UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Luiz Henrique Dias Lima

Estudo de Técnicas de Detecção de Plágio em Código Fonte

Uberlândia, Brasil

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Luiz Henrique Dias Lima

Estudo de Técnicas de Detecção de Plágio em Código Fonte

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: André Ricardo Backes

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2021

Luiz Henrique Dias Lima

Estudo de Técnicas de Detecção de Plágio em Código Fonte

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 3 de novembro de 2021:

André Ricardo Backes
Orientador

Rodrigo Sanches Miani
Professor convidado

Wendel Alexandre Xavier de Melo
Professor convidado

Uberlândia, Brasil
2021

Agradecimentos

Agradeço primeiramente a Deus pela oportunidade de concluir mais essa etapa em minha vida.

Aos meus pais que sempre me apoiaram e auxiliaram nessa caminhada.

Ao professor André Backes por toda atenção, correções e ensinamentos feitos durante o desenvolvimento deste trabalho.

Por fim, gostaria de agradecer aos demais professores da FACOM pela disposição e prazer em ensinar.

"It is not enough for code to work." - Robert C. Martin

Resumo

Em virtude da gravidade e consequências do plágio no ambiente universitário, neste trabalho foram estudadas e combinadas técnicas de detecção de plágio em código em fonte, visando obter resultados eficazes. Partindo da hipótese que a utilização de conceitos de RI seja eficaz na detecção de similaridades entre pares de códigos, foi desenvolvido um *software* nomeado YouPlag. De modo a mensurar a eficácia deste novo sistema de detecção de plágio com os do estado da arte (MOSS e JPlag), três bases de códigos de níveis de dificuldade distintos foram submetidas aos sistemas, através das quais foi possível calcular a precisão e exatidão. Os *softwares* mostraram resultados semelhantes com o limiar de 50%. Além disso, tendo em vista a necessidade de automatizar a escolha do limiar, foi apresentada uma técnica para calculá-lo através do limiar de Otsu, a qual melhorou significativamente a exatidão ao passo que a precisão fora afetada em algumas bases. No geral, observou-se que as técnicas de RI implementadas geraram resultados satisfatórios, principalmente com a ponderação TF-IDF.

Palavras-chave: Recuperação da Informação, plágio, detecção de similaridades

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

MOSS	*Measure of Software Similarity*
RI	Recuperação da Informação
SMs	*Structure Measure*
ATMs	*Attribute Measure*
TF	*Term Frequency*
IDF	*Inverse Document Frequency*

Sumário

1 Introdução

O plágio em disciplinas de programação é um assunto delicado e preocupante. De acordo com Cosma e Joy (2008), essa prática é ainda mais forte em trabalhos que valem mais nota. Com a praticidade da internet, alunos têm uma variedade de sites (e.g. Github e StackOverflow) para copiar códigos que não são de sua autoria e entregá-los sem os devidos créditos.

As formas com as quais estudantes "ocultam" o plágio em códigos variam bastante, desde alterações de nomes de variáveis, nome de funções, inserção ou deleção de comentários, mudança na estrutura e formatação, etc. a utilização de comandos semelhantes (e.g. *for* por *while*, *if* por *switch*, etc.) (Whale, 1990) (Zuhoor et al., 2010). O grande problema é que para disciplinas com muitos alunos matriculados, fica praticamente inviável para o docente analisar código por código manualmente.

Por conta da dificuldade de analisar os trabalhos manualmente, algumas ferramentas de software são utilizadas para encontrar similaridades entre códigos. As mais comuns são: JPlag, executado localmente (Prechelt; Malpohl; Philippsen, 2003); e MOSS (*Measure of Software Similarity*), sistema online mantido pela universidade de Stanford (Sanders, 1997). Além de ferramentas desenvolvidas em artigos (Wise, 1992). É de extrema importância salientar que nenhum algoritmo de identificação de plágio afirma se houve ou não o ato, apenas classifica códigos como possíveis cópias a partir de um limiar T (Whale, 1990).

Algoritmos para identificação de plágio podem ser classificados em: ATMs (*Attribute Measure*), em que os atributos dos códigos são quantificados, tais como operadores, operandos, quantidade total de operadores e quantidade total de operandos (Ottenstein, 1976), através de alguma fórmula matemática na qual a similaridade é calculada (Zuhoor et al., 2010); SMs (*Structure Measure*), em que a estrutura do código é analisada através da comparação entre *streams* de *tokens* para encontrar segmentos comuns (Prechelt; Malpohl; Philippsen, 2003).

Ainda no ambiente universitário, é muito comum que professores permitam o reuso de códigos utilizados em sala de aula ou discutidos em laboratório, além de exemplos providos em livros. Portanto, é necessário que a ferramenta escolhida pelo docente seja capaz de excluir o código permitido dos arquivos enviados (Kermek; MATIJA, 2016) ou permita a definição de um *threshold* (limiar) de similaridade.

A definição de um *threshold* permite ao usuário ajustar seu valor de acordo com a dificuldade de solução do problema. Como apontado por (Whale, 1990), se o problema apresentado permite várias soluções, então a probabilidade de coincidência entre dois

pares é consideravelmente reduzida.

1.1 Objetivos

O objetivo deste trabalho é estudar e combinar técnicas já existentes de identificação de plágio para definir similaridades entre códigos fontes, de modo a desenvolver uma ferramenta capaz de identificar possíveis cópias entre códigos fontes escritos em C. Partindo da hipótese que técnicas de Recuperação da Informação, RI, sejam capazes de identificar semelhanças entre arquivos de código.

Os objetivos específicos deste trabalho são:

- Desenvolver um algoritmo para gerar *streams* de *tokens*;
- Investigar o método mais adequado para ponderar termos contidos em um documento de código;
- Investigar a eficácia do modelo vetorial e coeficiente de Dice na identificação de similaridades entre códigos fontes;
- Empregar o limiar de Otsu como maneira de separar possíveis cópias de pares de códigos independentes;
- Comparar os resultados do YouPlag com os obtidos pelo MOSS e JPlag.

1.2 Justificativa

Um dos principais problemas do plágio em códigos na faculdade é que, por meio dele, muitos estudantes não aprendem a programar de fato. Por consequência, estes vão para o mercado de trabalho apenas com a qualificação (diploma), mas sem o conhecimento necessário para resolução de problemas e demandas do mercado de trabalho, podendo ainda manchar a imagem da universidade (Lancaster, 2003).

Por meio de questionários na pesquisa feita por (Chuda et al., 2012) na Universidade de Informática e Tecnologia da Informação (FIT), Bratislava, Eslováquia, cerca de 80% dos docentes revelaram que utilizam a experiência como forma de analisar os códigos enviados pelos alunos, o que acaba sendo um grande problema para turmas com muitos alunos; pois, muitas cópias de programas podem ser ignoradas, fazendo com que alguns discentes sempre busquem formas de trapacear.

Portanto, neste trabalho será estudado como detectar similaridades entre códigos fontes e foi desenvolvido um *software*, YouPlag, capaz de informar quais são os pares cujos níveis de similaridade indicam possíveis plágios. Desse modo, boa parte do trabalho

manual de análise de códigos pelo docente poderá ser reduzida e possíveis pares similares serão facilmente identificados.

1.3 Organização do trabalho

No capítulo 2 é apresentado o referencial bibliográfico, abordando temas relacionados à Recuperação da Informação (RI) e elementos necessários à detecção de plágio em código fonte. No capítulo 3, o estado da arte é descrito, e, logo em seguida, no capítulo 4, os trabalhos correlatos são apresentados. O capítulo 5 descreve como o YouPlag fora desenvolvido. Por fim, no capítulo 6, os resultados do YouPlag são comparados com os obtidos pelo JPlag e MOSS.

2 Referencial Bibliográfico

Este capítulo apresenta os principais conceitos para a realização deste trabalho.

2.1 Recuperação da Informação

De acordo com Gerard Salton, recuperação da informação (RI) é: "[...] uma área de pesquisa preocupada com a estrutura, análise, organização, armazenamento, busca e recuperação da informação" (Croft; Metzler; Strohman, 2015). Um dos grandes desafios de RI é encontrar documentos que sejam relevantes para o usuário (Croft; Metzler; Strohman, 2015). O simples ato de comparar dois arquivos de textos diretamente pode implicar em resultados insatisfatórios aos usuários; pois, duas sequências de caracteres podem ser distintas, mas podem expressar o mesmo significado.

Portanto, para mitigar resultados irrelevantes, pesquisadores definem modelos de recuperação, tais como o modelo vetorial e o BM25, e testam sua eficácia através do ranking de documentos obtido pelos algoritmos (Croft; Metzler; Strohman, 2015).

Algo importante a ser observado é que esses algoritmos frequentemente utilizam conceitos estatísticos ao invés de análises puramente textuais (Croft; Metzler; Strohman, 2015), por exemplo o modelo vetorial utiliza o ângulo obtido entre o documento e a consulta (*query*) para definir o grau de similaridade (Prabhakaran, 2020).

