Uso do Algoritmo Distância de Edição com Técnicas de Pré-Processamento para Apoiar a Identificação de Plágio em Códigos-Fonte de Problemas de Programação Introdutória

Rodrigo Elias Francisco[1,2], **Ana Paula Ambrósio**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Câmpus Samambaia – Goiânia – GO – Brasil

[2]Instituto Federal Goiano (IFGoiano) - Campus Morrinhos
Rodovia BR-153, Km 633 – Morrinhos – GO – Brasil

rodrigo.francisco@ifgoiano.edu.br, apaula@inf.ufg.br

Abstract. *This paper addresses the problem of plagiarism in introductory programming education in the context of an automatic correction and management system for lists of exercises. Aiming to identify a plagiarism detection tool that could be incorporated into the system, a literature review was undertaken that detailed the available solutions. Given the lack of open-source solutions that could be executed locally with favorable results, we chose to develop a new tool, adapted to the context of introductory programming, with simple problems, and lexical changes as the main plagiarism strategy. The proposal combines a normalization process with the Edit Distance algorithm. The proposal was tested with real data obtained from interactions of introductory programming students with an online judge tool. The paper also addressed the analysis of plagiarism in a broader context, analysing the behavior of a student or a cohort regarding a list of exercises. The results indicate that detailed analysis of the data occurring in an interval of time may bring a different approach which allows a more global view of the learning environment.*

Resumo. *Este trabalho aborda o problema de plágio no ensino de programação introdutória no contexto de um sistema de administração e correção automática de listas de exercícios. Com o objetivo de identificar uma ferramenta de detecção de plágio que pudesse ser incorporada no sistema, foi feito um levantamento bibliográfico que detalhou as soluções disponíveis. Diante da falta de soluções open-source que pudessem ser executadas localmente com resultados favoráveis, optou-se pelo desenvolvimento de uma ferramenta própria, adaptada ao contexto de programação introdutória, com problemas simples, e modificações léxicas como principal estratégia de plágio. A proposta apresentada combina uma estratégia de normalização com o algoritmo Distância de Edição. A proposta foi testada com dados reais, oriundos de interações de alunos de programação introdutória com uma ferramenta de juiz online. O trabalho também aborda a análise do plágio em um contexto mais abrangente, como o comportamento de um aluno ou de uma turma com relação a uma lista de exercícios. Os resultados indicam que análises detalhadas dos dados ocorrendo em um intervalo de tempo podem trazer uma perspectiva diferente que permite uma visão mais global do ambiente de ensino.*

FRANCISCO, R. E.; AMBROSIO, A. P.;
Uso do Algoritmo Distância de Edição com Técnicas de Pré-Processamento para Apoiar a Identificação de Plágio em Códigos-Fonte de Problemas de Programação Introdutória
iSys | Revista Brasileira de Sistemas de Informação, Rio de Janeiro, vol. 9, No. 2, pp. 32-52, 2016

1. Introdução

Lidar com plágio em códigos-fonte de programas é um desafio. Esse desafio começa com a definição de plágio. Copiar parte ou todo um programa e apresentá-lo como seu é considerado plágio, mesmo quando o programa é o resultado de um trabalho conjunto [Wagner 2000, Joy and Luck 1999]. Além disso, existem casos de plágio em que se fazem modificações estruturais nos programas, mantendo as mesmas funcionalidades. Porém, citar fontes e dar o devido crédito ao autor pode ser o suficiente para passar de plágio para referência [Bakshi 2010].

Diversos relatos mostram que o plágio em programas de computadores é maior que em outras áreas [Baugh et al. 2012, Wagner 2000, Bakshi 2010]. Baugh et al. (2012) relata que em *Stanford University* 23% das violações de seu código de honra foram feitas por alunos de Introdução à Programação de Computadores, enquanto que estudantes de Ciência da Computação são somente 6.5% do total de alunos. Na *Brown University*, 70% dos casos de plágio analisados vieram do departamento de Ciência da Computação [Bakshi 2010]. Wagner (2000) apresenta casos semelhantes no *Massachusetts Institute of Technology* (MIT) e na *University of Texas*. No entanto, isso pode ser porque muitos cursos de Computação checam sistematicamente a cópia de programas de maneira automática, usando *software* específico, o que não ocorre em outras áreas [Bakshi 2010].

Vale ressaltar que o plágio em programas tem algumas características que o diferenciam do plágio em outras disciplinas. Os programas, muitas vezes, são bastante simples e são implementados em ambientes similares. Problemas mais simples implicam em códigos-fonte menores e mais parecidos. Por usarem uma linguagem de programação, que possui um conjunto de comandos limitado e sintaxe bem definida, a liberdade de mudança não é a mesma que em um texto. Além disso, em muitos casos, o enunciado do problema tem formatos da entrada e de saída pré-definidos limitando ainda mais a possibilidade de diferenciação do código. Este cenário, frequentemente encontrado na disciplina de Introdução à Programação de Computadores, traz dificuldades na detecção de plágio.

Várias razões são dadas para "justificar"o plágio [Wagner 2000] e estão ligadas a aspectos diversos como: organização da disciplina, currículo do curso, professores, ambiente de programação, e problemas pessoais. Razões incluem: *deadline* apertado, com pouca ou nenhuma nota ofertada para soluções parciais; alunos consideram o trabalho chato ou muito difícil; reprovação quando os exercícios não são entregues, mesmo com boas notas nos exames; uso de listas de exercícios de turmas anteriores; ter a visão de que programação é irrelevante em suas carreiras; falta de base; entre outros.

Os resultados apresentados na pesquisa realizada com 115 estudantes da área de informática em uma universidade privada dos Estados Unidos [Baugh et al. 2012] mostraram que: 52% dos alunos acreditam que copiar código-fonte da internet é errado, mas apenas 30% dos alunos acreditam que é preciso citar os códigos baixados da internet; 78% dos alunos disseram já ter pesquisado código-fonte na internet como apoio às atividades, e 17% dos alunos confirmaram já terem usado os códigos-fonte baixados como respostas em atividades do curso. Os estudantes pesquisados consideram ser mais errado copiar dos colegas do que pesquisar as respostas na *internet*.

Uma pesquisa feita na *School of Computer Science and Software Engineering*

da *Monash University* e na *School of Information Technology* da *Swinburne University* na Austrália [Sheard et al. 2002] aplicou um questionário com 18 situações e pediu aos alunos para classificarem, usando uma escala *Likert* de 5 pontos, se eles achavam que aquela situação era aceitável do ponto de vista ético. Na grande maioria das situações, envolvendo cola em exames, roubo e plágio explícito, os alunos concordaram que as situações não eram aceitáveis. No entanto, em algumas situações as opiniões variaram. Casos onde dois alunos colaboravam para resolver um problema que deveria ser resolvido individualmente, pedir ajuda através da internet, e submeter trabalhos de alunos de outros cursos ou turmas, foram considerados aceitáveis por vários alunos.

Mesmo junto a professores há diferentes opiniões sobre o aspecto de coloboração na realização de programas. *Georgia Tech* permite que seus alunos façam trabalhos colaborativos, mas depois pede que os alunos façam demonstrações orais do funcionamento do *software* [Baugh et al. 2012]. O mesmo ocorre na *Monash University* [Sheard et al. 2002]. Em outras universidades, onde a avaliação é feita através de submissão eletrônica, a detecção de plágio costuma ser feita de maneira automática, como é o caso da *University of Swinburne*. Nessas situações, cópia de programas não é aceitável.

Um estudo apresenta uma lista de técnicas usadas por alunos para plagiar [Joy and Luck 1999]. Estas incluem (1) mudanças léxicas, que podem ser feitas em um editor de texto e não exigem conhecimento da linguagem, como por exemplo inclusão, modificação ou remoção de comentários, mudanças na formatação, e troca de nomes de variáveis, ou (2) mudanças estruturais, que exigem maior conhecimento, incluindo a capacidade de analisar o código (*parse*), como por exemplo, mudar o tipo de *loop* (*for/while*), substituir *if's* aninhados por comandos do tipo *case*, trocar a ordem dos comandos sem afetar a estrutura, substituir chamadas a procedimentos pelo próprio procedimento, trocar a ordem dos operadores (e.g. x<y pode ser substituído por y >= x).

