UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

NATAN DOS SANTOS CAMARGOS

ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE EM DISCIPLINAS DE PROGRAMAÇÃO:

ADAPTAÇÕES DO ALGORITMO SHERLOCK PARA USO DOS COEFICIENTES DE SORENSEN-DICE E DE SOBREPOSIÇÃO

FORTALEZA

2013

NATAN DOS SANTOS CAMARGOS

ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE EM DISCIPLINAS DE PROGRAMAÇÃO:

ADAPTAÇÕES DO ALGORITMO SHERLOCK PARA USO DOS COEFICIENTES DE SORENSEN-DICE E DE SOBREPOSIÇÃO

Monografia apresentada ao Curso de Engenharia de Teleinformática do Departamento de Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial para obtenção do Título de Engenheiro de Teleinformática.

Orientador: Prof. Dr. José Marques Soares.

FORTALEZA

2013

NATAN DOS SANTOS CAMARGOS

ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE EM DISCIPLINAS DE PROGRAMAÇÃO:

ADAPTAÇÕES DO ALGORITMO SHERLOCK PARA USO DOS COEFICIENTES DE SORENSEN-DICE E DE SOBREPOSIÇÃO

Monografia apresentada ao Curso de Engenharia de Teleinformática do Departamento de Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial para obtenção do Título de Engenheiro de Teleinformática.

Orientador: Prof. Dr. José Marques Soares.

Aprovada em ___/___/_____.

BANCA EXAMINADORA

_____

Prof. Dr. José Marques Soares (Orientador)

Universidade Federal do Ceará (UFC)

_____

Prof. Dr. Giovanni Cordeiro Barroso

Universidade Federal do Ceará (UFC)

_____

Prof. Me. Wellington Wagner Ferreira Sarmento

Universidade Federal do Ceará (UFC)

A Deus.

Aos meus pais, Francisca e José.

AGRADECIMENTOS

A Deus, por ter sempre me guiado no caminho da verdade. Sem Ele, a conclusão deste trabalho teria sido impossível.

Aos meus familiares, pelo importante apoio incondicional que possibilitou ultrapassar todos os obstáculos para a conclusão dessa monografia.

Ao colega, Danilo, pelas reflexões, críticas e sugestões recebidas.

Ao meu orientador, Prof. Dr. José Marques, pela confiança e incentivo durante todo o processo de orientação acadêmica.

Aos demais professores do Departamento de Engenharia de Teleinformática, pelos conhecimentos transmitidos durante a minha graduação como Engenheiro.

“Tenha coragem de seguir o que seu coração e sua intuição dizem. Eles já sabem o que você realmente deseja. Todo resto é secundário.”

Steve Jobs

RESUMO

Este trabalho se contextualiza na detecção de plágio de códigos gerados por alunos em disciplinas introdutórias de programação. A detecção de plágio é realizada por uma ferramenta, denominada BOCA-LAB, que foi concebida a partir da adaptação do sistema BOCA, desenvolvido na USP, e integrado ao ambiente Moodle em uma arquitetura orientada a serviços em um trabalho de mestrado no Programa de Pós-Graduação em Engenharia de Teleinformática (PPGETI). Além de permitir a submissão de programas para compilação e validação remota, a ferramenta possui um módulo que permite identificar similaridades entre os programas desenvolvidos pelos alunos. Faz parte deste trabalho a implantação do BOCA-LAB e o acompanhamento da utilização em laboratórios de programação do curso de Engenharia de Teleinformática, visando o seu aperfeiçoamento a aspectos de operação e de interface. De maneira especial, o trabalho se concentrou no módulo de suporte à detecção de plágio ou de similaridade entre códigos, com o objetivo de aumentar a eficiência do mesmo bem como estendê-lo para uso em contextos de cunho mais pedagógico, como a análise da semelhança entre a solução empregada por diferentes alunos e/ou a modelos de código padronizados pelo professor.

Palavras-chaves: Detecção de plágio, Sherlock, laboratório de programação, avaliação.

ABSTRACT

This work is contextualized in plagiarism detection code generated by students in introductory programming courses. The plagiarism detection is performed by a tool called BOCA-LAB, which was designed from the adaptation of the BOCA system, developed at USP, and integrated with Moodle environment in a service oriented architecture in a master thesis in the Programa de Pós-Graduação em Engenharia de Teleinformática (PPGETI). Besides allowing the submission of programs for compilation and remote validation, the tool has a module that identifies similarities between the programs developed by the students. It makes part of this work the deployment of the BOCA-LAB and the attendance of the use in laboratories of programming in the course of Engenharia de Teleinformática, aiming at his improvement to aspects of operation and of interface. Besides, and in special way, the work was concentrated in the module of support to the detection of plagiarism or similarity between codes. The objective is to increase the efficiency of the same thing as well as to increase to spread out for use in contexts of more pedagogic hallmark, like the analysis of the similarity between the solution employed by the pupil and models of code standardized by the teacher.

Keywords: Plagiarism detection, Sherlock, laboratory programming, assessment.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

(BOCA) BOCA Online Contest Administrator

(MOODLE) Modular Object-Oriented Dynamic Learning Environment

(GST) Greedy-String-Tiling

(LCS) Longest Common Subsequence

(RKRGST) Running-Karp-Rabin Greedy-String-Tiling

(MOSS) Measure of Software Similarity

(HTML) HyperText Markup Language

(PHP) PHP Hypertext Preprocessor

SUMÁRIO

1. INTRODUÇÃO

O plágio em laboratórios de programação caracteriza-se pela cópia total ou parcial de soluções de colegas, frequentemente acompanhada de modificações para dificultar a percepção da mesma. Em turmas numerosas, essa prática muitas vezes não é identificada, devido à dificuldade inerente à detecção manual. Tal situação pode ser contornada com o uso de ferramentas que deem suporte à detecção de plágio.