Um sistema de RI não seria completo sem um motor de busca (*search engine*). De acordo com Croft, Metzler e Strohman (2015), esse elemento é definido como: "[...] *software* que compara consultas com documentos e produz listas de documentos ranqueados ". O modo pelo qual um motor de busca é implementado varia de acordo com a aplicação, por exemplo para um *software* que busca similaridades entre códigos fontes no ambiente universitário, é desejável que este seja capaz de processar muitos documentos e utilizar as nuances de cada linguagem de programação para compará-los eficazmente.

2.1.1 Term Frequency - TF

Componente utilizado para mensurar a importância de um termo em um documento D. É importante salientar que um documento é visto como um "saco de palavras" (*bag of words*) (Manning; Raghavan; Schütze , 2009). Assim sendo, o cálculo do peso associado a um termo não reflete a ordem em que este se encontra nos documentos da coleção. Em alguns casos quanto mais um termo se repete em um documento, mais representativo ele pode ser para o documento (Manning; Raghavan; Schütze , 2009). Também

é possível amenizar a influência da frequência do termo através do logaritmo. Assim, a seguinte equação é obtida:

$$w_{ij} = \begin{cases} 1 + \log f_{ij}, & \text{se } f_{ij} \geq 1 \\ 0, & \text{caso contrário} \end{cases} \tag{2.1}$$

Onde w representa o peso do i^{th} termo no j^{th} documento, e f_{ij} representa a frequência do termo.

2.1.2 Inverse Document Frequency - IDF

Diferentemente do TF, cujo objetivo é calcular o quão relevante um termo é para um documento, o IDF busca quantificar o quão importante é um termo dentro de uma coleção de documentos. Heuristicamente, quanto mais documentos em uma coleção possuírem um determinado termo, menor é sua importância para a recuperação da informação (Robertson, 2004). Logo, deve ser penalizado com um peso menor. Ao passo que quanto mais raro um termo é, maior é seu peso. Assim sendo, a seguinte fórmula é obtida:

$$IDF_i = \log \frac{N}{n_i} \tag{2.2}$$

Onde N representa o número total de documentos contidos na coleção e n_i representa o número de documentos que possuem um determinado termo.

2.1.3 TF-IDF

Os esquemas de ponderação TF-IDF consistem em multiplicar os pesos obtidos pelas operações TF e IDF. De acordo com Stephen Robertson, este modelo de ponderação é muito robusto e eficaz (Robertson, 2004). Em alguns casos, porém, não é interessante tornar a frequência do termo em um documento menos acentuada. Por conseguinte, quanto mais um termo se repete em um documento, melhor ele descreve seu conteúdo (Manning; Raghavan; Schütze , 2009); logo não deve ser "penalizado" com logaritmo. A seguinte equação é obtida:

$$\text{tf-idf}_{i,j} = \begin{cases} f_{ij} \times idf(t_i), & \text{se } \text{f}_{ij} \geq 1 \\ 0, & \text{caso contrário} \end{cases} \tag{2.3}$$

Dependendo da situação é interessante amenizar o impacto da frequência do termo. Essa técnica parte do seguinte princípio: quanto maior for um documento, maior será a probabilidade de repetição dos termos (Burrows; Tahaghoghi; Zobel, 2007). Supõe-se que o número de repetições muito elevado não significa que tal termo seja tão importante para

o documento (Manning; Raghavan; Schütze , 2009). Desse modo, a ponderação é dada por:

$$\text{wf-idf}_{i,j} = \begin{cases} (1 + \log f_{ij}) \times idf(t_i), & \text{se } \mathrm{f}_{ij} \geq 1 \\ 0, & \text{caso contrário} \end{cases} \tag{2.4}$$

Manning, Raghavan e Schütze (2009) definiram esta técnica como escala TF sublinear. Neste trabalho, ela será caracterizada pela sigla WF-IDF.

2.1.4 Modelo Vetorial

O modelo vetorial é uma das técnicas mais utilizadas em RI para calcular a similaridade entre dois documentos, formalmente entre uma *query* q e um documento d. De acordo com (Croft; Metzler; Strohman, 2015), os principais benefícios desse modelo são: simplicidade na implementação da ponderação de termos, ranqueamento, e feedback de relevância (neste trabalho este conceito não será abordado).

No modelo vetorial, os documentos são tratados em um espaço vetorial com t dimensões, onde t representa o número de termos contidos em um documento d e uma *query* q (Croft; Metzler; Strohman, 2015). Geralmente, cada ponto representa um peso atribuído a um termo em d e q. A Figura 1 exemplifica um espaço tridimensional.

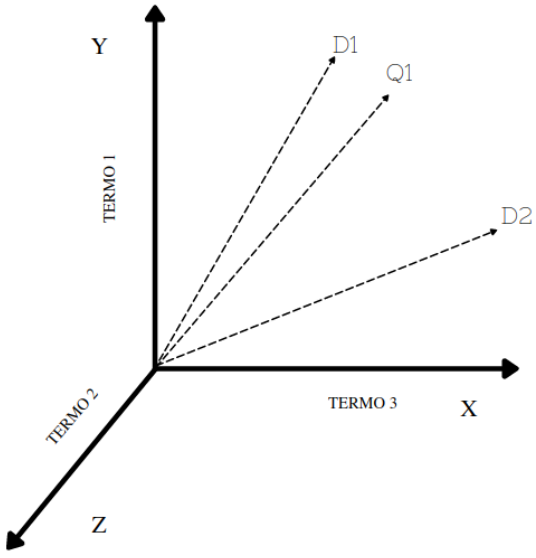


Figura 1 – Representação de um espaço tridimensional. Onde as coordenadas x, y, e z representam os termos contidos nos documentos e na *query*.

Fonte: O autor

Uma das possíveis formas de calcular a similaridade entre dois documentos é através da diferença da magnitude de seus vetores. Porém, se dois documentos, compartilhando

alguns termos em comum, possuírem tamanhos muito discrepantes, então o resultado final não será tão satisfatório (Manning; Raghavan; Schütze , 2009).

Assim, uma das formas de calcular a similaridade eficazmente é através da similaridade por cosseno. Quanto menor o ângulo entre os vetores que representam d e q, maior será a similaridade entre eles. Ao passo que quanto maior for o ângulo, mais dissimilares o documento e a *query* são (Croft; Metzler; Strohman, 2015). A equação abaixo apresenta como o grau de semelhança pode ser obtido:

$$\cos(D_i, Q) = \frac{\sum_{j=1}^{t} d_{ij} \times q_j}{\sqrt{\sum_{j=1}^{t} d_{ij}^2 \times \sum_{j=1}^{t} q_{ij}^2}} \tag{2.5}$$

Onde D_i representa o i^{th} documento na coleção e Q representa a *query*.

2.2 Coeficiente de Dice

Método estatístico que busca definir a semelhança entre dois documentos através da interseção de seus conjuntos (um conjunto pode ser representado pelos termos ou vetor de pesos). Pode ser definido, de acordo com Stefanovič, Kurasova e Štrimaitis (2019), da seguinte maneira:

$$dice(D1, D2) = 2\frac{D1 \times D2}{\sum_{j=1}^{t} D_1{}_j^2 \times \sum_{j=1}^{t} D_2{}_j^2} \tag{2.6}$$

Onde D1 e D2 representam um par de documentos presentes em uma coleção.

2.3 Tokenização

O processo de tokenização consiste em modificar a estrutura de um texto, de acordo com as necessidades da aplicação (Croft; Metzler; Strohman, 2015). Por conta disso, seria possível excluir ou modificar as seguintes características de um arquivo de programa: espaços em branco, comentários, letras maiúsculas, pontuação, nome de variáveis, etc. (Schleimer; Wilkerson; Aiken, 2003). Desse modo, a comparação entre um par de documentos não seria influenciada por esses fatores.

Em alguns algortimos estudados neste trabalho, a tokenização também reduz possíveis disfarces de plágio, como a utilização de comandos semelhantes (Lancaster, 2003). De acordo com Whale (1990), esta representação faz com que a seletividade (i.e. capacidade de ignorar diferenças) de um sistema de identificação de similaridades aumente.

A Tabela 1 exemplifica o processo de tokenização com classes de equivalência (Manning; Raghavan; Schütze , 2009).

Tabela 1 – Exemplo de tokenização para linguagem C.

Keyword	*Token*
break	STMT
if, switch	COND
for, while	LOOP

2.4 Índices

Para lidar com muitas informações de muitos documentos contidos em uma coleção, é necessária a utilização de uma estrtura de dados eficiente. O uso de *arrays* para armazenar dados a respeito de termos e suas respectivas ocorrências não satisfaz esse critério. Ao contrário da tabela *hash*, a qual é mais eficiente tanto na busca quanto na inserção.

No índice invertido, cada termo, devidamente tokenizado, aponta para uma lista que possui informações a respeito dos documentos em que ocorrera e sua respectiva frequência — eis o porquê do nome. Uma das possíveis estruturas para representar o índice invertido é tabela *hash* (Croft; Metzler; Strohman, 2015), cujas chaves são os termos e os valores podem ser representados por listas.