Outro trabalho faz análise semelhante. No artigo, os autores afirmam que é possível analisar a similaridade entre códigos-fonte a partir da semelhança representacional ou comportamental [Goya et al. 2014]. A semelhança representacional é sintática e refere-se ao programa sendo uma sequência de caracteres, enquanto a semelhança comportamental é semântica e pode ser definida pelas funções que os programas implementam. As ferramentas para análise de similaridade diferenciam-se no método de comparação e qual tipo de semelhança se pretende analisar. Esse cenário, aliado à dificuldade na definição dos tipos de similaridade, traz desafios para a pesquisa sobre plágio e a necessidade de análise humana durante o processo.

No contexto específico de disciplinas introdutórias de programação, a maioria dos casos de plágio utiliza mudanças léxicas, que são mais fáceis de serem realizadas a partir de alterações nos arquivos dos programas em editores de texto ou ambientes de desenvolvimento. Mudanças estruturais exigem um conhecimento de programação que os alunos muitas vezes ainda não possuem, e que na maioria das vezes é a razão que leva o aluno a plagiar.

Este artigo apresenta uma estratégia para apoio à identificação de plágio em códigos-fonte de programação introdutória. A estratégia foi projetada para uso no

sistema *PROBOCA*, que estuda a adaptação do *BOCA* [De Campos and Ferreira 2004], um sistema de juiz *online*. Esse contexto de ensino trabalha inicialmente com problemas simples, que possuem soluções com poucas linhas de código, e que vão aumentando gradativamente de nível de dificuldade. Essa simplicidade traz requisitos adicionais que devem ser tratados e serão discutidos no decorrer do trabalho.

A estratégia proposta, que visa ser parte de um sistema de apoio à identificação de plágio na disciplina de Introdução à Programação de Computadores, contribui para resolver o problema em questão e é fortemente adaptada ao seu contexto. Isto é, ela foi desenvolvida para trabalhar com problemas simples em C, com poucos comandos; foca o plágio do tipo léxico, mais comum em disciplinas introdutórias e verificadas na prática do dia-a-dia na sala de aula; é de fácil uso, sem exigência de configurações complicadas; é *open-source*; e é executada localmente, podendo ser incorporada no sistema.

É uma proposta simples e eficaz, com ênfase no pré-processamento dos programas submetidos. A solução adota um algoritmo conhecido, algoritmo de Distância de Edição, e um conjunto restrito de normalizações, o que facilita sua implementação e reprodução. O pré-processamento foi definido a partir da análise de normalizações propostas na literatura e refinado por meio de vários testes usando o banco de problemas disponível.

Diversos *softwares* já foram desenvolvidos para detectar plágio em código-fonte de programas. A seção 2 apresenta uma variedade de ferramentas de apoio à análise de plágio existentes e contextualiza-as neste trabalho. A seção 3 mostra o algoritmo proposto juntamente com o esquema de normalização. A seção 4 apresenta as ações tomadas para validar a estratégia proposta. Para isso, foram realizados testes com dados reais a partir do banco de dados do sistema *BOCA*. A seção 5 apresenta uma proposta de uso da informação disponibilizada pelo mecanismo de detecção de plágio para a geração de relatórios. A seção 6, que conclui o trabalho, aborda os contribuições e desafios da pesquisa realizada.

2. Trabalhos correlatos

Existem vários programas para apoiar a identificação de plágio em código-fonte. Entre eles: *CodeMatch* [Zeidman 2006], *CPD* [Copeland 2003], *JPlag* [Prechelt et al. 2002], *Marble* [Hage et al. 2010], *MOSS* [Schleimer et al. 2003], *Plaggie* [Ahtiainen et al. 2006], *Sherlock* [Sherlock nd], *SIM* [Hage et al. 2010], *SID* [Chen et al. 2004] e *YAP3* [Wise 1996]. Apesar de implementarem algoritmos distintos, algumas estratégias e conceitos são comuns.

Os programas calculam um índice de similaridade entre textos de códigos-fonte, oferecendo suporte para a verificação de plágio. Os programas possuem as etapas de pré-processamento, processamento e pós-processamento [Kleiman 2007]. O pré-processamento visa gerar uma sequência de átomos para o processamento, e eventualmente trabalha com regras de normalização. O processamento compara os conjuntos de átomos das duas submissões usando um algoritmo, e o pós-processamento calcula um grau de similaridade a partir dos resultados obtidos nas comparações, apresentando-o para o usuário.

O conceito de átomo, do inglês *token*, é frequentemente usado nas estratégias de análise de similaridade. Kleiman (2007) o define como elementos indivisíveis de uma

cadeia ou sequência. O que é considerado um átomo pode variar. *Hash* é outro conceito bastante usado nos algoritmos de análise de similaridade. Trata-se de uma função que recebe como entrada um conjunto de dados, e.g. um arquivo ou uma *string*, e o transforma em uma sequência menor de informações, que pode ser representada por uma *string* ou uma sequência de bits. Esse conceito é bastante usado na área de Criptografia e Segurança da Informação.

CodeMatch é um programa que usa força bruta para calcular a correlação entre pares de código-fonte usando diversos aspectos do código. Ele combina cinco algoritmos de comparação [Zeidman 2006, SAFE]: *Statement Matching*, *Comment/String Matching*, *Instruction Sequence Matching*, *Identifier Matching*, *Correlation Score*. As similaridades entre os pares são apresentadas como uma pontuação que varia de 0 a 100.

O algoritmo *Statement Matching* compara cada linha funcional do código-fonte de ambos os arquivos, excluindo as linhas de comentário e as linhas que possuem somente palavras reservadas da linguagem. O algoritmo conta o número de linhas de instrução correspondentes nos dois arquivos. O algoritmo *Comment/String Matching* compara cada linha de comentário e cada sequência de caracteres de ambos os arquivos. As sequências de espaços em branco são convertidos em espaços individuais. O algoritmo *Instruction Sequence Matching* compara a primeira instrução de cada linha de origem no par de arquivos, ignorando linhas em branco e linhas de comentários. A função do *Identifier Matching* é comparar o número de identificadores que não são palavras reservadas da linguagem de programação. Por fim, *Correlation Score* determina uma correlação para a semelhança entre os pares de arquivos.

O programa *CPD* [Copeland 2003] foi desenvolvido para identificar partes duplicadas em um mesmo código-fonte, criadas a partir de copiar/colar, quando a mesma funcionalidade é reutilizada. Essas partes copiadas trazem problemas para a manutenção de *software*, além de dificultar o reuso. Apesar de possuir alguns aspectos relacionados com a identificação de plágio, é importante ressaltar que foi projetado para atender a requisitos diferentes.

Para analisar a similaridade entre as partes, no *CPD*, é criada uma tabela que possui uma coluna com sequências de átomos agrupados em uma *string* e outra coluna contendo o conjunto de posições em que a *string* aparece no código-fonte. A comparação para identificar a similaridade é feita com base nessas tabelas.

O *JPlag* [Prechelt et al. 2002] compara um fluxo contínuo de átomos de um arquivo com partes extraídas do outro. O valor da similaridade é calculado a partir das semelhanças encontradas. No pré-processamento do *JPlag*, os arquivos de código fonte são normalizados. Para isso ocorrem as remoções de espaços em branco, comentários e nomes de identificadores. Após essa normalização, faz-se a adição de informação semântica, que expressa a finalidade dos comandos, nos átomos quando possível.

