexto desta dissertação, é importante que método seja minucioso ao fazer a classificação, para que não acuse injustamente um aluno de plágio, conforme mencionado na Seção 4.3.

Como pode-se perceber, na Tabela 5.2, o modelo probabilístico BM-25 obteve melhores níveis de precisão, não só quando aplicado na *coleção natural* mas também quando aplicado na *coleção sintética*. Portanto, o modelo escolhido para compor o método de detecção de plágio proposto nesta pesquisa foi o probabilístico BM-25, por apresentar uma precisão maior do que o modelo vetorial nas duas coleções de códigos avaliadas.

Logo, o modelo vetorial não será aplicado nas próximas comparações. Entretanto, no Apêndice A estão presentes as tabelas com os níveis de precisão e revocação obtidos nos experimentos com o modelo vetorial. A próxima seção trata do experimento com o método proposto e sua comparação com os *baselines*.

5.4.2 Comparação entre Modelos baseados em BM-25 e Baselines

Nesta seção, são apresentados os resultados obtidos com o método proposto no Capítulo 4. As Seções 5.4.2.1 e 5.4.2.3 apresentam as análises do método no cenário em que os plágios são naturais e sintéticos, respectivamente.

5.4.2.1 Análise de Plágios Naturais

Esta seção apresenta a análise do método proposto no cenário em que os plágios são naturais. Conforme descrito na Seção 5.1.1, os plágios naturais compreendem os plágios oriundos do CodeBench identificados manualmente. Nessa análise, em particular, são comparadas as variantes propostas nesta dissertação e escolhida a variante com maior precisão para comparação com os *baselines*.

O resultado dessa análise é apresentado na Tabela 5.3. Nessa tabela, as linhas representam as variantes de *idf* consideradas pelo método proposto e as colunas mostram os níveis de precisão (P) e revocação (R) obtidos por cada variante. Os valores em destaque são os melhores níveis de precisão obtidos.

É importante salientar que para esta dissertação, o foco principal pauta-se nos valores de precisão, devido a questão dos falsos positivos, conforme mencionado anteriormente. Em virtude disso, diante dos resultados apresentados na Tabela 5.3, as variantes das linhas 9, 12 e 16 se destacaram nos níveis de precisão em relação as demais variações propostas nesta pesquisa.

Dentre as três melhores variantes apresentadas na Tabela 5.3, a variante da linha 12 usa a combinação de 3 métricas: $idf(t)$, $idf_p(t,p)$ e $idf_q(t,q)$, a variante da linha 16 usa a combinação de 4 métricas: $idf(t)$, $idf_p(t,p)$, $idf_q(t,q)$ e $idf_a(t,a)$, enquanto

Tabela 5.3: Níveis de precisão (P) e revocação (R) com as variantes de *idf* propostas nesta pesquisa para a análise de plágios naturais.

	Variante	P	R
1	$idf(t)$	0,915	0,878
2	$idf_a(t,a)$	0,924	0,880
3	$idf_p(t,p)$	0,910	0,871
4	$idf_q(t,q)$	0,885	0,853
5	$idf(t) \times idf_p(t,p)$	0,915	0,863
6	$idf(t) \times idf_q(t,q)$	0,915	0,870
7	$idf_p(t,p) \times idf_q(t,q)$	0,910	0,855
8	$idf(t) \times idf_p(t,p) \times idf_q(t,q)$	0,917	0,855
9	$idf(t) + idf_p(t,p)$	**0,927**	0,867
10	$idf(t) + idf_q(t,q)$	0,923	0,861
11	$idf_p(t,p) + idf_q(t,q)$	0,907	0,873
12	$idf(t) + idf_p(t,p) + idf_q(t,q)$	**0,927**	0,867
13	$idf(t) / idf_a(t,a)$	0,907	0,907
14	$idf_p(t,p) / idf_a(t,a)$	0,879	0,874
15	$idf_q(t,q) / idf_a(t,a)$	0,893	0,874
16	$idf(t) \times idf_p(t,p) \times idf_q(t,q) / idf_a(t,a)$	**0,927**	0,856
17	$idf(t) + idf_p(t,p) + idf_q(t,q) / idf_a(t,a)$	0,893	0,868

a variante da linha 9 usa a combinação de 2 métricas: $idf(t)$ e $idf_p(t,p)$. Ou seja, as variantes 12 e 16 possuem um quantitativo maior de combinações da métrica *idf* na sua composição. É possível observar que o uso da variante 9 é suficiente para se obter resultados da mesma ordem.

Dessa forma, a variante da linha 9 ($idf(t) + idf_p(t,p)$) foi escolhida como base de comparação com os *baselines* por possuir um quantitativo menor de combinações da métrica *idf*. Esse resultado provavelmente está relacionado com o fato de que os códigos da coleção utilizada nos experimentos foram desenvolvidos por alunos com pouca experiência em programação e, portanto, ainda sem um estilo bem definido, esses alunos certamente são influenciados pelo estilo do professor.