A Figura 2 apresenta um exemplo de índice invertido. Programaticamente, cada item da lista pode ser representado por uma tupla contendo o *id* do documento em que o termo ocorreu e sua frequência (Manning; Raghavan; Schütze , 2009).

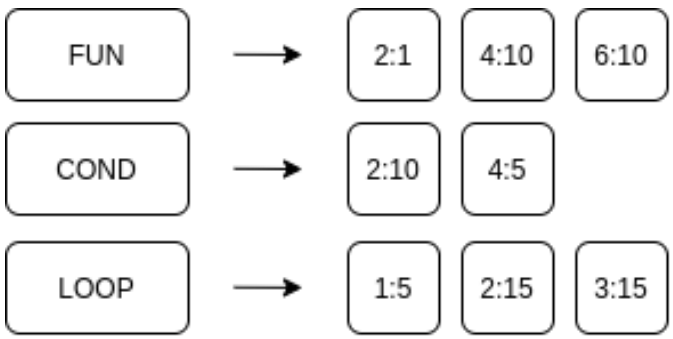


Figura 2 – Exemplo de índice invertido após o processo de tokenização.

Fonte: O autor

2.5 N-grams

No contexto de detecção de plágio em código fonte, algumas técnicas fazem o uso de *n-grams* para encontrar sequências comuns a dois documentos. *N-grams* é uma *substring* contígua de N caracteres, o valor de N pode ser definido pelo usuário (Schleimer; Wilkerson; Aiken, 2003).

Em se tratando de identificação de similaridade entre dois arquivos, um valor de N bem definido pode eliminar ruídos do texto (palavras que são muitos comuns, e podem atrapalhar o resultado final obtido pelo algoritmo) (Schleimer; Wilkerson; Aiken, 2003). Ao passo que um valor relativamente pequeno, tende a gerar mais falsos positivos. Definidos os *n-grams*, alguns algoritmos fazem o *hash* das *substrings* obtidas, e então selecionam valores de *hash* que são $0 \ mod \ p$, onde p é um valor fixo. Essa técnica é denominada *fingerprinting* (Schleimer; Wilkerson; Aiken, 2003).

2.6 Limiar de Otsu

Comumente utilizado em segmentação de imagens, o limiar de Otsu gera automaticamente um limiar capaz segmentar uma imagem em dois grupos (*background* e *foreground*) (Murzova; Seth, 2020). De acordo com Murzova e Seth (2020), os principais passos para obtenção desse limiar T são:

1. Processamento da imagem;
2. Obtenção do histograma;
3. Cálculo do limiar;
4. Substituição dos *pixels* maiores que T em branco; preto, caso contrário.

Neste trabalho, como não foram utilizadas imagens, o item 1 se deu através do cálculo da distribuição dos níveis de similaridades entre os códigos. Já o item 4, serviu para, a partir de T, separar as possíveis cópias de pares independentes.

O método pelo qual o limiar de Otsu escolhe T se dá através da minimização da variância interna das classes do histograma (Murzova; Seth, 2020). Também pode-se, de acordo com Greensted (2010), maximizar a variância entre classes. Assim sendo, segundo Greensted (2010), a variância entre classes é dada por:

$$\sigma_B^2 = W_b \times W_f(u_b - u_f)^2 \tag{2.7}$$

Onde σ_B^2 representa a variância entre classes; W_b, a média do *background* (valores menores que T); W_f, a média do *foreground* (valores maiores ou iguais a T); por fim, u_b e u_f representam a média ponderada do *background* e *foreground* respectivamente.

3 Estado da Arte

Esta seção tem como objetivo apresentar o estado arte, bem como os principais *softwares* de detecção de plágio. Com exceção do YAP3, todos os outros estão disponíveis para uso gratuito.

3.1 YAP3

O sistema YAP3, desenvolvido por Michael J. Wise, é capaz de identificar similaridades tanto em código fonte quanto em arquivos de texto (Wise, 1996). Este software é classificado como SMs, e, de acordo com Michael, o algoritmo que realiza a comparação entre as *streams* de *tokens* é o *Running-Karp-Rabin Greedy-String-Tiling* (RKR-GST) (Wise, 1996).

A formulação desse algoritmo surgiu a partir da observação de que alunos, no ato de disfarçar o plágio, buscam também mudar a estrutura do código, e.g. mudar a ordem de chamada das funções. Consequentemente, algoritmos cujo objetivo é encontrar o máxima subsequência comum possível (LCS), poderiam ser facilmente "enganados" por mudanças bruscas na estrutura do código (Wise, 1996).

Basicamente, o algoritmo é dividido em duas fases. A primeira, definida como *scanpattern* (escaneamento de padrões), busca encontrar todos os pares entre uma substring do padrão P e uma substring do texto T cujo tamanho seja igual ou maior a um mínimo valor definido (denominado *minimum match length*) (Wise, 1996) . É nesta etapa que o algoritmo *Running Karp Rabin* é executado: todas as substring de P de tamanho s (definido pelo autor como *search-length*) passam por uma função *hash*, o mesmo vale para T. Quando os valores de *hash* encontrados para T e P são iguais, então são provavelmente similares, e uma comparação é feita em cada elemento de T e P. Dado o resultado do *scanpattern*, a fase de *markstrings* é executada a fim de marcar as *substrings*, mas antes o algoritmo verifica se uma *substring* possui algum *token* que fora marcado anteriormente. Caso não exista, esta é então marcada (o autor do YAP3 definiu esta marcação como *tile*) (Wise, 1996).

A similaridade é obtida através do número de *tiles* obtidos em um par de documentos pelo total de tokens contidos em ambos os arquivos (Prechelt; Malpohl; Philippsen, 2003).

3.2 JPlag

Desenvolvido primeiramente por (Prechelt; Malpohl; Philippsen, 2003), o JPlag realiza comparações par-a-par entre documentos de códigos a fim de encontrar regiões similares. Atualmente, o *software* é mantido pela comunidade no *GitHub*. Inicialmente, o programa suportava apenas as linguages Java, C/C++, e Scheme. Hoje, porém, além dessas linguagens, ele também suporta Python 3, C#, CPP, e arquivos de texto.

O resultado é exibido em uma página HTML, onde o usuário tem a possibilidade de visualizar, a partir de um histograma, a distribuição de similaridade entre os códigos. Além do histograma, também é possível visualizar intuitivamente quais foram as regiões similares apontadas pelo *software*, vide Figura 3.

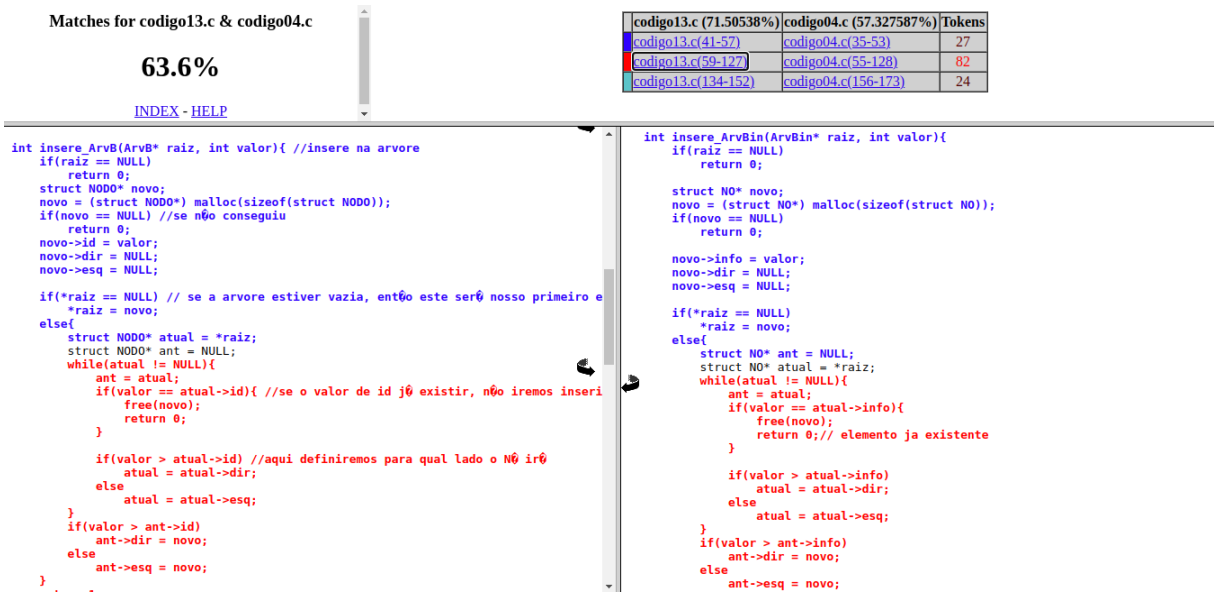


Figura 3 – Comparação par-a-par entre 2 arquivos de códigos. Regiões similares são representadas pelas mesmas cores.