Atualmente existem diversas ferramentas que oferecem suporte a detecção de plágio em códigos-fonte disponíveis. Dentre essas ferramentas, destaca-se o Sherlock (Pike, 2013), um software de código-aberto que se propõem a conciliar simplicidade e eficiência na detecção de plágio. A fácil integração do Sherlock com outros sistemas é um aspecto importante na ferramenta. Despois de compilado, é possível invocar o Sherlock via linha de comandos para comparar código-fonte armazenados localmente.

O Sherlock é parte fundamental no módulo de suporte detecção de plágio do sistema web BOCA-LAB (França e Soares, 2011), que foi integrado ao Moodle (Dougiamas e Taylor, 2003) e permite a submissão online das atividades de laboratório desenvolvidas nas disciplinas de programação. O BOCA-LAB compila e executa remotamente as soluções enviadas pelos alunos, efetuando, ainda, a verificação das saídas geradas pelos programas.

Este trabalho avalia especificamente a ferramenta Sherlock, e propõem alterações no seu comportamento, objetivando o aperfeiçoamento dos resultados na detecção de plágio. Adicionalmente são avaliadas técnicas de pré-processamento usando normalizações específicas, de modo a padronizar os códigos-fonte analisados antes da execução do Sherlock.

O presente trabalho está dividido em seis capítulos, incluindo-se a introdução aqui apresentada. O capítulo 2 contém uma fundamentação teórica a respeito da detecção de plágio em códigos-fonte. No capítulo 3, o Sherlock é descrito de forma detalhada para melhor compreensão das alterações propostas no capítulo seguinte. As alterações propostas para o Sherlock, e a adição de duas etapas ao processo de detecção de plágio do BOCA-LAB são apresentadas no capítulo 4. Os resultados obtidos com as alterações propostas são expostos no capítulo 5. Finalmente, o capítulo 6 apresenta as conclusões deste trabalho e as considerações finais acerca do trabalho realizado, bem como as sugestões para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. Principais abordagens

A contagem de atributos e a comparação de estruturas são as duas principais abordagens usadas nos sistemas de detecção de plágio em códigos-fonte. Sistemas baseados em contagem de atributos procuram e quantificam atributos escolhidos a priori em cada código analisado, sendo considerado indício de plágio a ocorrência de igual quantidade de atributos. De acordo com Kleiman (2007) e Cornic (2008), essa abordagem é eficiente apenas para casos em que ocorreram poucas tentativas de alteração.

Sistemas baseados em comparação de estruturas subdividem os códigos-fonte em pequenas estruturas, e essas são posteriormente comparadas usando métricas diversas. Essa abordagem torna possível a detecção plágio em trechos de código, algo que não ocorre na contagem de atributos. Em Verco e Wise (1996) é revelada a maior eficiência dos sistemas que usam comparação de estruturas em relação aos sistemas baseados em contagem de atributos.

2.2. História da detecção de plágio em códigos-fonte

Visando reduzir o trabalho associado à detecção de plágio em códigos-fonte, OTTENSTEIN (1976) descreveu um sistema capaz de inferir possíveis equivalências entre códigos-fonte escritos em FORTRAN. Esse sistema quantificava os códigos-fonte usando métricas de software definidas por Halstead (1972). Ottenstein defendeu que a probabilidade de que dois códigos-fonte diferentes possuíssem a mesma imagem em um espaço definido pelas métricas de Halstead fosse ínfima.

Conforme Verco e Wise (1996), os sistemas que se seguiram usaram essas métricas e outras como o número de laços e o número de procedimentos para encontrar medições de similaridade mais precisas entre pares de programas. Todos esses sistemas usavam a contagem de atributos como abordagem principal. Ainda segundo Verco e Wise, os sistemas de contagem de atributos tornaram-se bastante complexos, em alguns casos, chegando a incluir contagens representando estruturas de controle de fluxo, como no sistema de Faidhi e Robinson (1987).

Donaldson, Lancaster e Sposato (1981) criaram o primeiro sistema de detecção usando comparação de estruturas. Na verdade, o sistema era um híbrido das duas abordagens, pois adotava a contagem de atributos, e adicionalmente comparava estruturas que representavam o código-fonte para melhorar a detecção de similaridade. Os sistemas de detecção mais recentes usam, em sua maioria, a comparação de estruturas.

Whale (1990, a*pud* Cornic, 2008) desenvolveu um software chamado Plague. O sistema proposto gera perfis dos programas de entrada. Os perfis são sequências de *tokens* compostos de informações estruturais. Os perfis de estrutura semelhante são combinados e suas sequências de *tokens* comparadas para encontrar subsequências comuns.

Em 1993, Wise (1993, a*pud* Cornic, 2008) criou o algoritmo *Greedy-String-Tiling* (GST) usado para combinar sequências de *tokens*. Esse algoritmo foi usado na ferramenta YAP3 que também foi desenvolvida pelo autor. A ferramenta ainda encontra-se em uso.