O método utilizado no processamento do *JPlag* é o *String Tiling Greedy*. Seu objetivo é encontrar os maiores conjuntos de sequências de *strings* que se repetem nos dois arquivos e não se sobrepõem. Devido à complexidade computacional de encontrar essas ocorrências maximais de *substrings*, *String Tiling Greedy* é um algoritmo heurístico, i.e, visa trabalhar com os melhores resultados possíveis, que nem sempre são ótimos.

Plaggie é um programa criado para identificar a possibilidade de plágio entre

códigos-fonte escritos em Java [Ahtiainen et al. 2006]. A diferença do *Plaggie* com o *JPlag* é que o *Plaggie* é *open-source* e precisa ser instalado localmente.

O *YAP3* trabalha com as impressões digitais dos arquivos de código-fonte na análise de similaridade. Para isso utiliza a metodologia *Running-Karp-Rabin Greedy-String-Tiling* (RKS-GST) [Wise 1996]. O método empregado no *YAP3* também realiza uma normalização, no pré-processamento. A normalização remove dos códigos-fonte os comentários e *strings* constantes, transforma as letras maiúsculas em minúsculas, modifica os sinônimos para uma forma comum, reordena as funções para a sua ordem de chamada, e retira todos os átomos que não são da linguagem específica em análise. A estratégia para calcular a semelhança entre os arquivos busca um conjunto de regiões similares nos arquivos que não se sobrepõem. Nessa fase, ele usa o algoritmo guloso *RKS-GST*, que trabalha com impressões digitais em *hash-table*. Trata-se de um problema não polinomial. A estratégia de comparação possui custo O(n^2).

É importante observar que os programas *JPlag, Plaggie* e *YAP3* utilizam a estratégia *Greedy-String-Tiling* no seu método de comparação.

O programa *Marble* [Hage et al. 2010] foi desenvolvido em 2002 pela *Utrecht University* com finalidade de ser simples e de fácil manutenção para detectar casos suspeitos de similaridade em submissões de atividades de alunos. Ele faz normalizações nos arquivos gerando dois arquivos para cada código-fonte e computa a similaridade a partir do comando *diff* do Unix/Linux. O comando *diff* apresenta as linhas de um arquivo de texto que são diferentes das do outro. A similaridade é calculada a partir da razão entre esses números de linhas diferentes e o número total de linhas.

Marble faz suas normalizações analisando a estrutura do programa. Comentários, espaços em branco excessivos, *strings* constantes e declarações de importação são removidos. Outros símbolos são abstraídos ao seu "tipo", e.g, cada número hexadecimal é substituído por H e cada literal por L. O sistema computa duas versões normalizadas para cada arquivo. Na primeira versão, a ordem dos campos, métodos e classes internas é preservada como no arquivo original. Já na segunda, existem grupos para cada um desses componentes e esses são ordenados de maneira heurística. O usuário precisa escolher se usará somente uma versão normalizada ou se quer trabalhar com as duas. Essa escolha influencia nos resultados, pois o aluno pode ou não mudar a ordem de estruturas internas ao fazer plágio.

MOSS foi desenvolvido em 1994 em *Stanford University* e trabalha com comparação das impressões digitais dos documentos [Schleimer et al. 2003]. O funcionamento do *MOSS* pode ser entendido pelos seguintes passos: (a) é feita uma divisão do documento em *substrings* contíguos de tamanho k, sendo k escolhido pelo usuário; (b) aplica-se a função *hash* para cada *substring* gerada, formando as impressões digitais do documento; (c) seleciona-se as *hashs* conforme densidade configurada pelo usuário. Para isso é preciso calcular *(hash mod valor_configurado)*. Quanto menor o valor configurado, maior será o conjunto de *hashs* para comparação; (d) tendo executado os passos anteriores para os dois arquivos, calcula-se a similaridade com base no número de *hashs* semelhantes nos arquivos, considerando a densidade escolhida.

A *hash* gerada no *MOSS* é um valor numérico gerado a partir de um algoritmo que recebe uma *string*. Comparar *hashs* exige menos custo computacional do que comparar

strings caractere a caractere. O conceito de densidade tem a finalidade de reduzir a quantidade de *hashs* a serem comparadas e é implementado a partir da verificação de se a *hash* é divisível pelo valor configurado. Para isso é usado o resto da divisão (*mod*). É preciso configurar os parâmetros de densidade e tamanho das *substrings*. Essa configuração impacta no tempo de processamento e na qualidade dos resultados, e depende do material a ser analisado.

O programa *Sherlock* [Sherlock nd] foi proposto para auxiliar na detecção de plágio de textos a partir de similaridade. Ele trabalha com assinaturas digitais em partes do texto [Maciel 2014]. Na fase de pré-processamento, é feita a separação das palavras do texto em grupos de tamanho configurado pelo usuário. Cada conjunto de palavras é convertido em uma assinatura digital por meio de uma função *hash*. Na etapa de processamento, comparam-se as assinaturas digitais dos dois arquivos com a finalidade de calcular a similaridade.

Existem, no *Sherlock*, parâmetros a serem configurados pelo usuário. *Zerobits* refere-se à granularidade da comparação. Diminuir esse número implica em ter uma comparação mais exata e mais lenta. *Número de palavras* refere-se à quantidade de palavras que se deseja agrupar.

Maciel (2014) propõe o sistema *Sherlock N-Overlap*, que modifica o programa *Sherlock* para uso no contexto de plágio em respostas de problemas de programação introdutória. Para isso, foi feita uma modificação do coeficiente de similaridade, e a inclusão de quatro técnicas de normalização. As técnicas de normalização removem gradativamente partes específicas do código-fonte com a finalidade de preservar estruturas internas do mesmo. A pesquisa conclui que essa modificação estratégica possui resultados melhores comparados com o método tradicional do programa *Sherlock* quando aplicado a programas.

SIM [Hage et al. 2010] é um programa criado pela *VU University Amsterdam* para detectar plágio em códigos-fonte. É fortemente adaptado à situação de sua Universidade e por isso não é muito portátil. Ele divide cada código-fonte em um conjunto de átomos, formando tabelas de átomos para comparação das correspondências similares. Sua interface com usuário é por linha de comando, e possui diversos parâmetros, os quais, para serem configurados, exigem que o usuário tenha um certo conhecimento de sua estrutura interna [Grune 2012].

O programa *SID*, criado na *University Waterloo*, aceita submissões em Java e C++ [Chen et al. 2004]. Na fase de pré-processamento, faz uso de um analisador léxico específico da linguagem com finalidade de gerar sequências de átomos para o processamento. O processamento é feito com base na medida de informação compartilhada. Essa estratégia foi proposta a partir do conceito de complexidade de *Kolmogorov*. Sua estratégia de processamento utiliza um processo que foi inicialmente implementado para análise de DNA. Esse processo faz uso do algoritmo de Distância de Edição.

Um resumo das estratégias de pré-processamento para a normalização dos arquivos é apresentado na Tabela 1.

É importante observar que a etapa de pré-processamento tem forte influência na qualidade da análise de similaridade. Kleiman (2007), em sua conclusão, propõe que essa

Tabela 1. Estratégias de normalização

Legenda			
1	Remover comentários	10	Ordenar elementos internos do código-fonte
2	Remover linhas que possuem somente palavras reservadas da linguagem	11	Padronizar espaço entre elementos do código
3	Gerar sequência de token	12	Remover todos os caracteres situados entre aspas
4	Remover espaços em branco	13	Remover todos os valores literais
5	Remover nomes de identificadores	14	Remover palavras reservadas da linguagem
6	Adicionar informação semântica nos tokens	15	Remover palavras que não são da linguagem
7	Remover strings constantes	16	Remover constantes de caracteres
8	Remover declarações de importação	17	Converter letras maiúsculas para minúsculas
9	Substituir símbolos por seu tipo, e.g hexadecimal é substituído por H e literal por L	18	Converter sinônimos para uma forma comum
		19	Reordenar as funções para sua órdem de chamada

etapa é até mais importante do que a aplicação do algoritmo em si. Isto é corroborado por Maciel (2014), que concluiu que as regras de normalização criadas melhoraram significativamente os resultados do *Sherlock*.