Com base nos resultados obtidos, a variante apresentada na linha 9 da Tabela 5.3 é referenciada a seguir como Método Proposto. O modelo clássico, apresentado na linha 1, também é escolhido como base de comparação e é referenciado a seguir como BM-25.

A Tabela 5.4 apresenta o resultado do método proposto com seus respectivos ganhos/perdas em relação aos *baselines*. Nessa tabela as linhas representam o método proposto e os *baselines* (*AuDeNTES*, *JPlag* e *CodeCompare*), e as colunas apresentam os níveis de precisão (P), revocação (R) e o percentual de ganho/perda ($G\%$) para cada método.

Observa-se, na Tabela 5.4, que o resultado que mais se destaca é o do método proposto nesta dissertação, o qual obteve uma precisão de 0,927. Esse resultado pode

Tabela 5.4: Níveis de precisão (P) e revocação (R) para o método proposto e *baselines*.

Método	P	$G(\%)$	R	$G(\%)$
Método Proposto	0,927	-	0,867	-
BM-25 [Robertson et al., 2000]	0,915	1,29	0,878	-1,27
AuDeNTES [Mariani & Micucci, 2012]	0,854	7,87	0,865	0,23
JPlag [Prechelt et al., 2002]	0,790	14,78*	0,931	-7,38
CodeCompare [Tao et al., 2013]	0,768	17,15*	0,854	1,50

O asterisco "*" indica ganhos estatisticamente significativos com, pelo menos, 95% de confiança.

ser explicado pelo fato de que os estilos de programação podem melhorar o processo de detecção de plágio. Como a coleção é composta por códigos de alunos iniciantes em programação, as questões possuem enunciados simples e, portanto, tendem a ter soluções parecidas. Por isso, os *baselines* não obtiveram resultados satisfatórios.

O método CodeCompare [Tao et al., 2013], apresentou um dos menores níveis de precisão em relação aos demais métodos comparados. Isso se deve ao fato da grande ocorrência de falsos positivos indicada pelo método. Esse método é baseado em *Abstract Syntax Trees* (AST), sua representação remove os fragmentos do código-fonte que representam os estilos de programação mantendo apenas a estrutura sintática do código-fonte, conforme mencionado na Seção 3.1.3. Em virtude disso, os códigos-fonte que representam soluções mais simples (códigos pequenos) provavelmente ficaram mais parecidos, o que acarretou na ocorrência de muitos falsos positivos.

O método AuDeNTES [Mariani & Micucci, 2012], apresentou o maior nível de precisão dentre os *baselines* avaliados. Este resultado provavelmente está relacionado com a solução de referência requerida por ele. Por meio da solução de referência, os fragmentos comuns entre o par de códigos-fonte analisado e essa solução são removidos. O que significa que existia uma solução mais similar ao par de códigos analisados e o método conseguiu identificar melhor os plágios que os outros *baselines* avaliados.

Por fim, o *JPlag* também não obteve um resultado promissor nesse experimento. O motivo desse resultado é explicado pelo fato de o método transformar os códigos-fonte em *tokens* ASTs. *Tokens* que representam os estilos de programação, tais como: espaços em branco, tabulações e quebras de linha são removidos. Da mesma forma que no método *CodeCompare*, provavelmente essa representação deixou os códigos-fonte pequenos com uma maior similaridade. Esse método obteve um nível de revocação maior dentre todos os analisados, o que significa que mais casos de plágio foram descobertos, entretanto sua precisão obteve o segundo menor nível, apresentando uma grande ocorrência de falsos positivos.

5.4.2.2 Discussão

Em geral, a partir dessa análise é possível concluir que os *baselines* não obtiveram resultados satisfatórios em consequência das suas formas de representação. Métodos que se baseiam em *tokens* ASTs e métodos que se baseiam puramente em ASTs não se comportam de maneira adequada quando os códigos-fonte são pequenos e as soluções são parecidas. A remoção dos fragmentos que representam os estilos de programação, tais como: espaços em branco, tabulações e quebras de linha, faz com que os códigos fiquem praticamente iguais. Desta forma, os *baselines* apresentaram várias ocorrências de falsos positivos.

Na análise realizada na coleção de plágios naturais, é possível perceber que o método proposto apresenta um resultado promissor diante dos demais métodos estudados com ganhos estatisticamente comprovados pelo Teste-T, nos níveis de precisão, em relação a JPlag e CodeCompare. Embora o método proposto não tenha obtido tais ganhos quando comparado ao AuDeNTES [Mariani & Micucci, 2012], ele ainda se mostra bastante promissor, visto que não necessita da solução de referência requerida por ele. Conforme discutido na Seção 5.2, fez-se necessária a realização de experimentos com todas as soluções desenvolvidas pelos alunos para uma dada questão a fim de encontrar a solução que apresentasse o melhor resultado.

Por fim, vale ressaltar que o modelo clássico BM-25 já é suficiente para se obter ganhos sobre os *baselines* avaliados. No entanto, ao combinar as métricas de *idf*, o método proposto apresenta resultados melhores que o modelo tradicional.