Prechelt, Malpohl e Phlippsen (2000) apresentaram um sistema chamado JPlag. A ferramenta transforma os programas em sequências de *tokens* e compara as sequências usando o algoritmo *Running-Karp-Rabin Greedy-String-Tiling* (RKRGST). É disponibilizada exclusivamente por meio de um *WebService* e possui código fechado, e requer o cadastramento e a requisição de autorização para o acesso ao serviço.

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

2.3. Algoritmos usados na detecção de plágio em códigos-fonte

Em (Kleiman, 2007) são apresentados alguns algoritmos para a detecção de plágio em códigos-fonte, os principais são: o *Winnowing*, o *Greedy String Tiling*[1] e o *Running Karp-Rabin Greedy String Tiling*[2]. Esses algoritmos são descritos nas próximas seções.

2.3.1. Winnowing

O winnowing tem como objetivo melhorar a eficiência do processo de comparação de documentos com base em assinatura única por documento. Segundo Schleimer *et. al* (2003), o algoritmo obtém uma assinatura para cada documento de forma que essa assinatura possa ser usada para identificá-lo e detectar similaridade.

Esse algoritmo utiliza o conceito de *k-gramas*. Conforme Kleiman (2007), os *k-gramas* de uma cadeia *S* são as sub cadeias de comprimento *k* contiguas e sobrepostas da cadeia *S*. Os 5-gramas da frase “o rato roeu a roupa" são: ‘orato’, ‘rato’, ‘atoro’, ‘toroe’, ‘oroeu’, ‘roeua’, ‘oeuar’, ‘uarou’, ‘aroup’ e ‘roupa’, por exemplo.

O processo de obtenção da assinatura inicia-se com a divisão do texto em *k-gramas*. Na sequência, cada *k-grama* é representado por um valor numérico, e, por fim, um subconjunto desses valores é obtido da peneiração do superconjunto de todos os valores para ser a assinatura do documento.

2.3.2. Greedy String Tiling

O algoritmo GST foi primeiramente descrito por Wise (1993). De acordo com Cornic (2008), o algoritmo compara duas cadeias de caracteres para determinar o seu grau de similaridade. A capacidade de lidar com permutações é o diferencial do algoritmo. Antes de descrever o algoritmo é necessário definir alguns termos que serão usados. A cadeia de caracteres padrão ou simplesmente cadeia padrão é uma referência à menor das cadeias de caracteres comparados, enquanto que a maior é

[1] *GST.*
[2] *RKRGST.*

referenciada por cadeia texto. Sendo *P* a cadeia padrão e *T* a cadeia texto, Wise (1993) descreve o algoritmo da seguinte forma:

> Um casamento-máximo acontece quando uma subcadeia P_p de uma cadeia padrão iniciada em *p*, casa perfeitamente, elemento por elemento, com uma subcadeia T_t da cadeia texto iniciada em *t*. Considera-se o casamento mais longo possível, isto é, até que um elemento que não casa ou o final da cadeia seja encontrado, ou até que um dos elementos encontrados esteja marcado. Casamentos-máximos são temporários e, possivelmente não são associações únicas, isto é, uma subcadeia envolvida em um casamento-máximo pode ser parte de muitos outros casamentos-máximos.
> Um ladrilho é uma associação (um para um) permanente e única de uma subcadeia de *P* com uma subcadeia que casa em *T*. Para se formar um ladrilho a partir de um casamento-máximo, os *tokens* das duas subcadeias são marcados e, posteriormente, tornam-se indisponíveis para casamentos futuros.

Adicionamento foi definido um valor mínimo para tamanho de casamento, de modo a estabelecer um limite inferior para o tamanho do casamento-máximo. Esse valor é destinado a melhorar a eficiência do algoritmo, eliminando casamentos com tamanho abaixo desse limite inferior.

O algoritmo tem como objetivo encontrar um conjunto de ladrilhos que maximize a cobertura sobre *T* através de *P*. O GST é baseado na ideia de que casamentos longos são mais interessantes do que os curtos, porque são mais prováveis de representar semelhanças entre as cadeias em vez de coincidências.

O algoritmo executa múltiplas passagens nos dados, cada um deles é composto de duas fases. Na primeira fase, os casamentos-máximos acima de um certo comprimento são coletados e armazenados em listas, conforme seus comprimentos.

A segunda fase constrói ladrilhos com casamento-máximo da primeira fase, começando com a mais longa. Para cada casamento, o algoritmo testa se ele está marcado. Se não, um ladrilho é criado com este casamento e os textos correspondentes em *P* e *T* são marcados. Quando os casamentos de comprimento considerado forem tratados, um comprimento menor é escolhido e começa novamente a busca da primeira fase. O algoritmo para quando o texto completo estiver marcado e, em seu pior caso, possui a complexidade $O(n^3)$.

2.3.3. Running Karp-Rabin Greedy String Tiling

O algoritmo *Karp-Rabin* foi criado por Richard M. Karp e Michael O. Rabin em 1987, com o objetivo de encontrar casamentos em cadeias de caracteres. Ele utiliza assinaturas para encontrar ocorrências de uma cadeia em outras. A principal ideia do *Karp-Rabin* é usar uma função *hash* que pode rapidamente calcular a assinatura do $i_{ésimo}$ k-grama a partir da assinatura do $(i_{ésimo} - 1)$ k-grama. Se um k-grama $c_1 \dots c_k$ é representado por um número de *k-dígitos* em uma base *b* qualquer, a assinatura $H(c_1 \dots c_k)$ desse k-grama é:

$$c_1 \times b^{k-1} + c_2 \times b^{k-2} + \cdots + c_{k-1} \times b + c_k \quad (1)$$

Para calcular a assinatura do k-grama $H(c_2 \dots c_{k+1})$ a partir de (1), basta subtrair o elemento mais significativo, multiplicar o número por *b* e somar c_{k+1}. Dessa forma, tem-se a seguinte identidade:

$$H(c_2 \dots c_{k+1}) = (H(c_1 \dots c_k) - c_1 \times b^{k-1}) \times b + c_{k+1} \quad (2)$$

Este método permite encontrar as assinaturas para os k-gramas de um documento em tempo linear. O algoritmo calcula a assinatura da cadeia padrão e compara com as assinatura dos k-gramas cadeia texto.