A Tabela 2 apresenta uma comparação entre os programas discutidos. A Tabela mostra se o programa detecta mudanças léxicas, e/ou mudanças estruturais, e um resumo do pré-processamento que ele realiza usando como referência a Tabela 1. As mudanças léxicas são feitas quando é gerado um novo programa com código-fonte diferente do anterior e com a mesma estrutura, e.g. quando há mudanças em nomes de variáveis e inclusão de comentários. Já as mudanças estruturais podem ser percebidas quando um trecho de código tem sua estrutura alterada mas a semântica permanece a mesma, e.g. quando uma equação matemática possui sua estrutura alterada e preserva as mesmas propriedades da anterior.

Tabela 2. Comparação de estratégias usadas nos programas para análise de similaridade

Programa	Detecta mudanças léxicais	Detecta mudanças estruturais	Pré-processamento
CodeMatch	SIM	NÃO	1, 2
CPD	SIM	NÃO	3
Jplag	SIM	NÃO	1, 4, 5, 6
Plaggie	SIM	NÃO	1, 4, 5, 6
Marble	SIM	SIM	1, 4, 7, 8, 9, 10
MOSS	SIM	NÃO	• Informação Indisponível
Sherlock	SIM	NÃO	* Não possui
Sherlock N-Overlap	SIM	NÃO	1, 4, 5, 8, 11, 12, 13, 14.
SIM	SIM	NÃO	• Informação Indisponível
YAP3	SIM	SIM	1, 16, 17, 18, 19, 15
SID	SIM	NÃO	3

Marble, como pode ser visto na Tabela 2, detecta algumas mudanças estruturais nos programas, e para isso reordena elementos internos das classes como parte do pré-processamento. Já o *YAP3* detecta as mudanças estruturais relacionadas à ordem de

funções. Para tratar essa questão ele reordena as funções conforme a ordem de chamada. Porém, esses programas não conseguem detectar alguns tipos de mudanças estruturais, e.g. a troca da ordem dos operadores e a mudança da ordem dos comandos em um programa sem afetar o resultado.

Uma análise das ferramentas para a identificação de plágio, visando sua integração em um sistema que trabalha com programas de computadores foi feita por Martins (2014). As características comparadas foram: (1) Linguagens suportadas; (2) Possibilidade de adicionar linguagens; (3) Qualidade dos resultados; (4) Interface; (5) Possibilidade de ignorar o código-base; (6) Submeter grupos de arquivos para processar; (7) Possibilidade de executar o programa local, sem depender de acesso à internet; e se a ferramenta é (8) *Open source*. A Tabela 3 mostra o resultado desse levantamento.

Tabela 3. Comparação de programas para analisar similaridade [Martins et al. 2014]

PROGRAMA	1 Linguagens suportadas	2 Possibilidade de adicionar linguagens	3 Qualidade dos resultados	4 Interface	5 Possibilidade de ignorar o código-base	6 Submeter grupos de arquivos para processar	7 Possibilidade de executar o programa local	8 Open source
CodeMatch	36	-	X	X	X	-	X	-
CPD	6	X	-	X	?	?	X	X
JPlag	6	-	X	X	X	X	-	-
Marble	5	X	X	-	X	?	X	-
MOSS	25	-	X	X	X	X	-	-
Plaggie	1	?	?	X	?	?	X	X
Sherlock	1	-	-	-	-	-	X	?
SIM	7	?	X	-	-	-	X	?
YAP	5	?	X	-	X	?	X	-

Os dados da Tabela 3 apresentam 6 programas com bons resultados na qualidade dos resultados (característica 3). Porém nenhum dos programas classificados com bons resultados são *open-source* e podem ser executados localmente, características necessárias para uso com sucesso em outros projetos.

Para atender aos requisitos do projeto relacionado a esta pesquisa é preciso que o programa possa ser baixado e usado sem nenhum impedimento legal, e para isso é necessário que as características (7) e (8) existam. Esse cenário motiva a criação de uma nova ferramenta.

Além disso, os testes apresentados mostraram que conforme são simulados plágios mais bem elaborados, os programas vão se mostrando insuficientes para identificá-los [Martins et al. 2014]. O programa *MOSS*, por exemplo, se esforça muito para tratar os falsos-positivos e tem muitos problemas com diferentes tipos de plágio. O *Sherlock* tem problemas com comentários nos códigos-fonte por não ignorá-los. Já o *CodeMatch* tem problemas com mudança de identificadores nos códigos-fonte.

O problema em lidar com a mudança de nomes de identificadores na ferramenta *CodeMatch* poderia ser abordado usando normalização, ao remover os nomes e usos das

variáveis nas submissões. Mas isso faria os códigos-fontes de submissões se tornarem mais parecidos, recaindo no problema de gerar muitos falsos positivos, casos em que o programa classifica como plágio situações que não são.

Kleiman (2007) identifica, a partir de testes com ferramentas, uma vulnerabilidade no pré-processamento, que a depender da estratégia usada pode ignorar grande parte da estrutura do programa ou deixar de descartar partes sem importância. Foi proposto um pré-processamento que envolve ordenação lexicográfica do programa. Essa ordenação é feita na árvore sintática do programa de maneira recursiva em uma abordagem de baixo para cima seguindo convenções. Após essa ordenação é gerada uma sequência de átomos e a árvore é destruída.

Essa estratégia foi proposta após encontrar problemas na primeira abordagem, que em um primeiro momento traduzia cada átomo do programa em caracteres ASCII a partir da análise léxica e ignorava nomes de identificadores, e em outro momento houve a preocupação de relacionar cada caractere gerado pela análise léxica com uma posição no programa original.

Entre os algoritmos explorados, o *RKS-GST* foi o que teve melhores resultados.

Com o objetivo de validar as ferramentas e a solução proposta, foi feito um levantamento de bases disponíveis de programas classificados que pudessem ser usadas. Infelizmente, não encontramos nenhuma, fator este que pode ser encarado como um desafio para pesquisas nesta área.

3. Estratégia proposta

O desenvolvimento de um mecanismo para a detecção de plágio insere-se no contexto de um sistema, denominado *PROBOCA*, que visa adaptar o juiz online *BOCA*, usado na Maratona de Programação, para facilitar seu uso em disciplinas de Introdução à Programação de Computadores. No BOCA, os professores definem listas de exercícios, que são resolvidos pelos alunos usando a linguagem C, e submetidos online. As soluções são corrigidas automaticamente melhorando a produtividade do professor e trazendo benefícios qualitativos ao aluno. No entanto, por não ter sido desenvolvido com o objetivo de servir de apoio para o ensino de programação, problemas como gerenciamento de turmas, geração de listas de exercícios, acompanhamento de alunos e verificação de plágio não são tratados. O sistema *PROBOCA* expande o sistema *BOCA* para tratar essas questões. Este artigo foca a detecção de plágio no sistema *PROBOCA*.

Por se tratar de um sistema voltado para a disciplina de Introdução à Programação de Computadores, os problemas definidos pelos professores são inicialmente simples, com soluções compostas por poucas linhas de código, e.g. problemas com atribuições e expressões aritméticas, e finalizam em problemas um pouco mais difíceis, e.g. envolvendo matrizes. Atualmente o sistema conta com um banco de dados com 100 problemas abordando os seguintes tópicos da disciplina: entrada de dados, saída de dados, estruturas de seleção, estruturas de repetição, cadeia de caracteres (*string*), estruturas de dados (vetores), uso de funções pré-definidas, definição de funções, passagem de parâmetros, domínio de lógica de programação, estrutura de dados (matriz), matemática. Esses problemas foram extensamente usados em sala de aula, gerando um banco de soluções que está sendo usado para validar o sistema proposto. Atualmente o banco de dados possui 19.613 registros de soluções (programas submetidos).