Fonte: O autor

Diferentemente de técnicas do tipo ATMs, o JPlag compara as estruturas do programas para calcular a similaridade entre dois arquivos de código fonte (Prechelt; Malpohl; Philippsen, 2003). O algoritmo utilizado neste *software* é basicamente o mesmo que fora utilizado por (Wise, 1996), com pequenas alterações de modo a melhorar a performance.

De acordo com Prechelt, Malpohl e Philippsen (2003), o algoritmo do JPlag é executado em duas fases:

- Os códigos fontes são convertidos em *tokens*;
- O algoritmo *Greedy String Tiling* é executado para identificar regiões similares e definir o grau de semelhança.

A fase de comparação entre os *tokens* é realizada em dois passos: primeiro, busca-se todas as *strings* comuns entre dois documentos, A e B, a cada *match* o algoritmo almeja estender o máximo possível a correspondência entre as *substrings* de A e B — os valores das substrings dos documentos são obtidos através de uma função *hash*. O uso de *hashes* foi utilizado como forma de otimizar a complexidade do algoritmo *Greedy String Tiling*, inicialmente $\theta((|A|+|B|)^3)$ (Prechelt; Malpohl; Philippsen, 2003). Para tanto, foi utilizado o algoritmo de Rabin-Karp. Dado o objetivo de melhorar a performance do algoritmo nesta fase, uma tabela *hash* é utilizada para armazenar os valores de *hash* das *substrings* de B que possuam correspondências com as de A (Prechelt; Malpohl; Philippsen, 2003). Porém, mesmo com essa otimização, o pior caso (quando dois *tokens* possuem o mesmo valor de *hash*), de acordo com (Prechelt; Malpohl; Philippsen, 2003), continua sendo $\theta((|A|+|B|)^3)$.

Na segunda etapa, com o conjunto de correspondências definido na primeira iteração, o algoritmo busca marcar todas as substrings que não foram sobrepostas anteriormente (Prechelt; Malpohl; Philippsen, 2003). Essa marcação é definida como um *tile*. De acordo com (Prechelt; Malpohl; Philippsen, 2003), o cálculo de similaridade é dado pela seguinte fórmula:

$$sim(A, B) = \frac{2 \times coverage(tile)}{(|A| + |B|)} \tag{3.1}$$

A ferramenta também permite definir o valor do tamanho mínimo de *match* entre dois arquivos. Por padrão este valor é igual a 12. Quanto menor for este valor, maior é a probabilidade de falsos positivos (Prechelt; Malpohl; Philippsen, 2003).

3.3 MOSS

O sistema de detecção de similaridades MOSS, acrônimo para *Measure of Software Similarity*, funciona de modo análogo ao JPlag: o usuário envia uma coleção de programas, a qual pode conter códigos cujo uso fora permitido pelo docente, e o serviço retorna uma página HTML sumarizando os pares semelhantes.

O algoritmo que realiza a comparação entre os *tokens* de dois arquivos é o *winnowing* robusto, cuja ideia é: pegar valores mínimos de *hash* que foram previamente selecionados em uma janela w antecessora à atual. Caso tal seleção não seja possível, opta-se por escolher o menor valor à direita da janela atual (Schleimer; Wilkerson; Aiken, 2003). Cada *fingerprint* guarda informação à respeito da posição da *substring* no documento. Tal dado serve para, posteriormente, mostrar ao utilizador do sistema quais foram as linhas supostamente plagiadas, vide Figura 4.

O uso de *hashes* garante mais flexibilidade ao programa. Tendo em vista um tokenizador específico para cada linguagem suportada, o processo de comparação não sofreria alterações em seu funcionamento (Schleimer; Wilkerson; Aiken, 2003).

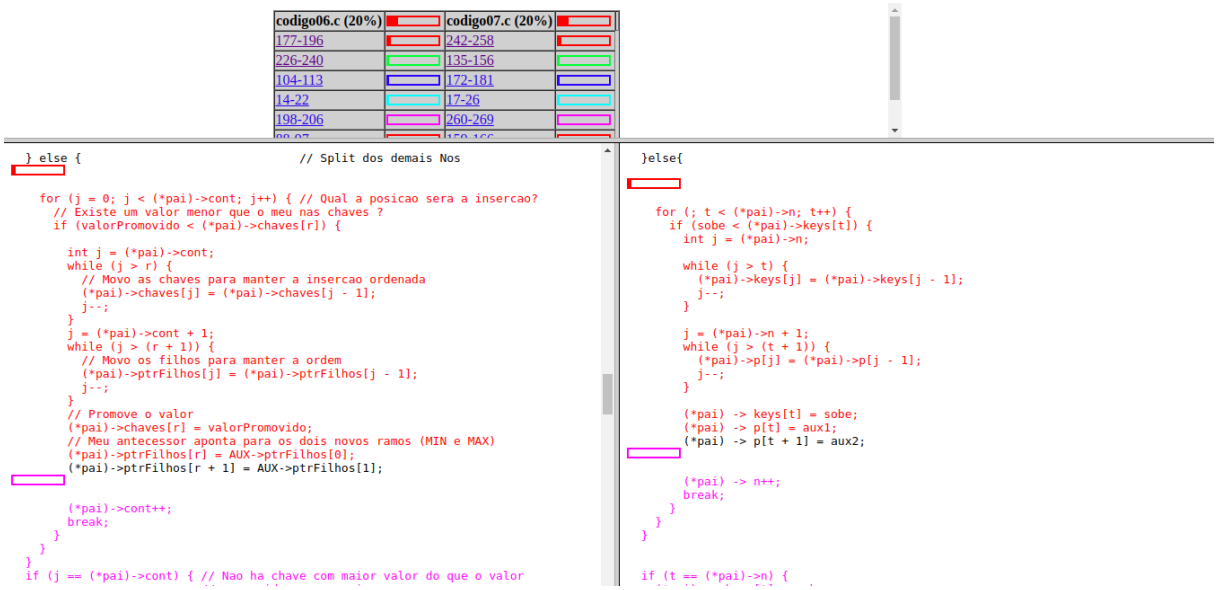


Figura 4 – Exibição de linhas similares.

Fonte: O autor

4 Trabalhos Correlatos

O trabalho de Zuhoor et al. (2010) teve como foco comprovar que o uso da técnica de ATMs é efetivo para encontrar possíveis plágios em programas cujo nível de dificuldade é baixo. Os atributos observados em cada programa foram: palavras-chaves únicas, métodos únicos, interfaces únicas, construtores únicos, número total de palavras-chaves, número total de métodos, número total de interfaces, número total de construtores, número total de instruções, e número total de atribuições (Zuhoor et al., 2010).

Para escolher o método estatístico mais adequado para calcular a similaridade entre pares de documentos, os autores testaram dois esquemas de cáclulo diferentes: razão da equivalência, cuja semelhança entre dois documentos é calculada pela divisão do número de equivalências do par pela média de tokens do par (Zuhoor et al., 2010); coeficiente de correlação, cujo objetivo é calcular o quão relacionadas duas variáveis são; o resultado varia em uma escala de -1 a 1. Utilizando uma base de 17 programas, concluiu-se que o método estatístico coeficiente de correlação produziu muitos falsos negativos e não gerou resultados satisfatórios para cópias parciais (Zuhoor et al., 2010). Logo, a razão da equivalência foi utilizada.

O trabalho de Ottenstein (1976) teve como objetivo definir uma função capaz de determinar a similaridade entre um par de documentos. Partindo da observação que quase toda linguagem de programação possui 3 principais atributos: operadores, operandos, e símbolos de estilização (e.g. comentários), então uma função que levasse em conta esses atributos poderia identificar pares similares satisfatoriamente.

Ottenstein (1976), a fim de comprovar a eficácia dessa função, escreveu um programa para identificar similaridades em códigos escritos em FORTRAN (Ottenstein, 1976). Nos programas testados foram levados em consideração os seguintes atributos: número de operadores únicos, número de operandos únicos, número total de operadores, e número total de operandos (Ottenstein, 1976).

De acordo com o autor, este método não seria facilmente manipulado por mudanças comésticas na estrutura do código. Porém, para cópias parciais o algoritmo não seria capaz de retornar resultados bons (Ottenstein, 1976).

O trabalho desenvolvido Burrows, Tahaghoghi e Zobel (2007) utilizou a função de ranqueamento *Okapi BM25* como fase inicial para encontrar possíveis plágios em grandes bases de dados. Com o resultado obtido, os documentos cujas similaridades fossem maiores do que 30% teriam suas similaridades "refinadas" através do alinhamento local de sequências (Burrows; Tahaghoghi; Zobel, 2007).

A fim de testar a precisão e a escabilidade desse algoritmo, foi utilizada uma base contendo 300 programas. Burrows, Tahaghoghi e Zobel (2007) identificaram que a utilização do método de alinhamento múltiplo de sequências produziu resultados ainda melhores.

5 Desenvolvimento

Um dos objetivos desse trabalho é desenvolver um algoritmo capaz de identificar semelhanças entre códigos fontes através de técnicas de RI. Este capítulo, portanto, tem por objetivo explicar os principais componentes do YouPlag para detectar plágio entre arquivos fontes. Esta ferramenta foi desenvolvida em Java, versão 11. Como foram utilizados conceitos de RI, o YouPlag é classificado como *attribute measure*.