Wise (1993) aplicou a ideia do algoritmo *Karp-Rabin* ao algoritmo GST e criou o algoritmo *Running Karp-Rabin Greedy String Tiling* (RKRGST). Este algoritmo é um dos mais usados nos sistemas detectores de plágio, sua complexidade no pior caso, segundo Wise (1993), é $O(n^3)$.

De acordo com Cornic (2008), algumas adaptações foram feitas para a criação do RKRGST: O valor da assinatura é calculado para os k-gramas desmarcados da cadeia padrão, em vez de apenas um valor para o padrão. O mesmo é feito para os k-gramas desmarcados da cadeia texto. Os valores das assinaturas de cada k-grama da cadeia padrão *P* são comparados com os valores da assinatura da cadeia texto *T*. Para reduzir a complexidade, é criada uma *hash-table* de assinaturas. A busca na tabela retorna as posições dos k-gramas com o mesmo valor. Depois que uma ocorrência for encontrada, o algoritmo tenta estender o casamento, símbolo por símbolo. Após cada iteração, o comprimento da cadeia pesquisada (k) é reduzido até o casamento de tamanho mínimo.

2.4. Ferramentas para detecção de plágio em códigos-fonte

As duas ferramentas para detecção de plágio em códigos-fonte mais citadas são apresentadas nas seções seguintes.

2.4.1. JPlag

A ferramenta JPlag (Prechelt, 2002) opera em duas fases:

- Antes da comparação efetiva, os códigos-fonte passam por uma análise sintática[3] (dependendo da linguagem), e são convertidos em cadeias de *tokens*.
- Essas cadeias de *tokens* são comparadas em pares para determinar a similaridade de cada par. O algoritmo usado é basicamente o *Greedy String Tiling* (Wise, 1993). Durante cada comparação, o JPlag tenta cobrir uma cadeia de *tokens* com subcadeias (ladrilhos) advindas de outro código-fonte. A similaridade é igual ao percentual de cadeias de *tokens* cobertos. Os resultados do JPlag são exibidos em páginas HTML.

2.4.1.1. Convertendo códigos-fonte em cadeias de tokens.

A etapa de conversão dos códigos-fonte em cadeias de *tokens* é a única com dependência de linguagem no JPlag. Isso porque apenas as linguagens Java e Scheme passam por um analisador sintático, enquanto que C++ e C passam apenas por um analisador léxico.

Como regra, os *tokens* devem ser escolhidos de forma que a essência da estrutura dos códigos-fonte seja caracterizada, ao invés de aspectos superficiais, que são mais simples de se alterar. O JPlag ignora espaços em branco, comentários e nomes de identificadores. Entretanto, a ferramenta insere alguma informação semântica nos *tokens* sempre que possível, de forma a reduzir falsos casamentos de subcadeias que podem ocorrer por puro acaso.

[3] Também conhecida pelo termo em inglês ***parsing***.

2.4.1.2. Comparando duas cadeias de tokens.

O algoritmo usado para comparar das cadeias de *tokens* é essencialmente o *GST*. Ao comparar duas cadeias *A* e *B*, o foco é achar o máximo conjunto de subcadeias contíguas que tenham as seguintes propriedades: cada subcadeia que ocorre em *A* e *B*, é a mais longa possível e não cobre um *token* de alguma outra subcadeia. Para evitar falsos casamentos, um tamanho mínimo de casamento *M* é aplicado.

O GST é um algoritmo heurístico, porque ao garantir o casamento máximo para o conjunto de subcadeias achadas, a busca se torna demasiadamente dispendiosa. O pseudocódigo do GST é apresentado no anexo A. O algoritmo é sucintamente descrito por Prechelt (2002) em dois passos que se repetem a cada iteração:

> **Passo 1:** As duas cadeias são vasculhadas na procura pelo maior casamento contínuo. Tecnicamente, isso é feito por 3 laços aninhados: o primeiro itera através de todos os *tokens* em *A*, o segundo compara o *token* de *A* com cada *token* em *B*. Se um par de *tokens* casa, o laço mais interno procura pelo fim desse casamento. Esses laços aninhados coletam o conjunto de todas as maiores subcadeias comuns.
>
> **Passo 2:** Marca todos os casamentos de tamanho máximo não sobrepostos do passo 1. Isso significa que todos os *tokens* são marcados, e, portanto, não podem ser usados para mais casamentos no passo 1 em uma iteração subsequente. Na terminologia de Wise, os *tokens* marcados nesse passo se tornam ladrilhos.

Esses dois passos são repetidos até que nenhum casamento seja encontrado. Como o número de *tokens* marcados em cada passo aumenta gradualmente, o algoritmo sempre termina. O resultado do algoritmo é uma lista de ladrilhos que é usada para calcular a similaridade, que deve refletir a fração dos *tokens* dos códigos originais que são cobertos por casamentos.