Ao longo dos semestres de uso do *BOCA* como ferramenta de ensino, foi possível verificar que muitos alunos copiam as soluções que submetem. Ou da internet ou de outros alunos. Ao fazer o plágio, os alunos executam diversas ações, como: mudar a indentação do programa, mudar os nomes das variáveis, mudar o escopo das variáveis, adicionar variáveis, adicionar comentários, adicionar bibliotecas. A constatação da extensão em que o plágio é adotado pelos alunos motivou a inclusão de um mecanismo para detectar esta prática.

Inicialmente tentou-se integrar um sistema já existente de detecção de plágio. Mas devido ao fato do único programa encontrado, *Sherlock N-Overlap*, segundo a literatura, com as características de (a) ter bons resultados, (b) ser *open source*, e (c) ter a possibilidade de ser executado localmente, ter sido testado para o contexto desta pesquisa sem apresentar bons resultados e necessitar de configuração específica para cada programa, tornou-se necessário desenvolver uma estratégia para resolver o problema.

Apesar da maioria dos sistemas identificados usarem o algoritmo *Greedy-String-Tiling*, que procura *substrings* que não se sobrepõem sem considerar a ordenação geral da *string* principal, nossa proposta, adaptada aos requisitos da disciplina de Introdução à Programação de Computadores, trabalha com o algoritmo Distância de Edição. O fato dessa disciplina ter programas menores influenciou na escolha do algoritmo Distância de Edição, pois ele não despreza a sequência do programa como um todo, como faz o *Greedy-String-Tiling*. Acreditamos que o algoritmo *Greedy-String-Tiling* seja mais útil para apoiar a identificação de plágio em programas maiores, porém é necessário realizar experimentos que comprovem.

A estratégia proposta utiliza o algoritmo de Distância de Edição, *Levenshtein Distance* [Gilleland 2006], juntamente com a normalização feita no pré-processamento. O algoritmo Distância de Edição foi desenvolvido pelo cientista Russo Vladimir Levenshtein em 1965, e tem aplicação em verificação ortográfica, reconhecimento de fala, análise de DNA e detecção de plágio. Ele mede a distância entre duas *strings*, o que possibilita calcular a similaridade entre elas.

Para calcular a distância entre duas *strings*, o algoritmo faz uma contagem da quantidade de operações necessárias (inserção, deleção, modificação) para que uma *string* se iguale à outra [Gilleland 2006]. A distância entre as *strings* "carro"e "cartas", é 3, porque para transformar "carro"em "cartas"é preciso fazer 2 modificações e 1 inserção. Já a distância entre as *strings* "computador"e "computador"é 0.

O algoritmo Distância de Edição é composto de sete passos, descritos no Algoritmo 1. O algoritmo recebe como argumento duas *strings* referenciadas como s e t.

```
Data: s, t
Passo 1: Defina n com o tamanho de s;
Passo 1: Defina m com o tamanho de t;
Passo 1: Se n=0 , retorne m e saia;
Passo 1: Se m=0 , retorne n e saia;
Passo 1: Construa uma matriz contendo m linhas e n colunas;
Passo 2: Inicialize a primeira linha na posição 0 de n;
Passo 2: Inicialize a primeira coluna na posição 0 de m;
for Passo 3: Examine cada caractere de s (i de 1 a n) do
    for Passo 4: Examine cada caractere de t (j de 1 a m) do
        if Passo 5: s[i] == t[j] then
            custo = 0;
        else
            custo = 1;
        end
        Passo 6: a = A célula imediatamente acima + 1: d[i-1,j] + 1;
        Passo 6: b = A célula imediatamente à esquerda + 1: d[i,j-1] + 1;
        Passo 6: c = A célula na diagonal acima e à esquerda mais o custo:
        d[i-1,j-1] + custo;
        Passo 6: A célula d[i,j] da matriz recebe o menor valor entre a, b, c;
    end
end
Result: Passo 7: A distância é o valor contido na célula d[n,m]
```

Algorithm 1: Passos do algoritmo Distância de Edição [Gilleland 2006]

O passo 1 objetiva construir uma matriz cujo tamanho se refere ao tamanho das duas *strings* de entrada. O passo 2 posiciona um cursor na primeira posição da matriz para percorrer toda a matriz. Os passos 3 e 4 fazem com que o programa percorra todas as posições da matriz para preenchê-las conforme os passos que antecedem o passo 7. O passo 5, para cada iteração, faz comparações cruzadas entre o valor referente à posição da linha da matriz em uma das *strings* com o valor referente à posição da coluna da matriz na outra *string*. Após isso, define-se um valor denominado custo que se refere ao custo de alteração. O passo 6 preenche o valor da posição visitada com o menor valor entre as somas de cada um dos vizinhos (posição anterior, posição superior, posição na diagonal anterior) mais próximo da posição visitada com o custo calculado no passo anterior. Por fim o passo 7 retorna o valor da última posição da matriz como a distância entre as duas *strings*.

De maneira resumida, pode-se dizer que o Algoritmo 1 calcula o número de operações para transformar uma *string* na outra, para isso ele organiza os dados do processamento em uma matriz que acumula o resultado em uma perspectiva diagonal.

A Figura 1 ilustra os passos percorridos pelo Algoritmo 1. O exemplo mostra a última célula com o valor referente ao número de operações necessárias para transformar a *string* "GUMBO"na *string* "GAMBOL", que implica na modificação do caractere "U"para "A"e da inserção do caractere "L"na primeira *string*.

Uma implementação do algoritmo Distância de Edição com programação dinâmica [ALISSON nd] apresenta a complexidade de tempo de O(|s1| . |s2|), sendo

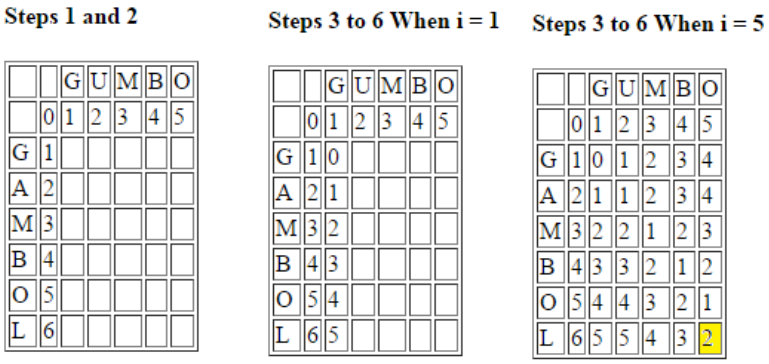

Figura 1. Ilustração dos passos percorridos pelo algoritmo Distância de Edição [Gilleland 2006]

s1 e *s2* as duas *strings* de entrada. Ao considerar que as *strings* sejam de tamanhos iguais, a título de análise, a complexidade de tempo é O(n^2). A complexidade de espaço é a mesma da de tempo apresentada, porém é possível implementar o algoritmo reduzindo a complexidade de espaço para O(|s1|) ou O(n).

Visto que o juiz online *BOCA*, e sua extensão *PROBOCA*, foram desenvolvidos em *PHP*, isso motivou o uso da implementação do algoritmo Distância de Edição disponível no endereço `http://php.net/manual/pt_BR/function.levenshtein.php`.

O algoritmo recebe como entrada duas *strings* normalizadas geradas a partir de dois arquivos de texto e retorna o número de operações necessárias para transformar uma *string* na outra, i.e a distância. Como a detecção de plágio tenta verificar a similaridade entre os códigos, considera-se que quanto menor for a distância entre o par de *strings* mais eles são similares. Neste contexto foi necessário criar uma equação para transformar a distância retornada pelo algoritmo para um percentual de similaridade.

$$maiorSubmissao = max(tamanhoSubmis1, tamanhoSubmis2)$$

$$similaridade = 100 - (\frac{distancia}{maiorSubmissao} * 100)$$

A variável *similaridade* recebe a transformação do valor retornado pela Distância de Edição entre as duas *strings* em um percentual de similaridade. Para isso, é utilizado o quociente da Distância de Edição pelo número de caracteres da maior string normalizada, armazenada em *maiorSubmissao*.