5.4.2.3 Análise de Plágios Sintéticos

Esta seção apresenta a análise do método proposto no cenário em que os plágios são sintéticos. Conforme descrito na Seção 5.1.2, os plágios sintéticos compreendem os plágios criados para simular a ocorrência de plágios na coleção de códigos. Da mesma forma que na seção anterior, são comparadas as variantes propostas nesta dissertação e escolhida a variante com maior precisão para comparação com os *baselines*.

O resultado dessa análise é apresentado na Tabela 5.5. Nessa tabela, as linhas representam as variantes de *idf* consideradas pelo método proposto e as colunas mostram (P) e (R) que são os níveis de precisão obtidos por cada variante. Os valores em destaque são os melhores níveis de precisão obtidos.

Conforme mencionado anteriormente, o foco principal desta dissertação pauta-se nos valores de precisão. Dessa forma, observa-se na Tabela 5.5 que as variantes das linhas 7 e 9 obtiveram os melhores níveis de precisão em relação as demais variações.

Tabela 5.5: Níveis de precisão (P) e revocação (R) com as variantes de *idf* propostas nesta pesquisa para a análise de plágios sintéticos.

	Variante	P	R
1	$idf(t)$	0,977	1,000
2	$idf_a(t,a)$	0,967	1,000
3	$idf_p(t,p)$	0,970	0,980
4	$idf_q(t,q)$	0,970	0,980
5	$idf(t) \times idf_p(t,p)$	0,980	1,000
6	$idf(t) \times idf_q(t,q)$	0,980	1,000
7	$idf_p(t,p) \times idf_q(t,q)$	**0,990**	1,000
8	$idf(t) \times idf_p(t,p) \times idf_q(t,q)$	0,980	1,000
9	$idf(t) + idf_p(t,p)$	**0,990**	1,000
10	$idf(t) + idf_q(t,q)$	0,980	1,000
11	$idf_p(t,p) + idf_q(t,q)$	0,970	0,980
12	$idf(t) + idf_p(t,p) + idf_q(t,q)$	0,980	1,000
13	$idf(t) / idf_a(t,a)$	0,937	0,960
14	$idf_p(t,p) / idf_a(t,a)$	0,947	0,960
15	$idf_q(t,q) / idf_a(t,a)$	0,947	0,960
16	$idf(t) \times idf_p(t,p) \times idf_q(t,q) / idf_a(t,a)$	0,970	0,980
17	$idf(t) + idf_p(t,p) + idf_q(t,q) / idf_a(t,a)$	0,947	0,960

Dentre os melhores resultados destacados na Tabela 5.5, nota-se que as duas variantes possuem a mesma quantidade de combinações da métrica *idf*, a variante da linha 7 usa a combinação de $idf_p(t,p)$ e $idf_q(t,q)$, e a variante da linha 9 usa a combinação de $idf(t)$ e $idf_p(t,p)$.

Como forma de padronização, no que tange a questão das variantes, optou-se por usar a variante da linha 9 como base de comparação com os *baselines*, visto que essa variante obteve o melhor resultado nos experimentos demonstrados na seção anterior. Esse resultado só reforça a hipótese de que os alunos iniciantes em programação não possuem um estilo bem definido, e muito possivelmente são influenciados pelo estilo do professor.

Com base nos resultados obtidos, a variante apresentada na linha 9 da Tabela 5.5 é referenciada a seguir como Método Proposto. O modelo clássico, apresentado na linha 1, também é escolhido como base de comparação e é referenciado a seguir como BM-25.

A Tabela 5.6 apresenta o resultado do método proposto com seus respectivos ganhos/perdas em relação aos *baselines*. Nessa tabela as linhas representam o método proposto e os *baselines* (*AuDeNTES*, *JPlag* e *CodeCompare*), e as colunas apresentam os níveis de precisão (P), revocação (R) e o percentual de ganho/perda ($G\%$) para cada método.

De acordo com a Tabela 5.6, nota-se que o método proposto é o que mais se destaca em níveis de precisão em relação a todos os *baselines* estudados. Os *baselines*,

Tabela 5.6: Níveis de precisão (P) e revocação (R) para o método proposto e *baselines*.

Método	P	$G(\%)$	R	$G(\%)$
Método Proposto	0,990	-	1,000	-
BM-25 [Robertson et al., 2000]	0,977	1,31	1,000	0,00
AuDeNTES [Mariani & Micucci, 2012]	0,518	47,68*	0,900	10,00
JPlag [Prechelt et al., 2002]	0,450	54,55*	0,680	32,00*
CodeCompare [Tao et al., 2013]	0,682	31,11*	0,880	12,00*

O asterisco "*" indica ganhos estatisticamente significativos com, pelo menos, 95% de confiança.

exceto o *CodeCompare*, apresentaram uma forte queda nos níveis de precisão em relação à análise feita na seção anterior. Essa queda pode ser explicada pelo fato de que os códigos dessa coleção sofreram modificações em suas estruturas sintáticas, conforme explanado na Seção 5.1.2.