A similaridade do JPlag é obtida da função $sim(A, B)$, definida pela seguinte equação:

$$sim(A, B) = \frac{2 \times cobertura(ladrilhos)}{|A| + |B|} \quad (3)$$

Onde $cobertura(ladrilhos)$ é obtida do somatório abaixo ($|l|$ é o comprimento do ladrilho):

$$cobertura = \sum_{l \in ladrilhos} |l| \quad (4)$$

2.4.2. MOSS

O MOSS (*Measuse of Software Similarity*) recebe como entrada um conjunto de documentos enviados por meio de um *script* fornecido pelo autor. A ferramenta retorna um série de páginas HTML indicando onde seções significantes dos pares de documentos analisados são semelhantes. O MOSS é majoritariamente usado para a detecção de plágio em tarefas de programação, embora diversos outros tipos de texto sejam suportados. De acordo com Schleimer (2003), o algoritmo utilizado é o *Winnowing* robusto, mas detalhes da implementação não são revelados.

Assim como em várias outras ferramentas para detecção de plágio em códigos-fonte, um pré-processamento é aplicado para reduzir ou eliminar a informação irrelevante nos códigos analisados antes da comparação efetiva usando um algoritmo especifico. No caso do MOSS, para cada formato de documento existe um pré-processamento particular que no geral elimina características que não distinguem os documentos, entretanto, poucos detalhes desse pré-processamento são informados.

Após a comparação efetiva dos códigos-fonte analisados, os resultados são exibidos em páginas HTML, de maneira que os casamentos encontrados nos códigos sejam destacados visualmente, facilitando uma comprovação visual.

3. OBJETO DE ESTUDO: SHERLOCK

A versão original do Sherlock era formada por dois programas, *sig* e *comp* e foi criada por Rob Pike, engenheiro de software canadense que trabalhou na Bell Labs e, mais recentemente, na Google. O programa *sig* gerava assinaturas digitais que representavam cada código-fonte analisado, e as armazenava em um arquivo. O programa *comp* usava esse arquivo de assinaturas para computar e exibir as similaridades. Posteriormente, Loki combinou os dois programas em um único, chamado Sherlock.

A assinatura digital é a representação de uma estrutura retirada do código, o que o enquadra o Sherlock na abordagem de comparação de estruturas. O processo de detecção da ferramenta pode ser resumido em duas etapas: geração de assinaturas e comparação de assinaturas. As seções seguintes descrevem melhor essas etapas.

O Sherlock apresenta uma série de vantagens em relação à outras ferramentas de mesmo propósito, pois:

- É capaz de detectar plagio em códigos-fonte escritos em diversas linguagens.
- Possui código aberto, permitindo alterações e melhorias.
- É multiplataforma, devido a adoção da linguagem C.
- Pode ser executada localmente.
- Tem baixo tempo de resposta, graças à baixa complexidade do algoritmo que implmenta.

Essa vantagens são contrabalanceadas por três desvantagens, são elas:

- Não possui interface gráfica.
- Pouca documentação acerta do seu funcionamento.
- Análise de similaridade limitada à elementos léxicos.

Entre as desvantagens, destaca-se a limitação do Sherlock à análise apenas de elementos léxicos. Essa limitação diminui o escopo de atuação da ferramenta, pois incapacita a detecção de similaridade em códigos que possuam equivalência semântica como na figura 1.

Figura 1: Equivalência semântica.

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

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

Outro problema associado à análise puramente léxica, é a utilização de elementos léxicos que não contém informação importante para a comparação. Códigos-fonte com identificadores de variáveis alterados constituem plágio, mas a utilização desses mesmos identificadores, elementos léxicos irrelevantes, na comparação de estruturas pode levar a resultados imprecisos. Elementos léxicos irrelevantes representam ruído para a análise de similaridade, analogamente ao ruído da Teoria da Informação.

Figura 2: Alteração de elementos léxicos de pouca importância.

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

```
// Soma os elementos de um vetor
int soma(int *vetor, int l) {
      int k, resultado = 0;
      for (k = 0; k < l; k++) {
            resultado += vetor[k];
      }
      return resultado;
}
```

3.1. Geração de assinaturas

O início dessa etapa se dá pela extração de características, que retira do código-fonte estruturas menores chamadas de *tokens*. Os espaços, as tabulações e as nova linhas delimitam o início e o fim de um *token*. O trecho de código da figura 3 tem os seus *tokens* destacados.

Figura 3: Trecho de código com *tokens* sublinhados.

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

Na sequência, *tokens* são mapeados em assinaturas por meio da seguinte função de espalhamento:

$$h(K) = K \; mod \; M \tag{5}$$

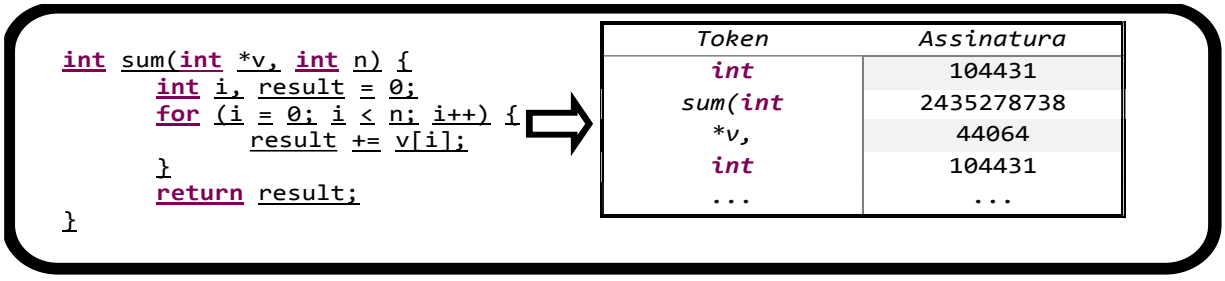
O inteiro *K*, que representa um *token*, é mapeado em um inteiro dentro do intervalo $[0\ M-1]$, onde $M = 2^{32}$. O inteiro *K* é obtido do somatório:

$$K = \sum_{i=0}^{n-1} t[i] \times 31^{n-1-i} \tag{6}$$

em que n é o número de caracteres no *token*, e $t[i]$ corresponde à representação ASCII do i-ésimo caractere.

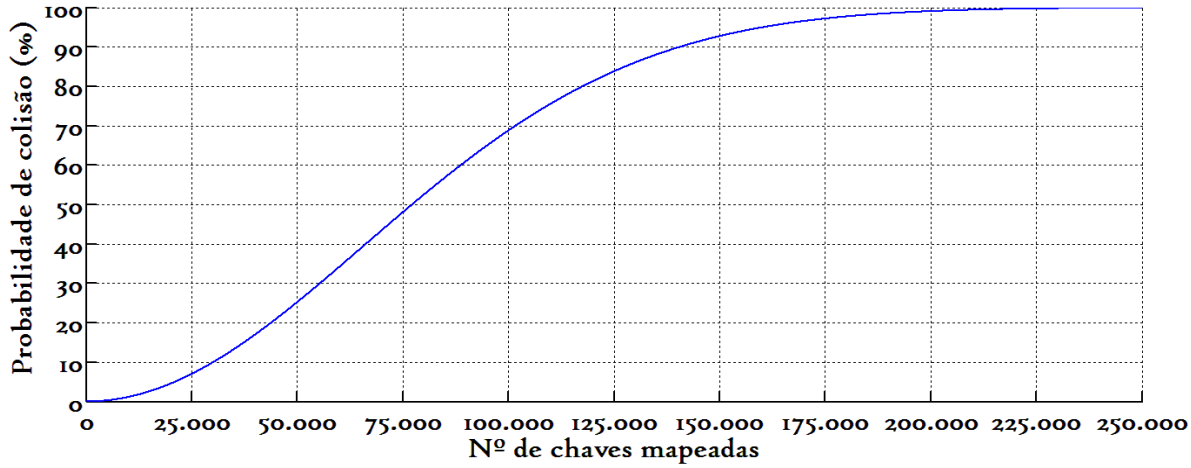
No final dessa etapa, o código-fonte é representado por um vetor de assinaturas como na ~~figura abaixo~~. A próxima etapa utilizará esses vetores de assinaturas para comparações.

Figura 4: Assinaturas geradas.



É importante ressaltar que algumas colisões podem ocorrer porque a função de espalhamento não é injetora, mas a probabilidade de que isso aconteça é pequena. Essa probabilidade *p* de se mapear *N* chaves (*tokens*) consecutivas sem colisão em um intervalo de tamanho *M* pode ser calculada segundo (Feller, 1968). A ocorrência de uma colisão significa que dois *tokens* foram mapeados em uma mesma assinatura, implicando em erros no resultado de similaridade.

Figura 5: Probabilidade de colisão por nº de *tokens* mapeados.



3.2. Comparação de assinaturas

Nessa etapa, as assinaturas que identificam um código-fonte são comparadas às assinaturas de outro até que todos os códigos analisados sejam comparados, exigindo um número de comparações igual à combinação sem repetição de n elementos 2 a 2.

$$C\binom{N}{2} = \frac{N \times (N-1)}{2} \tag{7}$$

A complexidade do Sherlock pode ser obtida em termos dessa etapa. A complexidade é $O(f(n)N^2)$ onde N é o número de arquivos, e $f(n)$ o tempo gasto na comparação de arquivos com tamanho n.

A similaridade é definida como a percentagem de semelhança entre os dois arquivos de texto comparados, e pode ser formulada da seguinte maneira:

$$l_1 = length(file_1) = |A| + |B|$$

$$l_2 = length(file_2) = |A| + |C|$$

Onde$|A|$ é o número de assinaturas similares encontradas em ambos os arquivos ($file_1$ e $file_2$), $|B|$ é o número de assinaturas encontradas exclusivamente em $file_1$ e $|C|$ o número de assinaturas encontradas exclusivamente em $file_2$.

A similaridade é dada por:

$$sim(A,B) = 100 \times \frac{|A|}{l_1 + l_2 - |A|} = 100 \times \frac{|A|}{|A| + |B| + |C|} \quad (8)$$

3.3. Parâmetros do Sherlock

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

- **Zero bits (z):** controla a granularidade da comparação. Quanto maior o número, mais crua será a comparação, porém mais rápida. Quanto menor o número, mais exata a comparação, porém mais lenta e isso pode dificultar a detecção de plágio, pois pequenas mudanças no texto serão percebidas pelo programa e não serão tratadas como semelhança. O valor padrão é 4, mas pode ser qualquer um no intervalo [0-31].