5.1 Base de Dados

Para verificar a eficácia dos algoritmos de detecção de plágio em código fonte, foram utilizadas três bases de dados, cedidas por um professor da Faculdade de Computação da Universidade Federal de Uberlândia (FACOM/UFU), contendo arquivos em linguagem C:

- Base 1: *Skiplist*
  - Objetivo: Implementação de operações em *skiplist*;
  - Nível: Intermediário;
  - Quantidade: 14 códigos;
  - Plágios identificados manualmente: 3.
- Base 2: Árvore B
  - Objetivo: Implementação de operações em árvore B;
  - Nível: Avançado;
  - Quantidade: 15 códigos;
  - Plágios identificados manualmente: 1.
- Base 3: Operações matemáticas
  - Objetivo: Manipulação e exibição de resultados de operações matemáticas baseadas nos *inputs* do usuário;
  - Nível: Iniciante;
  - Quantidade: 12 códigos;
  - Plágios identificados manualmente: 3.

5.2 Métricas Avaliadas

As bases citadas foram submetidas aos *softwares* mencionados anteriormente. O MOSS e o JPlag analisam as estruturas dos códigos, i.e., SMs. Já o *YouPlag*, define a similaridade a partir dos atributos de cada arquivo, i.e., ATMs.

De modo a mensurar a eficácia obtida pelos *softwares*, foi montada uma lista, criada manualmente por um docente, contendo os plágios em cada coleção de fontes. Com isso, foi possível calcular a precisão (proporção de documentos relevantes (Manning; Raghavan; Schütze , 2009)) e exatidão (proporção de documentos relevantes identificados corretamente (Manning; Raghavan; Schütze , 2009)).

5.3 Análise dos Resultados

Para a coleta dos dados, o ambiente utilizado foi o Ubuntu 18.04.4 LTS, distribuição gratuita do Linux. O processo de análise dos dados é descrito abaixo:

- Por questões de segurança e privacidade, os nomes de alunos contidos nos cabeçalhos de alguns arquivos foram removidos;
- Submissão das bases para os *softwares* descritos;
- Cálculo da precisão e exatidão;
- Interpretação e avaliação dos resultados obtidos.

5.4 Tokenização

Similarmente ao texto escrito, linguagens de programação também possuem suas próprias sintaxes, comandos sinônimos (e.g. *for* e *while*), símbolos de pontuação, espaços em branco, indentação, etc. Por conta dessas características, construir o índice invertido sem antes converter o conteúdo do código fonte para um padrão poderá gerar resultados insatisfatórios.

A tokenização, portanto, se faz necessária para representar as principais características de uma linguagem de programação (Burrows; Tahaghoghi; Zobel, 2007). Por exemplo, controles de estrutura são importantes, logo devem ser mantidos. Já comentários, não devem, neste caso, ser analisados; portanto, são descartados. O mesmo vale para elementos que podem não constituir em plágio, e.g. a diretiva *#include*.

Primeiramente, o conteúdo de cada arquivo foi convertido para letras minúsculas, logo após, através do uso de expressões regulares (REGEX), o processo de tokenização se deu da seguinte maneira:

- Remoção da diretiva *#include* para evitar o surgimento de muitos falsos positivos. Já a diretiva *#define* foi mantida e convertida para seu respectivo token;
- Tokenização de criação de *structs*;
- Tokenização de dígitos;
- Tokenização de estruturas condicionais;
- Tokenização de laços de repetição;
- Tokenização de criação de funções;
- Tokenização de operadores relacionais;
- Tokenização de operadores lógicos;
- Tokenização de funções embutidas da linguagem C (e.g. *free* e *printf*);
- Tokenização de chamadas de funções;
- Tokenização de atribuições de variáveis;
- Tokenização de instruções de salto (e.g. return, break, etc.);
- Tokenização de operadores aritméticos;
- Tokenização de chaves e parêntesis.

Terminada a conversão do conteúdo presente no arquivo fonte, são criados *n-grams* de tamanho 4 (i.e. quadrigrams). Este valor mostrou-se eficaz nas bases de códigos analisadas. A Tabela 2 apresenta como cada elemento do excerto de código da Figura 5 foi tokenizado.

Tabela 2 – *Tokens* obtidos a partir do excerto do código 1.

Elemento	*Token*
#define MAXN 5	Y
geraNivel()	F
{	W
int lvl =	A
1	D
+	O
rand()	Ç
%	V
return	R
}	Z

Código 1: Código original retirado da base.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #define MAXN 5
4.
5. int geraNivel(){
6.         int lvl = 1+rand()%MAXN;
7.         return lvl;
8. }
```

Tokens:
YFWADOÇVCRZ

4-grams:
YFWA FWAD WADO ADOÇ DOÇV OÇVC ÇVCR VCRZ

Figura 5 – Representação de *4-grams* de um trecho de código.

Fonte: O autor

5.5 Índice Invertido

Após a criação dos *n-grams* para cada um dos documentos presentes na coleção, a fase de criação do índice invertido é iniciada. Por conta da praticidade e alta performance (tempo constante para inserções e buscas), uma tabela *hash* foi utilizada para armazenar as informações dos termos.

Assim sendo, cada chave representa um termo t na coleção e a ela está associado outra tabela *hash*, cuja chave representa o documento que contém t e a frequência do termo dentro dele. Posteriormente, este valor será utilizado para ponderar o termo.

5.6 Ponderação de termos

Com o índice invertido devidamente preenchido, a fase de ponderação de termos busca definir o quão importante é um termo. Para calcular o peso foram utilizadas as duas técnicas abordadas na seção 2.1.3, i.e., TF-IDF e WF-IDF.

5.7 Consulta

Com o resultado obtido da ponderação dos termos, os documentos da base são pareados de modo a mensurar o grau de unicidade entre eles (quanto mais perto de 0, maior é probabilidade do arquivo não ser cópia do outro). O número possíveis de pares é dado por: *n*(n-1)/2*, em que n representa o número de códigos fontes submetidos ao *software*. Devido às características do processo de consulta em RI, é importante salientar que o grau de semelhança entre dois documentos é simétrico, ou seja, se D1 possui 90% de semelhança com D2, o inverso também é verdade.

São duas as formas pelas quais o sistema pode calcular a similaridade entre os pares: similaridade por cosseno e coeficiente de Dice. Especificada a métrica para calcular a simililaridade, apenas os resultados maiores ou iguais a um limiar T são exibidos ao usuário em ordem descendente. Juntamente aos resultados que satisfazem o critério de similaridade, são apresentados o número de *tokens* em cada documento do par. Para suprimir o problema da simetria entre D1 e D2, a inclusão, i.e. quantificação de quantos porcentos do conteúdo de D1 está contido em D2, é calculada e exibida, vide Figura 6.