O método *CodeCompare* apresentou o melhor nível de precisão dentre os *baselines* avaliados. Esse resultado provavelmente está relacionado ao fato de esse método possuir como foco alterações nas estruturas condicionais, conforme mencionado na Seção 3.1.3. Desta forma, alterações desse tipo não afetam a eficácia do método. Entretanto, as modificações realizadas não são somente nas estruturas condicionais, conforme visto na Seção 5.1.2. Por isso, o método apresentou uma queda nos níveis de precisão.

O método AuDeNTES [Mariani & Micucci, 2012], apresentou o menor nível de precisão dentre todos os métodos comparados. Esse resultado se deve à solução de referência que o método adota. Nesse caso, somente os fragmentos comuns à solução de referência são removidos. Como os códigos sofreram várias modificações, os fragmentos que sobraram provavelmente eram os modificados e, portanto, o método não conseguiu identificá-los.

Por fim, o *JPlag* apresentou os menores níveis de precisão e revocação dentre todos os métodos. Conforme explicado na seção anterior, isso se deve ao fato à sua forma de representação. Esse resultado mostra que o método foi o mais sensível as modificações realizadas.

5.4.2.4 Discussão

No que diz respeito a eficácia dos métodos no cenário em que os códigos são oriundos de plágios criados propositalmente, os *baselines* não obtiveram resultados satisfatórios. Esse resultado provavelmente está relacionado com as modificações feitas nas estruturas dos códigos, tais mudanças fazem com que os códigos que apresentem a mesma estrutura sejam mais similares que as próprias versões modificadas. Esses resultados mostraram que os métodos são sensíveis a mudanças na estrutura sintática dos códigos-fonte. É importante salientar que os códigos da coleção são pequenos, e muito

provavelmente por isso os *baselines* apresentaram esse resultado.

Na análise realizada na coleção de plágios sintéticos, é possível perceber que o método proposto se manteve promissor diante dos demais métodos estudados, com ganhos nos níveis de precisão estatisticamente comprovados pelo Teste-T. Tal resultado se deve novamente aos estilos de programação considerados pelo método. Portanto, as alterações refletem nos fragmentos modificados, e os outros fragmentos que representam os estilos de programação são preservados.

É importante observar que, da mesma forma como no experimento apresentado na seção anterior, o modelo clássico BM-25 obteve ganhos sobre os *baselines* avaliados. Entretanto, com o uso dos estilos é possível se obter resultados melhores.

5.5 Considerações Finais

Neste capítulo, foram apresentados os experimentos realizados para avaliação do método proposto e sua comparação com os *baselines* adotados nesta dissertação. Foram feitas três análises. Na primeira, foram avaliados os modelos Vetorial e Probabilístico BM-25 com o objetivo de escolher o modelo para compor o método de detecção. Na segunda e na terceira análise, o método proposto foi avaliado no cenário de plágios naturais e plágios sintéticos, respectivamente. Através dos resultados obtidos, é possível concluir que o método proposto foi mais preciso ao detectar plágio do que os *baselines* estudados. Os resultados também indicam que, como esperado, os professores influenciam o estilo de programação dos alunos iniciantes. Com isso, comprova-se que o uso de estilos de programação podem melhorar o processo de detecção de plágio.

O próximo capítulo apresenta as conclusões obtidas nesta pesquisa, e em seguida são tratadas as limitações, as contribuições alcançadas e os tópicos que servirão de base para direcionar os trabalhos futuros.

Capítulo 6

Conclusão

Este capítulo finaliza esta dissertação de mestrado e apresenta: (i) as considerações finais da pesquisa; (ii) as limitações encontradas; (iii) as contribuições alcançadas; e (iv) as possibilidades de trabalhos futuros.

6.1 Considerações Finais

Nesta dissertação, foi proposto um método para detecção de possíveis plágios entre códigos-fonte gerados em aulas práticas de turmas de programação, com o intuito de auxiliar o professor/monitor no acompanhamento e avaliação de turmas numerosas, o que ocorre com frequência nessas turmas iniciais.

Em virtude de se tratar de uma análise que se concentra especificamente na elaboração de um método que seja capaz de identificar, de maneira eficaz, possíveis plágios entre os códigos-fonte de alunos, foram discutidas as principais modificações empregadas por eles, o que serviu de base para a criação da *coleção sintética*. Além disso, as principais técnicas encontradas na literatura no que concerne à detecção de plágio em códigos-fonte foram apresentadas e discutidas.

A contribuição desta dissertação se concentra principalmente na formulação de um método que use informações de estilos de programação de aluno, professor e questão. Essas informações foram incorporadas a dois modelos clássicos de RI: o modelo Vetorial e o modelo Probabilístico BM-25, e foram feitos experimentos para que fosse escolhido o modelo que apresentasse melhor precisão no processo de detecção de plágio. Como resultado, o modelo que apresentou melhor precisão foi o Probabilístico BM-25, sendo este o escolhido para compor o método proposto nesta pesquisa.