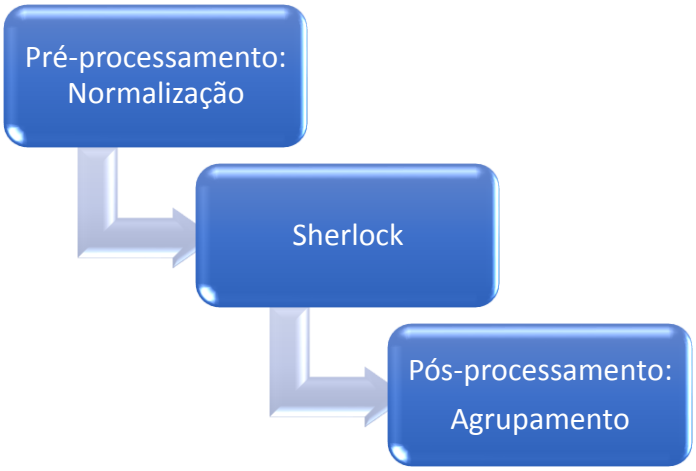
- **Number of words (n):** controla quantas palavras formam uma assinatura digital. Isto também contribui para a granularidade da comparação. Quanto maior o número, maior a exatidão. Entretanto, a comparação será mais lenta. O valor padrão é 3, que funciona bem em muitos casos.

- **Threshold (t):** controla o quanto similar devem ser os textos antes de serem mencionados no resultado final. O sucesso (ou fracasso) ao detectar o plágio com o Sherlock está intimamente ligado aos valores utilizados nestes parâmetros.

4. ALTERAÇÕES PROPOSTAS

As alterações propostas para o módulo de suporte à detecção de plágio do BOCA-LAB definem três etapas para a análise de similaridade dos códigos-fonte: Pré-processamento, execução do Sherlock e Pós-processamento.

Figura 6: Etapas da análise de similaridade de códigos-fonte.



A etapa de pré-processamento constitui-se da normalização dos códigos-fonte, juntamente com a retirada de elementos léxicos pouco relevantes para a execução do Sherlock. Maciel (2012) definiu 4 normalizações com melhorias de resultados para o algoritmo Sherlock. Essa etapa é essencial para a obtenção de resultados mais fidedignos, devido à sensibilidade do Sherlock. Comentários, identificadores, valores literais e outros elementos irrelevantes são retirados dos códigos-fonte nessa etapa.

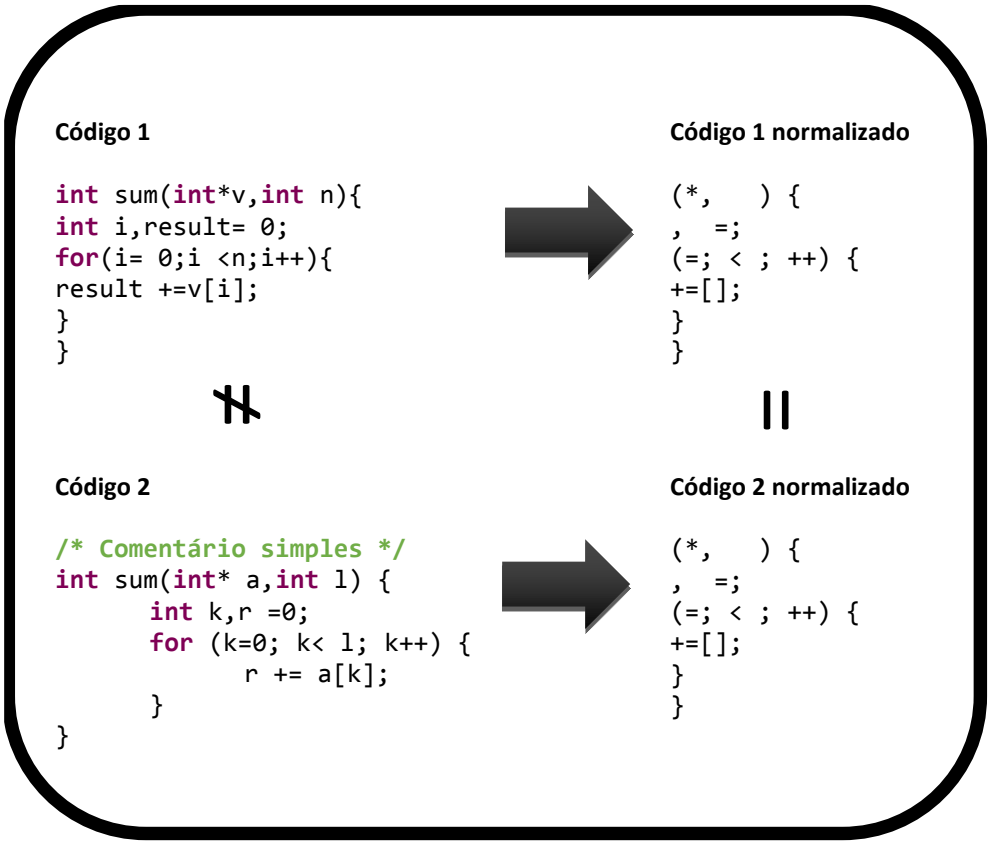
Após a etapa de pré-processamento, o Sherlock é executado usando os códigos resultantes da etapa anterior. O Sherlock executado nessa etapa usa medidas de similaridade que garantem uma maior confiabilidade aos resultados, principalmente para a adição de código nulo.