```
(fonte09, fonte11) -> 64.52%
fonte09 : 299 tokens
fonte11 : 319 tokens
Containment of fonte09 in fonte11 : 83.88%
Containment of fonte11 in fonte09 : 76.1%
-----------------------------------------------------------------
```

Figura 6 – Nível de similaridade entre um par de documentos.

Fonte: O autor

5.8 Limiarização

Devido aos diferentes níveis de dificuldade dos exercícios passados aos discentes, nem sempre um valor fixo de limiar (*threshold*) trará resultados relevantes ao usuário.

Levando em conta a necessidade de automatizar esse processo, o limiar de Otsu (GREENSTED, 2010), comumente utilizado em segmentação de imagens, foi implementado como forma de melhorar a seletividade do *YouPlag*, i.e. evitar ao máximo falso negativos e diminuir falsos positivos.

Como o grau de semelhança entre códigos está no intervalo de 0% a 100%, as porcentagens de similaridade entre todos os pares foram dispostas em um histograma. A partir do qual foi possível calcular o limiar de Otsu.

5.9 Parâmetros

Para dar mais versatilidade ao usuário, o *YouPlag* provê alguns parâmetros customizáveis. A Tabela 3 apresenta todas as configurações possíveis do sistema.

Tabela 3 – Lista de parâmetros.

Nome	Parâmetro	Padrão
Especificação do diretório	-d	N/A
Linguagem	-l	c
Ponderação	-tfidf ou -wfidf	-tfidf
Métrica de similaridade	-dice ou -cosine	-dice
Limiar	-t	Limiar de Otsu
Programas	-p	N/A

Para o correto funcionamento do YouPlag, é necessária a definição do diretório onde os códigos fontes estão localizados.

De modo a obter o nível de similaridade entre os fontes, é essencial que a linguagem seja definida para que suas classes de equivalência e *tokens* sejam criados respeitando as nuances da linguagem de programação especificada. Atualmente, o *software* é capaz de analisar apenas a linguagem C.

Como forma de mensurar a importância dos termos em um documento, o *YouPlag* provê duas técnicas: TF-IDF (-tfidf) e o WF-IDF (-wfidf). Tendo em mente os resultados obtidos, o esquema de ponderação TF-IDF foi definido como padrão.

Com o objetivo de testar a eficácia de diferentes modelos de recuperação, a similaridade por cosseno e o coeficiente de Dice foram implementados. Devido aos resultados apresentados, o padrão é o coeficiente de Dice.

É imprescindível que um sistema de detecção de plágio tenha como característica a capacidade de detectar similaridades (Whale, 1990). Pares de códigos cujas similaridades ultrapassem um limiar T, são classificados como possíveis plágios. Dada a dificuldade de inicialmente definir um T ótimo, o limiar de Otsu foi definido como padrão.

Uma vez definido o diretório, a especificação dos programas indica quais serão

submetidos para análise de similaridade. É possível enviar todos arquivos através de um ponto seguido da extensão da linguagem (e.g. *.c*).

6 Resultados

Este capítulo tem por objetivo apresentar e avaliar os resultados obtidos pelos três *softwares* analisados, i.e. YouPlag, JPlag, e MOSS. Para mensurar e comparar corretamente os resultados obtidos, foi montada uma lista com os códigos classificados como possíveis cópias. A fim de avaliar a eficácia dos resultados obtidos, a precisão e a exatidão foram calculadas.

A precisão refere-se à proporção de documentos relevantes ao usuário, i.e. verdadeiros positivos (Manning; Raghavan; Schütze , 2009). É definida por:

$$Precisão = \frac{\text{Verdadeiros positivos}}{\text{Verdadeiros positivos} + \text{Falsos positivos}} \tag{6.1}$$

A exatidão, por sua vez, busca definir se os resultados apresentados são de fato corretos (Manning; Raghavan; Schütze , 2009). É dada por:

$$Exatidão = \frac{\text{Verdadeiros positivos}}{\text{Verdadeiros positivos} + \text{Falsos negativos}} \tag{6.2}$$

6.1 Resultados obtidos com limiar de 50%

Os resultados apresentados nessa seção foram atingidos utilizando um limiar de 50%. Os *softwares* foram executados em suas configurações padrões. Para o *YouPlag*, além da configuração padrão, também foram testados e combinados parâmetros distintos.

6.1.1 Resultados obtidos na base 1

A Tabela 4 apresenta os resultados obtidos na base de exercícios de operações em *skiplist*. Nela é possível observar que os melhores resultados pertencem ao *YouPlag* na configuração padrão e personalizada (utilizando similaridade por cosseno e TF-IDF), apresentando precisão e exatidão de 60% e 100%, respectivamente.

Os 4 primeiros resultados na Tabela 4 obtiveram exatidão de 100%, indicando que nenhum possível plágio foi omitido. Porém, nenhum *software* obteve precisão acima dos 70%. Assim sendo, o trabalho de análise dos códigos reportados como prováveis dependências ao usuário seria em vão para alguns códigos; tendendo a piorar à medida que o número de fontes aumenta.

Os resultados menos satisfatórios pertecem ao *YouPlag* com a utilização do parâmetro WF-IDF. Acredita-se que isso se deva à distribuição do número de *4-grams* entre

os códigos. Na Figura 7, percebe-se que as duplas ({fonte05 e fonte10} e {fonte09 e fonte11}), identificados como possíveis plágios, apresentam quantidades de *4-grams* não tão discrepantes. E, tendo em vista que, ao passo que o número de linhas em um código fonte aumenta, maior é a probabilidade de *keywords* repetidas, a ponderação WF-IDF não atribuiu similaridade maior que 50% onde a diferença entre o número de *4-grams* em um par fosse relativamente grande; vide {fonte05 e fonte14} identificados pelo docente como possíveis plágios.

Tabela 4 – Resultados na base de exercícios de *skiplist*.

Software	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
JPlag	7	3	4	0	43%	100%
MOSS	8	3	5	0	38%	100%
YouPlag -dice -tfidf	5	3	2	0	60%	100%
YouPlag -cosine -tfidf	5	3	2	0	60%	100%
YouPlag -cosine -wfidf	2	2	0	1	100%	67%
YouPlag -dice -wfidf	2	2	0	1	100%	67%

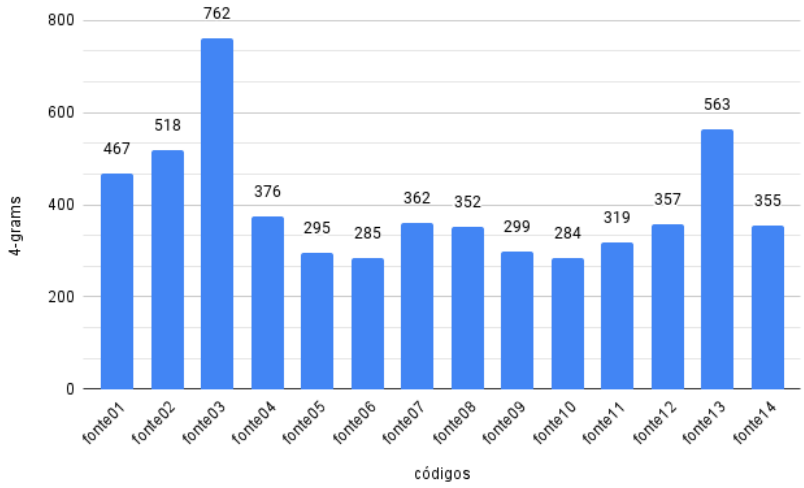


Figura 7 – Distribuição de *4-grams* entre os documentos da base 1 obtidos pelo YouPlag.

Fonte: O autor

6.1.2 Resultados obtidos na base 2

Os resultados obtidos na base de exercícios de operações em árvore B são apresentados na Tabela 5. Verifica-se que todos os *softwares* em análise obtiveram exatidão de 100%. O *YouPlag*, nas configurações padrão e personalizada utilizando similaridade por cosseno e TF-IDF, atingiu precisão de 50%.

Novamente, percebe-se que a distribuição do número de *4-grams* granjeado em cada arquivo influencia na ponderação dos termos, e, por consequência, no grau de similaridade. Nessa base, o par (codigo02 e codigo06), classificado como independente pelo docente, obteve grau de similaridade de aproximadamente 58.9% no esquema de ponderação TF-IDF. A distribuição dos *4-grams* é apresentada na Figura 8.