Como mencionado anteriormente, o algoritmo recebe strings normalizadas. A normalização visa remover dos códigos-fonte informações irrelevantes quanto ao plágio e padronizá-los. Para isso foram removidos: declarações de bibliotecas, comentários, caracteres entre aspas e espaços em branco.

Inicialmente também tentou-se remover identificadores das variáveis como parte do pré-processamento. Essa ação foi implementada e os testes não trouxeram bons resultados, fazendo com que os programas ficassem naturalmente mais parecidos e acabou gerando muitos falsos-positivos.

A experiência corrobora a ideia da forte influência do pré-processamento

na análise de plágio [Kleiman 2007, Maciel 2014]. Kleiman (2007) verificou a vulnerabilidade, no pré-processamento, quanto a remoção de partes do programa, que se não for bem planejada pode remover partes importantes ou deixar de descartar partes insignificantes.

Essa influência do pré-processamento, associada ao fato de que a disciplina de Introdução à Programação de Computadores trabalha muito com problemas simples e que os alunos que fazem plágio dificilmente teriam conhecimento suficiente para fazer modificações estruturais avançadas foram levadas em consideração na proposta apresentada, definindo as ações levadas a cabo no pré-processamento.

A Tabela 4 faz uma comparação da estratégia proposta com as demais, apresentadas na seção anterior. A coluna "Pré-processamento"é associada com os dados da Tabela 1. Percebe-se que a proposta deste trabalho não detecta mudanças estruturais e tem um pré-processamento simplificado. Essa simplificação foi adotada para tratar a vulnerabilidade apresentada por Kleiman (2007).

Tabela 4. Comparação de estratégias usadas nos programas para análise de similaridade com a proposta

Programa	Detecta mudanças léxicais	Detecta mudanças estruturais	Pré-processamento
CodeMatch	SIM	NÃO	1, 2
CPD	SIM	NÃO	3
Jplag	SIM	NÃO	1, 4, 5, 6
Plaggie	SIM	NÃO	1, 4, 5, 6
Marble	SIM	SIM	1, 4, 7, 8, 9, 10
MOSS	SIM	NÃO	* Texto não apresenta
Sherlock	SIM	NÃO	* Não possui
Sherlock N-Overlap	SIM	NÃO	1, 4, 5, 8, 11, 12, 13, 14.
SIM	SIM	NÃO	* Texto não apresenta
YAP3	SIM	SIM	1, 16, 17, 18, 19, 15
SID	SIM	NÃO	3
PROBOCA	SIM	NÃO	1, 4, 8, 7

Vale ressaltar, que antes de utilizar o algoritmo de Distância de Edição, foi feita uma tentativa de usar o programa *Sherlock N-Overlap* [Maciel 2014]. Como o código-fonte não estava disponível, foram feitas as modificações no programa original *Sherlock* e as atividades de pré-processamento implementadas. Após isso, o programa foi testado e os resultados não foram satisfatórios. Foi verificado que o *Sherlock N-Overlap* possui diferentes configurações que precisam ser feitas para cada contexto e não foi identificada uma configuração que pudesse ser usada em todos os problemas. A configuração que poderia ser utilizada para o maior número de problemas atendia apenas 58% dos casos testados. Assim essa abordagem foi abandonada.

4. Validação e Discussão

A estratégia de análise de similaridade projetada foi testada a fim de validar o trabalho feito. O algoritmo de similaridade foi executado para todos os pares de submissões

corretas no banco de problemas do sistema *BOCA*. O processamento da análise de similaridade para as soluções corretas dos alunos gerou 596.234 registros, sendo que cada um contém o par de arquivos analisados e seu respectivo grau de similaridade.

Os problemas são organizados no banco de dados por assuntos e níveis de dificuldade. Foram criados grupos de assuntos para permitir identificar a qual grupo cada problema pertence. O grupo 1 possui os seguintes assuntos: entrada de dados, saída de dados, domínio de lógica de programação, e matemática. O grupo 2 possui estruturas de seleção. O grupo 3 possui estruturas de repetição. O grupo 4 possui: estruturas de dados (vetores), e cadeia de caracteres (*string*). O grupo 5 possui: uso de funções pré-definidas, definição de funções, passagem de parâmetros, e estrutura de dados (matriz). A dificuldade foi definida sendo de 1, fácil, a 3, difícil.

Para comparar os resultados obtidos pelo algoritmo com uma análise humana a fim de verificar a eficácia da proposta, foram feitos 2 testes. O primeiro teste foi feito com 50 duplas de submissões, sendo 10 duplas escolhidas de maneira aleatória para cada um dos 5 grupos de assuntos. A classificação manual resultou em 24 duplas similares e 26 não-similares.

O segundo teste foi feito com 200 duplas divididas por problemas. Foram escolhidos 2 problemas de cada grupo, sendo um de menor nível de dificuldade e outro de maior nível. Para cada um dos 10 problemas foram selecionadas aleatoriamente 20 duplas, sendo 10 com similaridade menor ou igual à 85% e as outras 10 com similaridade maior que 85%, totalizando 100 duplas similares e 100 não similares. Para esse teste também foi feita uma classificação de plágio manualmente para cada registro selecionado e os resultados foram comparados.

Os resultados dos testes, na Tabela 5, são divididos em número de acertos (AC), número de falsos negativos (FN) e número de falsos positivos (FP). Os falsos negativos são os casos de plágio que o sistema classifica de maneira errada como não sendo plágio. Já os falsos positivos são os casos que não são plágio e o sistema classifica-os como plágio. Os falsos positivos são críticos, pois podem levar o professor a cometer injustiças com os alunos.

Tabela 5. Testes da estratégia proposta com dados reais classificados manualmente

	AC	AC %	FN	FN %	FP	FP %
Experimento com 50 duplas (> 75%)	46	92,00%	3	6,00%	1	2,00%
Experimento com 50 duplas (> 80%)	45	90,00%	4	8,00%	1	2,00%
Experimento com 50 duplas (> 85%)	44	88,00%	5	10,00%	1	2,00%
Experimento com 50 duplas (> 90%)	44	88,00%	6	12,00%	0	0,00%
Experimento com 50 duplas (> 95%)	44	88,00%	6	12,00%	0	0,00%
Experimento com 200 duplas (> 75%)	187	93,50%	6	3,00%	7	3,50%
Experimento com 200 duplas (> 80%)	188	94,00%	7	3,50%	5	2,50%
Experimento com 200 duplas (> 85%)	187	93,50%	8	4,00%	5	2,50%
Experimento com 200 duplas (> 90%)	171	85,50%	28	14,00%	1	0,50%
Experimento com 200 duplas (> 95%)	163	81,50%	37	18,50%	0	0,00%

A Tabela 5 mostra os resultados dos testes realizados considerando diferentes limiares para a classificação do plágio, variando de 75% a 95%. O limiar adotado, 85%, priorizou o balanceamento do número de acertos e do número de falsos-positivos. Observa-se que, conforme o limiar da similaridade diminui, o número de falsos-positivos

aumenta. Conforme o limiar cresce, o número de falsos-positivos tende a reduzir, porém quando o limiar é maior, 80% no caso do experimento com 200 duplas, o número de acertos tende a diminuir. No sentido de obter um compromisso entre aumentar o número de acertos e reduzir os falsos-positivos, adotou-se o limiar de 85%, que não traz os melhores resultados, mas é um valor adequado por ser um pouco acima dos que trazem os melhores resultados, assim, reduzindo os falsos-positivos. Esse limiar pode ter que ser reavaliado quando o sistema for colocado em uso efetivo, sem esquecer que o sistema tem como objetivo servir como um apoio na detecção de plágio continuando a exigir uma análise por parte do professor, já que não consegue determinar com total certeza se ocorreu plágio ou não.