Para avaliação do método proposto e dos demais métodos estudados foram realizados experimentos usando as duas coleções de códigos criadas nesta dissertação, a

coleção natural e a *coleção sintética*. Como resultado dos experimentos, o método proposto obteve resultados satisfatórios quando aplicado às duas coleções. Isso comprova que o uso de informações de estilos de programação podem melhorar o processo de detecção de plágio.

Como resultado desta pesquisa, o método proposto alcançou ganhos nos níveis de precisão da ordem de 7% e 17% na *coleção natural*, e de 30% e 40% na *coleção sintética*. Diante disso, foi possível observar que uma coleção com códigos-fonte de alunos iniciantes em programação representa um desafio para métodos de detecção de plágio, visto que os *baselines* classificaram quase todos os códigos como sendo plágios.

6.2 Limitações

A principal limitação desta pesquisa está relacionada ao estilo de programação do aluno. Para exemplificar, considere um hipotético caso em que o aluno A está acostumado a plagiar código-fonte do aluno B. Neste caso, os estilos de programação dos alunos A e B podem ser confundidos pelo método proposto, uma vez que ele considera que cada aluno tem seu estilo próprio.

6.3 Contribuições Alcançadas

A principal contribuição desta dissertação está no desenvolvimento de um método de detecção de plágio em códigos-fonte. Além dessa, são também contribuições desta pesquisa:

1. Revisão bibliográfica sobre as técnicas de detecção de plágio, onde é realizada uma análise comparativa dos diferentes métodos propostos pela literatura.

2. Um método para detecção de possíveis plágios que combina estilos de programação de aluno, professor e questão.

3. Criação de duas coleções de códigos: (i) *coleção natural*, composta por casos de plágios reais e (ii) *coleção sintética*: composta por casos de plágios simulados.

6.4 Trabalhos Futuros

Embora esta dissertação apresente resultados satisfatórios, ela também abre caminho para novas pesquisas. Algumas perspectivas de trabalhos futuros estão elencadas a seguir:

1. Avaliar o método proposto em outras coleções de códigos-fonte, com o intuito de verificar seu comportamento em outros ambientes de programação.

2. Aplicar técnicas para que se mantenha nas coleções de códigos de aluno, professor e questão apenas os estilos mais comuns, o que faz com que tais coleções não cresçam desordenadamente.

3. Avaliar o impacto do tamanho das coleções de códigos de aluno, professor e questão no processo de detecção de plágio.

Referências Bibliográficas

Ali, A. M. E. T.; Abdulla, H. M. D. & Snásel, V. (2011). Overview and comparison of plagiarism detection tools. Em *DATESO*, pp. 161--172. Citeseer.

Anh, V. N. & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151--166.

Arasu, A.; Cho, J.; Garcia-Molina, H.; Paepcke, A. & Raghavan, S. (2001). Searching the web. *ACM Transactions on Internet Technology (TOIT)*, 1(1):2--43.

Badge, J. L. & Scott, J. (2009). Dealing with plagiarism in the digital age.

Baeza-Yates, R. & Ribeiro-Neto (2013). *Recuperação de Informação - Conceitos e Tecnologia das Máquinas de Busca*. Bookman.

Baker, B. S. (1992). A program for identifying duplicated code. *Computing Science and Statistics*, pp. 49--49.

Basit, H. A.; Puglisi, S. J.; Smyth, W. F.; Turpin, A. & Jarzabek, S. (2007). Efficient token based clone detection with flexible tokenization. Em *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, pp. 513--516, New York, NY, USA. ACM.

Baxter, I. D.; Yahin, A.; Moura, L.; Sant'Anna, M. & Bier, L. (1998). Clone detection using abstract syntax trees. Em *Software Maintenance, 1998. Proceedings., International Conference on*, pp. 368--377. IEEE.

Burrows, S. & Tahaghoghi, S. M. (2007). Source code authorship attribution using n-grams. Em *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*, pp. 32--39.

Burrows, S.; Tahaghoghi, S. M. M. & Zobel, J. (2007). Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper.*, 37(2):151--175. ISSN 0038-0644.

Bussab, W. d. O. (1990). *Introdução à análise de agrupamentos*. ABE, 1990. 105 p.

Cornic, P. (2008). Detection using model-driven software development in eclipse platform.

Cosma, G. & Joy, M. (2006). Source-code plagiarism: A uk academic perspective.

Croft, W. B.; Metzler, D. & Strohman, T. (2010). *Search engines: Information retrieval in practice*. Addison-Wesley Reading.

Ducasse, S.; Rieger, M. & Demeyer, S. (1999). A language independent approach for detecting duplicated code. Em *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pp. 109–118. ISSN 1063-6773.

Dumais, S. T. (1994). Latent semantic indexing (lsi) and trec-2. Em *The Second Text REtrieval Conference (TREC-2*, pp. 105--115.