No JPlag, o par de códigos (codigo02 e codigo06) alcançou aproximadamente 27.9% de similaridade. Geralmente, espera-se, no caso do JPlag, que, quando dois códigos possuem tamanhos diametralmente distintos, seus valores de similaridade sejam relativamente baixos em comparação à técnicas de ATMs, vide *YouPlag*. Pois esse *software* busca estender ao máximo a correspondência entre dois fontes a partir de um valor mínimo de correspondência (*minimum match length*).

Para a dupla analisada, o MOSS, assímetricamente, obteve os seguintes valores: (codigo02 e codigo06) 46% de similaridade; e, (codigo06 e codigo02) 25%. Tal resultado se dá por conta das características do algoritmo *winnowing* robusto.

Tabela 5 – Resultados na base de exercícios de árvore B.

Software	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
JPlag	1	1	0	0	100%	100%
MOSS	1	1	0	0	100%	100%
YouPlag -dice -tfidf	2	1	1	0	50%	100%
YouPlag -cosine -tfidf	2	1	1	0	50%	100%
YouPlag -cosine -wfidf	1	1	0	0	100%	100%
YouPlag -dice -wfidf	1	1	0	0	100%	100%

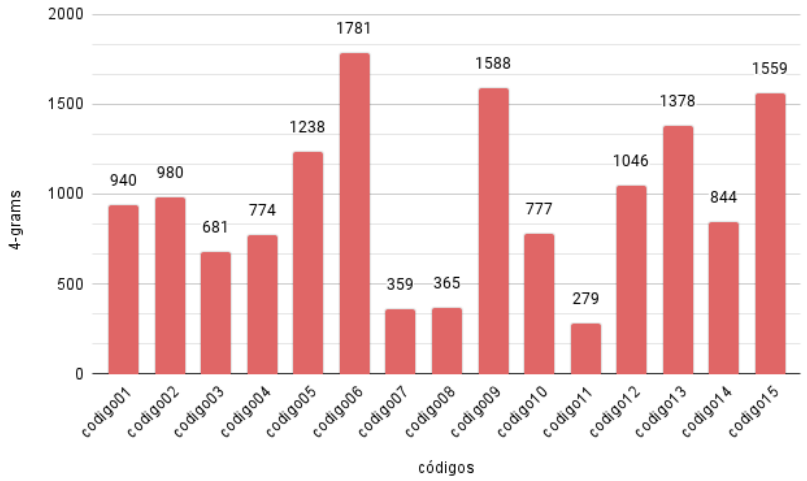


Figura 8 – Distribuição de *4-grams* entre os documentos da base 2 obtidos pelo YouPlag.

Fonte: O autor

6.1.3 Resultados obtidos na base 3

Através da Tabela 6, percebe-se um fenômeno curioso: todos os *softwares* alcançaram o mesmo valor de precisão e exatidão. Além disso, nenhum atribuiu grau de similaridade maior ou igual a 50% para os dois pares inspecionados manualmente e identificados como possíveis dependências.

A partir da Figura 9, nota-se que um limiar na faixa dos 30% poderia trazer resultados mais interessantes, e, por consequência, melhorar a exatidão do *YouPlag* e JPlag.

Infortunadamente, o MOSS não provê nenhuma forma de calcular a distribuição de similaridades entre os pares. Porém, mesmo com essa limitação, analisando-se os resultados gerados pelo MOSS, se um limiar de 30% fosse utilizado, a exatidão também melhoraria cerca de 33%.

Tabela 6 – Resultados na base de exercícios de operações matemáticas.

Software	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
JPlag	1	1	0	2	100%	33%
MOSS	1	1	0	2	100%	33%
YouPlag -dice -tfidf	1	1	0	2	100%	33%
YouPlag -cosine -tfidf	1	1	0	2	100%	33%
YouPlag -cosine -wfidf	1	1	0	2	100%	33%
YouPlag -dice -wfidf	1	1	0	2	100%	33%

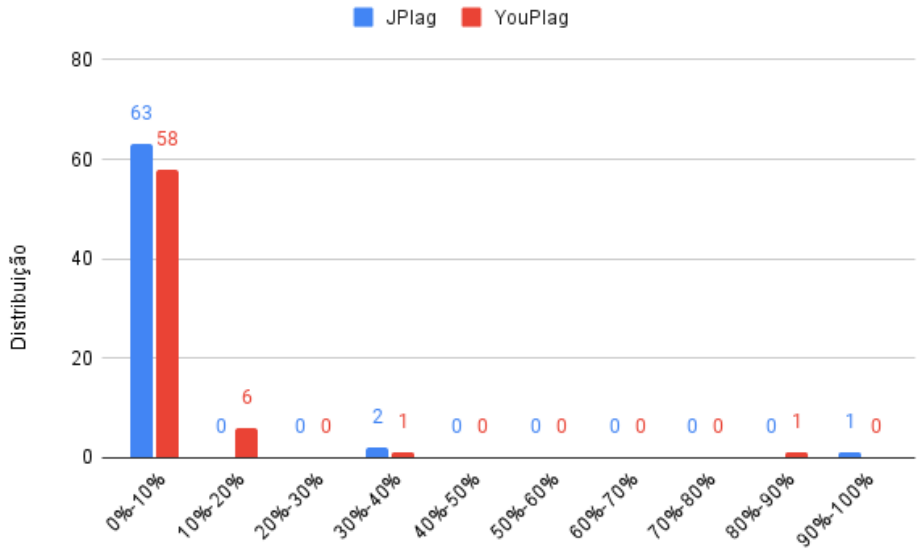


Figura 9 – Comparação da distribuição das similaridades obtidas pelo *YouPlag* e JPlag.

Fonte: O autor

6.2 Resultados obtidos com o limiar de Otsu

Como forma de superar o problema de definição manual do limiar T, esta seção tem por objetivo exibir os resultados alcançados com a automatização dessa tarefa através do limiar de Otsu. Conforme descrito na seção 5.8, as similaridades entre os pares foram dispostas em um histograma. Assim sendo, o possível limiar encontra-se entre 0% e 90%.

A escolha da automatização do limiar através do método de Otsu se deu através da seguinte observação: o histograma da distribuição de similaridades tende a ser bimodal. Logo, o limiar de Otsu poderá incrementar a separabilidade do YouPlag.

6.2.1 Resultados obtidos na base 1

A partir da Tabela 7, observa-se que todas as configurações do *YouPlag* obtiveram a mesma porcentagem de exatidão. No contexto de detecção de plágio, deseja-se um valor elevado de exatidão, ou seja, o algoritmo deve ser capaz de evitar ao máximo falsos negativos. Porém, nota-se que o limiar obtido automaticamente trouxe nesta base uma grande desvantagem: a precisão diminiu consideravelmente quando comparada aos resultados apresentados na Tabela 4. Consequentemente, a seletividade, i.e. capacidade ignorar falsos positivos (Whale, 1990), foi dratiscamente afetada pelo limiar de 20%.

Mesmo diante do declínio da precisão, a exatidão para as configurações utilizando WF-IDF, sem ter em conta o método de cálculo de similaridade, obteve uma melhora de aproximadamente 33% em relação ao limiar de 50%, vide Figura 10. À vista disso, um par classificado anteriormente como independente foi definido possível plágio.

Tabela 7 – Resultados na base de exercícios de *skiplist* utilizando o limiar de Otsu.

Software	Limiar%	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
YouPlag -dice -tfidf	20	14	3	11	0	21%	100%
YouPlag -cosine -tfidf	20	14	3	11	0	21%	100%
YouPlag -cosine -wfidf	20	9	3	6	0	33%	100%
YouPlag -dice -wfidf	20	9	3	6	0	33%	100%

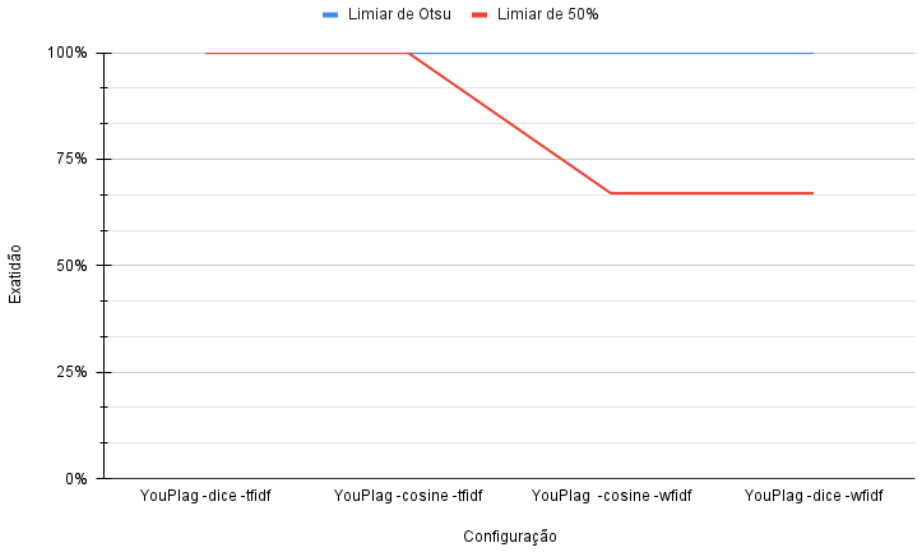


Figura 10 – Comparação entre os valores de exatidão.

Fonte: O autor

6.2.2 Resultados obtidos na base 2

Na Tabela 8, nota-se que todas as configurações, até em esquemas de ponderação distintos, obtiveram o mesmo limiar de 20%. Percebe-se que as execuções que fizeram uso

do WF-IDF retornaram apenas 4 documentos relevantes, dos quais apenas 1 foi classificado como possível plágio. Por conta disso, o limiar de Otsu obteve 25% de precisão, ocasionando uma queda de aproximadamente 75% quando comparada à Tabela 5.

Ao passo que o limiar de 50% apresentado na Tabela 5 obteve 50% de precisão para as configurações do *YouPlag* que fizeram uso do TF-IDF, o limiar obtido automaticamente diminuiu consideravelmente a precisão do *software*. Assim sendo, dos 15 códigos presentes na base, o docente teria que avaliar 9 programas fontes que são independentes. Novamente, verifica-se que a seletividade do algoritmo fora prejudicada.

Tabela 8 – Resultados na base de exercícios de árvore B utilizando o limiar de Otsu.

Software	Limiar%	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
YouPlag -dice -tfidf	20	10	1	9	0	10%	100%
YouPlag -cosine -tfidf	20	10	1	9	0	10%	100%
YouPlag -cosine -wfidf	20	4	1	3	0	25%	100%
YouPlag -dice -wfidf	20	4	1	3	0	25%	100%

6.2.3 Resultados obtidos na base 3

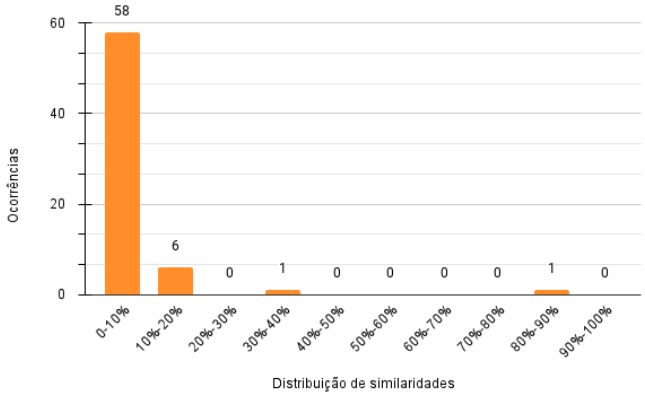
A Tabela 9 apresenta os efeitos do limiar de Otsu na base 3. Quando comparada à Tabela 6, percebe-se uma melhora significativa nas porcentagens de exatidão. Indicando que um par de códigos classificado anteriormente como independente foi definido como possível cópia através dos limiares de 30% e 10%.

Observa-se que as configuraçõe do YouPlag que utilizaram o WF-IDF, obtiveram limiar de 10%. Acredita-se que isso se deva à distribuição das similaridades, vide Figura 11, e à principal característica do limiar de Otsu: expandir a variância entre classes (GREENSTED, 2010).

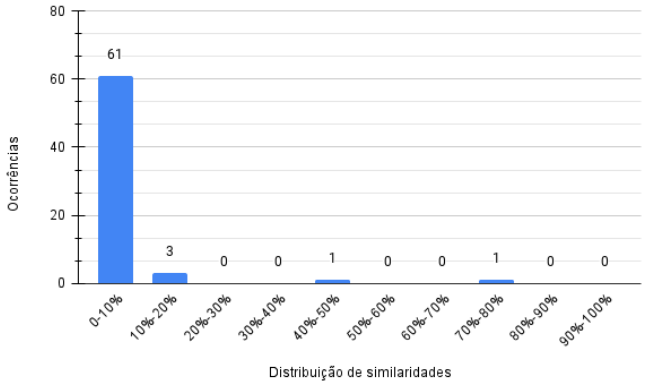
Comparando-se a Tabela 9 com a 6, observa-se que a precisão obtida para o WF-IDF apresentou uma considerável queda (cerca de 60%). A partir da análise dos resultados da base 3, é possível confirmar o que fora assertado por (Whale, 1990): a precisão e a exatidão tendem a ser inversamente proporcionais. Porém, mesmo diante desse fato, é desejável que o algoritmo de detecção de plágio mantenha um nível adequado entre essas duas métricas.