Assim, talvez o mais interessante não seja uma análise pontual de plágio, mas uma tendência de plágio pelo aluno ou pela turma. Aquele aluno ou turma que obtém resultados que indicam plágio na maioria dos exercícios de uma lista merece a atenção do professor para verificar a real situação.

5. Análise de plágio no contexto de uma turma

Muitas pesquisas na área de informática e educação têm o interesse em propor ferramentas que possam apoiar o trabalho do professor visando melhoria no ensino. As ferramentas, além de serem capazes de automatizar diversos processos relacionados aos conteúdos e atividades, podem gerar dados para que o docente consiga obter conhecimento e tome as decisões mais acertadas visando qualidade. Essa linha de trabalho, aliada à questão do plágio no ensino de programação introdutória, encorajam a levantar dados que relacionem os resultados obtidos pela ferramenta proposta e informações sobre turmas de alunos.

Para isso, foi desenvolvido um relatório que resume as informações sobre possibilidades de plágio para alunos de turmas especificas, i.e. dada a turma o relatório é gerado. Em cada linha são apresentados os dados dos alunos (nome, curso, ano, semestre, aprovação, nota final) e a relação de problemas resolvidos contendo: a identificação do problema, número de submissões, número de acertos, grau de similaridade. O "grau de similaridade"contém o maior índice de similaridade da solução correta submetida pelo aluno comparada às demais soluções oriundas da mesma turma.

No fim de cada linha é feito um resumo por aluno que apresenta: número de submissões, número de acertos, média de acertos por submissões, menor similaridade encontrada, similaridade média, maior similaridade encontrada, número de submissões com similaridade <= a 85%, e número de submissões com similaridade > que 85%. Na base do relatório, após apresentar todas as linhas para o conjunto de alunos, é apresentado o mesmo resumo por problema da lista. A intenção desse relatório é oferecer ao professor uma visão do comportamento dos alunos sobre plágio a partir de uma análise detalhada dos dados.

Além do benefício aos professores, essa abordagem pode ser interessante para pesquisadores que querem analisar o comportamento dos alunos quanto ao plágio. Duas análises desse tipo foram feitas com os dados disponíveis.

A Tabela 6 apresenta os dados de quatro turmas: número de alunos, representação das submissões (número de submissões, número de acertos, percentual de acertos por submissões), e representação da similaridade (número de submissões com <= 85%

de similaridade, número de submissões com > 85% de similaridade, e percentual de submissões com > 85% de similaridade por acertos).

Tabela 6. Resumo de similaridade por turma considerando aspectos de submissões

Turma	Número de alunos	Número de Submissões	Número de Acertos	Percentual de acertos por submissões	Núm. Submissões com <= 85% de similaridade	Núm. Submissões com > 85% de similaridade	Percentual de Submissões com >85% de similaridade por acertos
A	44	3276	1619	49,42%	924	731	45,15%
B	37	5316	3605	67,81%	1459	2175	60,33%
C	42	3210	1623	50,56%	680	964	59,40%
D	22	1955	1346	68,85%	339	1021	75,85%

O campo "percentual de submissões com > 85% de similaridade por acertos"reflete o total de submissões corretas com um grau de similaridade que sugere plágio sobre o total de submissões corretas, visando representar o índice de plágio da turma. Ao ordenar os registros da tabela por esse campo obtém-se a sequência de turmas (A, C, B, D), sugerindo que houve mais casos de plágio na turma D, o que reflete a avaliação informal do professor.

Já o campo "percentual de acertos por submissões"representa para o conjunto de todas as submissões, tanto corretas quanto erradas, qual é o tamanho do subconjunto de submissões corretas representado por um percentual. Esse campo foi criado a partir da constatação de que, geralmente, os alunos que respondem os problemas sem fazer plágio costumam errar nas primeiras tentativas e após isso enviam a resposta correta. Nessa analogia, alto percentual indica maior possibilidade de plágio. Ao ordenar as turmas por esse campo, obtém-se (A, C, B, D).

Observa-se que o mesmo conjunto ordenado de turmas, (A, C, B, D), foi encontrado pelos dois campos analisados. Isso confirma a suposição de que quando os alunos enviam poucas submissões para cada problema, a possibilidade de estarem plagiando tende a crescer.

Também foi feita uma análise da correlação entre a nota do aluno e tendência de plágio, para entender se existe alguma relação entre plágio e notas ruins na disciplina. Para isso, calculou-se o número de submissões com similaridade > 85% proporcional ao número de acertos do aluno. Mais uma vez essa divisão pelo número de acertos foi para nivelar a quantidade de submissões tendenciosas a plágio pela participação do aluno considerando seus acertos.

Resultados relativos ao Coeficiente de Correlação de *Pearson* (Tabela 7) mostram que há uma correlação negativa significativa entre o "número de submissões com similaridade > 85% proporcional ao número de acertos do aluno"e as notas dos alunos nas turmas A, B e C. Assim, maior "número de submissões com similaridade > 85% proporcional ao número de acertos do aluno"está associada a menor nota.

A turma D apresentou uma correlação positiva não significativa, o que nos permite concluir que não há associação entre as variáveis nesse caso. Isto quer dizer que não se verificou correlação entre plágio e nota. Uma análise mais detalhada do contexto teria que ser feita para entender esta diferença de resultados. É importante lembrar que existem vários fatores que podem influenciar esse resultado, e.g. os alunos podem

Tabela 7. Correlações de notas com submissões similaridade para turmas

Turma	*r*	*p*
A	-.44	.003
B	-.35	.035
C	-.27	.082
D	.17	.462

"colar"durante as provas e o professor não consegue identificar, os exercícios podem não estar relacionados à prova, a nota final pode ter sido obtida por um cálculo que não leva em consideração os exercícios, entre outros.

Envolvendo plágio e o comportamento do aluno, as informações obtidas são importantes e devem ser consideradas nos requisitos de sistemas de apoio ao ensino de programação introdutória. Elas contribuem como apoio ao corpo docente por meio de alertas sobre o comportamento dos alunos. Além disso, manter um registro dessas informações permite criar um banco de conhecimento para a continuação de estudos científicos nesta área.

6. Conclusão

Com o intuito de incluir um sistema de detecção de plágio em uma ferramenta para a administração e correção automática de listas de exercícios a ser usada no ensino de programação introdutória que está sendo desenvolvida, foi feito um levantamento dos programas para a detecção de plágio disponíveis.

Identificar plágio em código-fonte é uma tarefa bastante desafiadora. Vários programas vêm sendo desenvolvidos para ajudar na identificação de plágio. A análise de similaridade apresentada foi feita considerando as etapas de pré-processamento, estratégias usadas para remover informações dispensáveis dos códigos-fonte a analisar; de processamento, definição do algoritmo usado para calcular a similaridade; e de pós-processamento, apresentação do resultado para o usuário.

No entanto, definir com precisão a eficácia dessas ferramentas depende de um banco de dados para teste com duplas previamente julgadas quanto à existência ou não do plágio e que reflete o contexto desejado. Códigos feitos em disciplinas de programação introdutória são diferentes de programas mais avançados por serem simples e com pouca flexibilidade de diferenciação. Logo, apesar do número de possibilidades, verificou-se que a análise e reutilização desses programas não é trivial.

Utilizar ferramentas disponibilizadas *on-line* apresenta alguns riscos. É preciso cautela visto que a ferramenta poder ser usada por muitos sistemas e usuários ao mesmo tempo, além de executar algoritmos com custos computacionais elevados. A ferramenta precisa atender a requisitos de escalabilidade rigorosamente estabelecidos e testados, caso contrário problemas de indisponibilidade afetariam a segurança. Apesar dessas restrições, tentativas foram feitas para usar os programas *jPlag* e *MOSS* remotamente mas sem sucesso.