Ester, M.; Kriegel, H.-P.; Sander, J.; Xu, X. et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. Em *Kdd*, volume 96, pp. 226--231.

Faidhi, J. A. W. & Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.*, 11(1):11--19. ISSN 0360-1315.

Ferreira, A. (2009). *Mini Aurélio: dicionário eletrônico*. Ed. Positivo.

Gil, Y. R.; Palma, Y. d. C. T. & Lahens, E. F. (2014). Determination of writing styles to detect similarities in digital documents. *Revista de Universidad y Sociedad del Conocimiento*, 11(1):128--141.

Gitchell, D. & Tran, N. (1999). Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266--270. ISSN 0097-8418.

Jain, A. K.; Murty, M. N. & Flynn, P. J. (1999). Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264--323.

Java.net (2014). Java tree builder.

Jiang, L.; Misherghi, G.; Su, Z. & Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. Em *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 96--105, Washington, DC, USA. IEEE Computer Society.

Jing-Bin, L.; Liu, Y. & Bei, H. (2011). Research on parallel k-medoids algorithm based on multi-core platform [j]. *Application Research of Computers*, 2:026.

Johnson, J. H. (1993). Identifying redundancy in source code using fingerprints. Em *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pp. 171--183. IBM Press.

Joy, M. & Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on education*, 42(2):129--133.

Karp, R. M. & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249--260.

Koschke, R.; Falke, R. & Frenzel, P. (2006). Clone detection using abstract syntax suffix trees. Em *2006 13th Working Conference on Reverse Engineering*, pp. 253--262. IEEE.

Levine, J. R.; Mason, T. & Brown, D. (1992). *Lex & yacc*. "O'Reilly Media, Inc.".

Li, J. & Ernst, M. D. (2012). Cbcd: Cloned buggy code detector. Em *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 310--320, Piscataway, NJ, USA. IEEE Press.

Liu, C.; Chen, C.; Han, J. & Yu, P. S. (2006). Gplag: Detection of software plagiarism by program dependence graph analysis. Em *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pp. 872--881, New York, NY, USA. ACM.

Marcus, A. & Maletic, J. (2001). Identification of high-level concept clones in source code. Em *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 107–114. ISSN 1938-4300.

Mariani, L. & Micucci, D. (2012). Audentes: Automatic detection of tentative plagiarism according to a reference solution. *Trans. Comput. Educ.*, 12(1):2:1--2:26. ISSN 1946-6226.

Mayrand, J.; Leblanc, C. & Merlo, E. M. (1996). Experiment on the automatic detection of function clones in a software system using metrics. Em *Software Maintenance 1996, Proceedings., International Conference on*, pp. 244--253. IEEE.

Michaelis (2008). *Dicionário Escolar Língua Portuguesa*. Editora Melhoramentos.

Moraes, R. (2004). Plágio na pesquisa acadêmica: a proliferação da desonestidade intelectual.

Ohmann, A. (2013). Efficient clustering-based plagiarism detection using ippdc.

Page, L.; Brin, S.; Motwani, R. & Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

Palheta, M. (2013). Análise de estilo para atribuição de autoria de programas em cursos de programação.

Palsberg, J. (2004). Java.net the source for java technology collaboration.

Park, C. (2004). Rebels without a clause: towards an institutional framework for dealing with plagiarism by students. *Journal of Further and Higher Education*, 28(3):291–306.

Parr, T. (2007). The definitive antlr reference: Building domain-specific languages. *Pragmatic Programmers (May 2007)*.

Pithan, L. H. & Vidal, T. R. A. (2013). O plágio acadêmico como um problema ético, jurídico e pedagógico. *Direito & Justiça*, 39(1).

Poon, J. Y.; Sugiyama, K.; Tan, Y. F. & Kan, M.-Y. (2012). Instructor-centric source code plagiarism detection and plagiarism corpus. Em *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pp. 122--127, New York, NY, USA. ACM.

Prechelt, L.; Malpohl, G. & Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016.

Robertson, S. (2004). Understanding inverse document frequency: on theoretical arguments for idf. *Journal of Documentation*, 60(5):503–520.

Robertson, S. E. & Jones, K. S. (1976). Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129--146.

Robertson, S. E. & Walker, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. Em *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 232--241. Springer-Verlag New York, Inc.

Robertson, S. E. & Walker, S. (1997). On relevance weights with little relevance information. Em *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '97, pp. 16--24, New York, NY, USA. ACM.

Robertson, S. E. & Walker, S. (1999). Okapi/keenbow at trec-8. Em *TREC*, volume 8, pp. 151--162.

Robertson, S. E.; Walker, S. & Beaulieu, M. (2000). Experimentation as a way of life: Okapi at trec. *Inf. Process. Manage.*, 36(1):95--108. ISSN 0306-4573.

Roy, C. K. & Cordy, J. R. (2007). A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 115.