Tabela 9 – Resultados na base de exercícios de operações matemáticas utilizando o limiar de Otsu.

Software	Limiar%	Documentos Recuperados	Documentos Relevantes	Falsos Positivos	Falsos Negativos	Precisão	Exatidão
YouPlag -dice -tfidf	30	2	2	0	1	100%	67%
YouPlag -cosine -tfidf	30	2	2	0	1	100%	67%
YouPlag -cosine -wfidf	10	5	2	3	1	40%	67%
YouPlag -dice -wfidf	10	5	2	3	1	40%	67%



(a) Distribuição de similaridades através do TF-IDF.



(b) Distribuição de similaridades através do WF-IDF.

Figura 11 – Comparação entre as distribuições de similaridades obtidas entre dois esquemas de ponderação diferentes.

Fonte: O autor

7 Conclusão

O objetivo deste trabalho foi desenvolver um *software* de detecção de plágio em código fonte a partir do estudo e combinação de técnicas já existentes, visando melhorar os resultados apresentados aos usuários. Analisando-se os resultados, é possível dizer que as técnicas de RI implementadas geraram resultados satisfatórios e próximos ou até melhores aos alcançados pelo JPlag e MOSS. Nas bases analisadas, nota-se que tanto o coeficiente de Dice quanto a similaridade por cosseno obtiveram as mesmas porcentagens de precisão e exatidão.

Em relação à ponderação de termos nas execuções do YouPlag, verifica-se que o esquema TF-IDF, no geral, apresentou ser mais eficaz (levando em consideração o limiar de 50%); diferentemente do WF-IDF que tende a reduzir o nível de similaridade entre os pares. Outro fato importante a ser mencionado é a utilização do IDF juntamente aos *4-grams*. Em alguns casos, como observado na base 3, cujo exercício era de nível iniciante, espera-se que muitos códigos possuam a mesma sequência de *4-grams*, tendo em vista a trivialidade do problema (Whale, 1990). Logo, por conta dessa repetição, o IDF atribuiu um peso muito baixo (próximo de 0) para termos muito frequentes. Reduzindo, portanto, a probabilidade de aparição de falsos positivos.

Além disso, o limiar de Otsu foi implementado como forma de verificar a automação do limiar de similaridade. Baseando-se nos resultados apresentados na seção 6.2, observa-se que para todas as execuções do YouPlag as porcentagens de exatidão se mantiveram ou foram melhoradas comparando-as ao limiar de 50%. Porém, a precisão fora bastante afetada.

No contexto de detecção de plágio, a limiarização mostrou-se fundamental na eficácia do sistema. Assim sendo, seria interessante investigar quais seriam os métodos mais adequados para obter o limiar T ótimo. As formas poderiam ser distintas: desde apresentação de elementos visuais (dados estatísticos, grafos de dependência, etc.) a técnicas avançadas de inteligência artificial (IA).

Referências

Burrows, S.; Tahaghoghi, S.; Zobel, J. Efficient plagiarism detection for large code repositories. *SOFTWARE—PRACTICE AND EXPERIENCE*, v. 37, n. 2, p. 151–175, 2007. Citado 4 vezes nas páginas 15, 25, 26 e 28.

Chuda, D. et al. The issue of (software) plagiarism: A student view. *IEEE TRANSACTIONS ON EDUCATION*, v. 55, n. 1, p. 22–28, 2012. Citado na página 12.

Cosma, G.; Joy, M. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, v. 51, n. 2, p. 195–200, 2008. Citado na página 11.

Croft, W. B.; Metzler, D.; Strohman, T. *Search Engines - Information Retrieval in Practice*. [S.l.]: Pearson Education, Inc., 2015. Citado 4 vezes nas páginas 14, 16, 17 e 18.

GREENSTED, A. *Otsu Thresholding*. 2010. Disponível em: <http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html>. Acesso em: 10 out. 2021. Citado 3 vezes nas páginas 19, 32 e 39.

Kermek, D.; MATIJA, N. Process model improvement for source code plagiarism detection in student programming assignments. *Informatics in Education*, v. 15, n. 1, p. 103–126, 2016. Citado na página 11.

Lancaster, T. *Effective and Efficient Plagiarism Detection*. Tese (Doutorado) — School of Computing, Information Systems and Mathematics South Bank University, Londres, Reino Unido, 2003. Disponível em: <https://www.researchgate.net/publication/228729388_Effective_and_Efficient_Plagiarism_Detection>. Acesso em: 03 abr. 2021. Citado 2 vezes nas páginas 12 e 17.

Manning, C. D.; Raghavan, P.; Schütze , H. *An Introduction to Information Retrieval*. [S.l.]: Cambridge University Press, 2009. Citado 7 vezes nas páginas 14, 15, 16, 17, 18, 28 e 34.

Murzova, A.; Seth, S. *Otsu's Thresholding with OpenCV*. 2020. LearnOpenCV. Disponível em: <https://learnopencv.com/otsu-thresholding-with-opencv/>. Acesso em: 3 nov. 2021. Citado na página 19.

Ottenstein, K. J. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, v. 1, n. 1, p. 1–18, 1976. Citado 2 vezes nas páginas 11 e 25.

Prabhakaran, S. *Cosine Similarity – Understanding the math and how it works (with python codes)*. 2020. Machine Learning Plus. Disponível em: <https://www.machinelearningplus.com/nlp/cosine-similarity/>. Acesso em: 16 mai. 2021. Citado na página 14.

Prechelt, L.; Malpohl, G.; Philippsen, M. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, v. 8, n. 11, p. 1015–1038, 2003. Citado 4 vezes nas páginas 11, 21, 22 e 23.

Robertson, S. Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, v. 60, n. 5, p. 503–520, 2004. Citado na página 15.

Sanders, R. *On-line plagiarism detector helps computer science professors bust cheating programmers*. 1997. Berkley. Disponível em: <https://www.berkeley.edu/news/media/releases/97legacy/11_19_97b.html>. Acesso em: 14 abr. 2021. Citado na página 11.

Schleimer, S.; Wilkerson, D. S.; Aiken, A. Winnowing: Local algorithms for document fingerprinting. *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, v. 10, n. 1, p. 76–85, 2003. Citado 3 vezes nas páginas 17, 19 e 23.

STEFANOVIč, P.; KURASOVA, O.; ŠTRIMAITIS, R. The n-grams based text similarity detection approach using self-organizing maps and similarity measures. *Applied Sciences*, v. 9, p. 1870, 05 2019. Citado na página 17.

Whale, G. Identification of program similarity in large populations. *The computer journal*, v. 33, n. 2, p. 140–146, 1990. Citado 6 vezes nas páginas 11, 17, 32, 38, 39 e 41.

Wise, M. J. Detection of similarities in student programs: Yap'ing may be preferable to plague'ing. *ACM SIGCSE*, v. 24, n. 1, p. 268–271, 1992. Citado na página 11.

Wise, M. J. Yap3: improved detection of similarities in computer program and other texts. *ACM SIGCSE Bulletin*, v. 28, n. 1, p. 130–134, 1996. Citado 2 vezes nas páginas 21 e 22.

Zuhoor, A. et al. Plagdetect: A java programming plagiarism detection tool. *ACM Inroads*, v. 1, n. 4, p. 66–71, 2010. Citado 2 vezes nas páginas 11 e 25.