O *Sherlock N-Overlap* foi o único programa *open-source* que apresentou bons resultados segundo a literatura e que possibilita sua execução localmente. Como sua

implementação não estava disponível foi necessário refazê-la. Infelizmente, quando testada esta solução não trouxe resultados satisfatórios, pois necessita de configuração específica para cada problema e não foi encontrada uma configuração que pudesse ser usada por um conjunto de problemas de programação introdutória.

Diante dessa situação foi preciso implementar uma solução. Optou-se por uma solução que usa o algoritmo Distância de Edição, que mede a distância entre duas *strings*. Apesar de vários sistemas usarem o algoritmo *Greedy-String-Tiling* em sua etapa de processamento, nossa proposta optou pelo algoritmo Distância de Edição, pelo fato do algoritmo preservar a ordenação dos caracteres das *strings* comparadas como um todo, o que acreditamos ser importante já que o conteúdo de programação introdutória possui um conjunto de comandos muito limitado e os programas tendem a ser pequenos. É diferente do *Greedy-String-Tiling*, que ao procurar *substrings* que não se sobrepõem, cria sub-conjuntos de *strings* e com isso a preservação da ordenação está limitada a cada sub-conjunto.

A etapa de pré-processamento, que trabalha com a normalização textual dos códigos-fonte, tem forte influência na qualidade da análise de similaridade. O algoritmo implementado buscou balancear a quantidade de informação removida dos códigos fonte, a fim de manter a identidade dos códigos mas ao mesmo tempo permitindo uma comparação. Os programas são normalizados por meio de remoção de declarações de bibliotecas, comentários, caracteres entre aspas e espaços em branco. Inicialmente também tentou-se remover identificadores das variáveis como parte do pré-processamento, mas essa normalização foi abandonada visto que ela simplifica ainda mais os programas, que já são simples, gerando muitos falsos-positivos.

Vale observar que esse conjunto reduzido de normalizações é consequência direta do contexto já que mudanças significativas no código-fonte exige conhecimentos de programação mais avançados que alunos novatos não possuem. Se eles tivessem conhecimento suficiente para fazer essas mudanças, eles provavelmente teriam capacidade de desenvolver os problemas introdutórios de programação. Portanto, optou-se por não abordar modificações estruturais. A estratégia adotada apresentou bons resultados dentro do contexto desejado.

Esta solução é bem adaptada à detecção de plágio em códigos de programação introdutória ao levar em consideração o tipo de problema e as estratégias de plágio adotadas por alunos iniciantes com uma fase de pré-processamento adequada, que faz remoções no código-fonte de maneira equilibrada. Além disso, trabalha com comparação de *strings*, que considera a sequência do programa como um todo, evitando processos mais complexos como a definição de *tokens*. É uma solução de fácil implementação, o que permite seu uso em sistemas com os mesmos requisitos sem dificuldade. Não é necessário fazer configurações específicas para cada problema, como é feito no *Sherlock N-Overlap*, pois trata-se de um mecanismo único para usar em diferentes problemas da disciplina em questão.

Para validar a proposta foram realizados testes usando submissões reais, oriundas de interações de alunos com o juiz online *BOCA*. Conjuntos de pares dessas submissões foram analisados pelos pesquisadores e classificadas de acordo com sua percepção da existência de plágio ou não. Em comparação, a abordagem de validação

proposta por Martins [Martins et al. 2014] fez simulações que objetivaram reproduzir o comportamento dos alunos ao fazer plágio.

Outra questão interessante que foi abordada neste trabalho foi a análise do plágio em um contexto mais abrangente, como o comportamento de um aluno ou turma no contexto de uma lista de exercícios. Os resultados indicam que análises detalhadas dos dados ocorrendo em um intervalo de tempo podem trazer uma perspectiva diferente que permite uma visão mais global do aluno e da turma. Pode-se identificar alunos que praticam plágio de maneira consistente ou nível de plágio de uma turma. Essa compreensão do contexto permite que o professor possa tomar medidas para punir ou remediar essas situações.

Visto que o sistema armazena todas as submissões das várias turmas que utilizaram o sistema ao longo do tempo, análises temporais tornam-se possíveis. Explorar as possibilidades apresentadas por esse tipo de análise desponta como uma boa opção de pesquisa.

Referências

Ahtiainen, A., Surakka, S., and Rahikainen, M. (2006). Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 141–142. ACM.

ALISSON, L. (n.d.). Dynamic programming algorithm (dpa) for edit-distance. `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit` [Acessado em 28 de dezembro de 2015].

Bakshi, T. (2010). Computer science tops in academic violations last year. the brown daily herald. 1 november 2010. `http://www.browndailyherald.com/mobile/computer-science-tops-in-academic-violations-last-year-1.2388353` [Acessado em 20 de janeiro de 2016].

Baugh, J., Kovacs, P., and Davis, G. (2012). Does the computer programming student understand what constitutes plagiarism. *Issues in Information Systems*, 13(2):138–145.

Chen, X., Francia, B., Li, M., Mckinnon, B., and Seker, A. (2004). Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on*, 50(7):1545–1551.

Copeland, T. (2003). Detecting duplicate code with pmds cpd.

De Campos, C. P. and Ferreira, C. E. (2004). Boca: um sistema de apoio a competições de programação. In *Workshop de Educação em Computação*, pages 885–895.

Gilleland, M. (2006). Levenshtein distance, in three flavors. merriam park software. `http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Fall2006/Assignments/editdistance/Levenshtein%20Distance.htm` [Acessado em 28 de dezembro de 2015].

Goya, D., Barbosa, A. N., Okida, C., and Ruggiero, W. (2014). Método para análise pericial em código de software sob suspeita de plágio. *Actas de la 9ª Conferencia Ibérica de Sistemas y Tecnologías de Informacion. Barcelona, España.*

Grune, D. (2012). Sim(1). http://dickgrune.com/Programs/similarity_tester/sim.pdf [Acessado em 24 de janeiro de 2016].

Hage, J., Rademaker, P., and van Vugt, N. (2010). A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28.

Joy, M. and Luck, M. (1999). Plagiarism in programming assignments. *Education, IEEE Transactions on*, 42(2):129–133.

Kleiman, A. B. (2007). *Análise e comparaçao qualitativa de sistemas de detecçao de plágio em tarefas de programaçao*. PhD thesis, Instituto de Computaçao.

Maciel, D. L. (2014). Sherlock n-overlap: Normalização invasiva e coeficiente de sobreposição para análise de similaridade entre códigos-fonte em disciplinas de programação.

Martins, V. T., Fonte, D., Henriques, P. R., and da Cruz, D. (2014). Plagiarism detection: A tool survey and comparison. *3rd Symposium on Languages, Applications and Technologies (SLATE 14)*., page 4566.

Prechelt, L., Malpohl, G., and Philippsen, M. (2002). *J. UCS*, 8(11):1016.

SAFE. Codematch algorithms. http://www.safe-corp.biz/CodeMatch_algorithms.htm [Acessado em 21 de janeiro de 2016].

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM.

Sheard, J., Dick, M., Markham, S., Macdonald, I., and Walsh, M. (2002). Cheating and plagiarism: perceptions and practices of first year it students. In *ACM SIGCSE Bulletin*, volume 34, pages 183–187. ACM.

Sherlock (n.d.). Sherlock - the sherlock plagiarism detector. http://sydney.edu.au/engineering/it/~scilect/sherlock [Acessado em 28 de junho de 2015].

Wagner, N. R. (2000). Plagiarism by student programmers. http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html [Acessado em 20 de janeiro de 2016].

Wise, M. J. (1996). Yap3: improved detection of similarities in computer program and other texts. In *ACM SIGCSE Bulletin*, volume 28, pages 130–134. ACM.

Zeidman, R. (2006). Software source code correlation. In *Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on*, pages 383–392. IEEE.