Salton, G. & Lesk, M. E. (1968). Computer evaluation of indexing and text processing. *Journal of the ACM (JACM)*, 15(1):8--36.

Salton, G. & Yang, C.-S. (1973). On the specification of term values in automatic indexing. *Journal of documentation*, 29(4):351--372.

Satopaa, V.; Albrecht, J.; Irwin, D. & Raghavan, B. (2011). Finding a "kneedle"in a haystack: Detecting knee points in system behavior. Em *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, pp. 166--171, Washington, DC, USA. IEEE Computer Society.

Sheard, J.; Dick, M.; Markham, S.; Macdonald, I. & Walsh, M. (2002). Cheating and plagiarism: Perceptions and practices of first year it students. Em *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '02, pp. 183--187, New York, NY, USA. ACM.

Silva, O. S. F. (2008). Entre o plágio e a autoria: qual o papel da universidade. *Revista Brasileira de Educação*, 13(38):357.

Smith, T. F. & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, (1):195--197.

Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11--21.

Tao, G.; Guowei, D.; Hu, Q. & Baojiang, C. (2013). Improved plagiarism detection algorithm based on abstract syntax tree. Em *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pp. 714–719.

Trotman, A. (2003). Compressing inverted files. *Information Retrieval*, 6(1):5--19.

Đurić, Z. & Gašević, D. (2012). A source code similarity system for plagiarism detection. *The Computer Journal*, p. bxs018.

Vamplew, P. & Dermoudy, J. (2005). An anti-plagiarism editor for software development courses. Em *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05, pp. 83--90, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Vogts, D. (2009). Plagiarising of source code by novice programmers a c̈ry for help?̈ Em *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT 0́9, pp. 141–149, New York, NY, USA. ACM.

Wartik, S. P. (1992). Boolean operations. Em *Information Retrieval: Data Structures & Algorithms*, pp. 264--292.

Whale, G. (1990). Identification of program similarity in large populations. *Comput. J.*, 33(2):140–146. ISSN 0010-4620.

Wise, M. J. (1993). String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119.

Witten, I. H.; Moffat, A. & Bell, T. C. (1999). *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann.

Yuan, Y. & Guo, Y. (2012). Boreas: An accurate and scalable token-based approach to code clone detection. Em *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pp. 286--289, New York, NY, USA. ACM.

Zhang, L. & Liu, D. (2013). Ast-based multi-language plagiarism detection method. pp. 738–742. cited By 0.

Apêndice A

Resultados de Experimentos com o Modelo Vetorial

Conforme explicitado na Seção 5.4.1, neste Apêndice são apresentadas as tabelas com os níveis de precisão e revocação obtidos nos experimentos com o modelo vetorial. A Tabela A.1 apresenta os resultados da análise de plágios naturais e a Tabela A.2 apresenta os resultados da análise de plágios sintéticos.

Tabela A.1: Níveis de precisão (P) e revocação (R) para as variantes de *idf* propostas nesta pesquisa para a análise de plágios naturais com o modelo Vetorial.

	Variante	P	R
1	$idf(t)$	0,904	0,902
2	$idf_a(t,a)$	0,765	0,882
3	$idf_p(t,p)$	0,890	0,853
4	$idf_q(t,q)$	0,857	0,833
5	$idf(t) \times idf_p(t,p)$	0,920	0,856
6	$idf(t) \times idf_q(t,q)$	0,920	0,856
7	$idf_p(t,p) \times idf_q(t,q)$	0,930	0,835
8	$idf(t) \times idf_p(t,p) \times idf_q(t,q)$	0,930	0,841
9	$idf(t) + idf_p(t,p)$	0,931	0,898
10	$idf(t) + idf_q(t,q)$	0,921	0,878
11	$idf_p(t,p) + idf_q(t,q)$	0,870	0,833
12	$idf(t) + idf_p(t,p) + idf_q(t,q)$	0,921	0,878
13	$idf(t) / idf_a(t,a)$	0,745	0,905
14	$idf_p(t,p) / idf_a(t,a)$	0,782	0,849
15	$idf_q(t,q) / idf_a(t,a)$	0,772	0,849
16	$idf(t) \times idf_p(t,p) \times idf_q(t,q) / idf_a(t,a)$	0,817	0,846
17	$idf(t) + idf_p(t,p) + idf_q(t,q) / idf_a(t,a)$	0,782	0,892

Tabela A.2: Níveis de precisão (P) e revocação (R) para as variantes de *idf* propostas nesta pesquisa para a análise de plágios sintéticos com o modelo Vetorial.

	Variante	P	R
1	$idf(t)$	0,960	0,980
2	$idf_a(t,a)$	0,892	0,980
3	$idf_p(t,p)$	0,970	0,980
4	$idf_q(t,q)$	0,970	0,980
5	$idf(t) \times idf_p(t,p)$	0,950	0,960
6	$idf(t) \times idf_q(t,q)$	0,950	0,960
7	$idf_p(t,p) \times idf_q(t,q)$	0,980	0,980
8	$idf(t) \times idf_p(t,p) \times idf_q(t,q)$	0,930	0,940
9	$idf(t) + idf_p(t,p)$	0,960	0,980
10	$idf(t) + idf_q(t,q)$	0,970	0,980
11	$idf_p(t,p) + idf_q(t,q)$	0,970	0,980
12	$idf(t) + idf_p(t,p) + idf_q(t,q)$	0,970	0,980
13	$idf(t) / idf_a(t,a)$	0,823	0,980
14	$idf_p(t,p) / idf_a(t,a)$	0,828	0,980
15	$idf_q(t,q) / idf_a(t,a)$	0,832	0,980
16	$idf(t) \times idf_p(t,p) \times idf_q(t,q) / idf_a(t,a)$	0,882	0,960
17	$idf(t) + idf_p(t,p) + idf_q(t,q) / idf_a(t,a)$	0,809	0,960

Anexo A

Lista de Fragmentos da Linguagem C com seus respectivos símbolos

Tabela A.1: Operadores da Linguagem C que são mantidos no processo de tokenização.

Operador	Nome	Símbolo	Operador	Nome	Símbolo
(	parêntese esquerdo	aa	! =	diferente	wa
)	parêntese direito	ba	&	E *bit* a *bit*	xa
[	colchete esquerdo	ca	∧	XOR *bit* a *bit*	ya
]	colchete direito	da	|	OU *bit* a *bit*	za
− >	acesso indireto a membro	ea	&&	E lógico	ab
.	acesso direto a membro	fa	||	OU lógico	bb
++	incremento	ga	?	condição ternária	cb
−−	decremento	ha	=	atribuição	db
!	negação	ia	+ =	mais igual	eb
∼	complemento	ja	− =	menos igual	fb
∗	multiplicação	ka	∗ =	vezes igual	gb
/	divisão	la	/ =	dividido igual	hb
%	resto da divisão	ma	% =	resto igual	ib
+	adição	na	<<=	deslocamento à esquerda igual	jb
−	subtração	oa	>>=	deslocamento à direita igual	kb
<<	deslocamento à esquerda	pa	& =	E igual	lb
>>	deslocamento à direita	qa	∧ =	XOR igual	mb
<	menor que	ra	| =	OU igual	nb
>	maior que	sa	,	vírgula	ob
<=	menor ou igual a	ta	{	chave esquerda	he
>=	maior ou igual a	ua	}	chave direita	ie
==	igualdade	va			

Fonte: Adaptado de Palheta [2013].

Tabela A.2: Palavras-chave da Linguagem C que são mantidas no processo de tokenização.

Palavra-chave	Símbolo	Palavra-chave	Símbolo	Palavra-chave	Símbolo
auto	pb	do	ac	goto	lc
signed	qb	unsigned	bc	break	mc
double	rb	if	cc	sizeof	nc
void	sb	case	dc	else	oc
int	tb	static	ec	volatile	pc
char	ub	enum	fc	long	qc
struct	vb	while	gc	const	rc
extern	wb	register	hc	switch	sc
continue	xb	float	ic	return	tc
typedef	yb	default	jc	for	uc
short	zb	union	kc		

Fonte: Adaptado de Palheta [2013].

Tabela A.3: Palavras-chave de funções de entrada e saída da Linguagem C que são mantidas no processo de tokenização.

Palavra-chave	Símbolo	Palavra-chave	Símbolo	Palavra-chave	Símbolo
BUFSIZ	vc	FILE	pd	fscanf	jd
_IONBF	wc	rewind	qd	stderr	kd
clearerr	xc	FILENAME_MAX	rd	fseek	ld
L_tmpnam	yc	scanf	sd	stdin	md
EOF	zc	fopen	td	fsetpos	nd
NULL	ad	SEEK_CUR	ud	stdout	od
fclose	bd	FOPEN_MAX	vd	ftell	pd
perror	cd	SEEK_END	wd	tmpfile	qd
feof	dd	fpos_t	xd	fwrite	rd
printf	ed	SEEK_SET	yd	TMP_MAX	sd
ferror	fd	fprintf	zd	getc	td
putc	gd	setbuf	ad	tmpnam	ud
fflush	hd	fputc	bd	getchar	vd
putchar	id	setvbuf	cd	ungetc	wd
fgetc	jd	fputs	dd	gets	xd
puts	kd	size_t	ed	vfprintf	yd
fgetpos	ld	fread	fd	_IOFBF	zd
remove	md	sprintf	gd	vprintf	ae
fgets	nd	freopen	hd	_IOLBF	be
rename	od	sscanf	id	vsprintf	ce

Adaptado de Palheta [2013].

Tabela A.4: Caracteres de espaços em branco que são mantidos no processo de tokenização.

Caractere	Nome	Símbolo	Caractere	Nome	Símbolo
“ ”	espaço em branco	de	“\t”	tab	fe
“\r”	retorno de linha	ee	“\n”	nova linha	ge

Adaptado de Palheta [2013].