A etapa final realiza um agrupamento dos códigos-fonte semelhantes de modo a instrumentalizar o professor na identificação de soluções semelhantes. É uma etapa opcional, mas de grande relevância para propósitos pedagógicos. O professor pode identificar uma certa tendência nas resoluções dos problemas propostos, e tomar medidas adequadas para contornar certas dificuldades dos alunos, por exemplo.

4.1. Pré-processamento: Normalização dos códigos-fonte

A implementação dessa etapa foi desenvolvida em PHP para facilitar a integração com o módulo de suporte à detecção de plágio do BOCA-LAB. Essa implantação é majoritariamente formada por um conjunto de instruções regidas por expressões regulares para identificar e normalizar determinadas estruturas que possam influenciar os resultados do Sherlock. ~~A figura abaixo exibe~~ o resultado da normalização de dois códigos-fonte diferentes do ponto de vista do Sherlock.

Figura 7: Normalização de códigos-fonte.



A diferença observada antes da normalização é uma consequência da abordagem simplista do Sherlock na extração de características. Já no lado direito da figura, os códigos-fonte normalizados são idênticos para o Sherlock, constatando a importância da normalização na etapa de pré-processamento dos códigos-fonte.

4.2. Alterações no Sherlock

Como descrito anteriormente, as assinaturas são armazenadas em vetores no final da primeira etapa do Sherlock, indicando uma possível abordagem vetorial para o cálculo de similaridade, entretanto, tal possibilidade não se concretiza. Na verdade, o cálculo de similaridade do Sherlock é bastante semelhante ao coeficiente de Jaccard (Jaccard, 1901), diferindo apenas na natureza das entidades comparadas. O coeficiente de Jaccard é usado para calcular a similaridade entre dois conjuntos, baseando-se nas cardinalidades de interseção e união desses conjuntos.

Equação 1: Coeficiente ou índice de Jaccard.

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

Com o objetivo de aumentar a eficiência do Sherlock, esse trabalho propõem o uso de outras medidas de similaridade, e apresenta os resultados da implementação das mesmas.

Tabela 1: Medidas de similaridade propostas.

Coeficiente de Sorensen-Dice	*Coeficiente de Sobreposição*
$S(A,B) = \frac{2\|A \cap B\|}{\|A\| + \|B\|}$	$O(A,B) = \frac{\|A \cap B\|}{min(\|A\|, \|B\|)}$

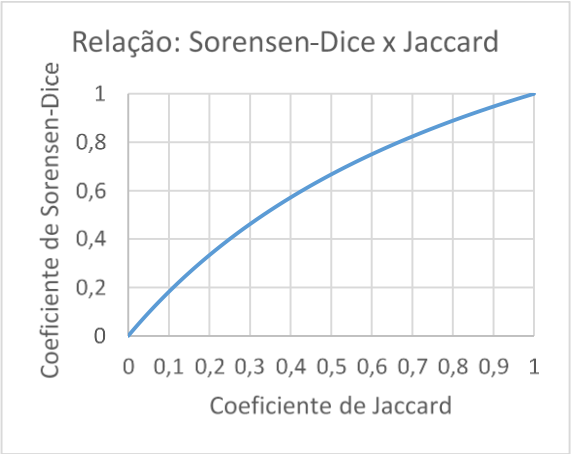
A escolha de uma medida adequada é primordial para evitar falsos resultados. Conforme Kleiman (2007), o falso positivo ocorre sempre que a ferramenta indica plágio quando isso não é verdade, enquanto que o falso negativo ocorre na situação oposta. Por mais que ambos sejam falsos resultados, deve-se ressaltar que o falso positivo é mais fácil de detectar. Dessa forma, é preferível que a medida de similaridade usada seja sensível demais, ao contrário de sensível de menos.

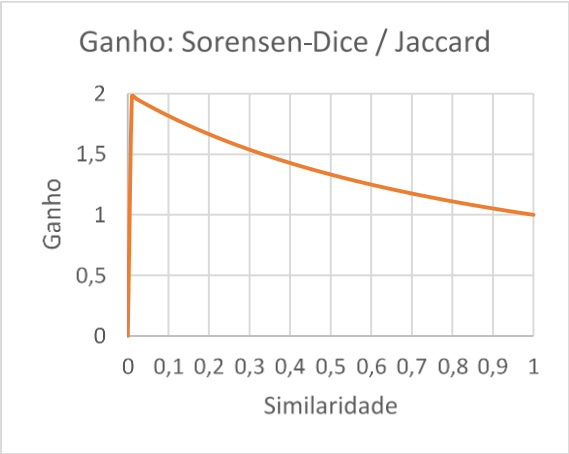
4.2.1. Coeficiente de Sorensen-Dice

A primeira medida proposta é o coeficiente de Sorensen-Dice, que é consideravelmente mais sensível que o coeficiente de Jaccard. Na verdade, o primeiro pode ser obtido em termos do segundo, ~~como segue~~:

Figura 8: Relações entre os coeficientes de Sorensen-Dice e Jaccard.

$$S(A,B) = \frac{2|A \cap B|}{|A| + |B|} = \frac{2|A \cap B|}{|A \cup B| + |A \cap B|} = \frac{\frac{2|A \cap B|}{1}}{\frac{|A \cup B| + |A \cap B|}{1}} \times \frac{\frac{1}{|A \cup B|}}{\frac{1}{|A \cup B|}} = \frac{2\frac{|A \cap B|}{|A \cup B|}}{1 + \frac{|A \cap B|}{|A \cup B|}} = \frac{2J(A,B)}{1 + J(A,B)}$$



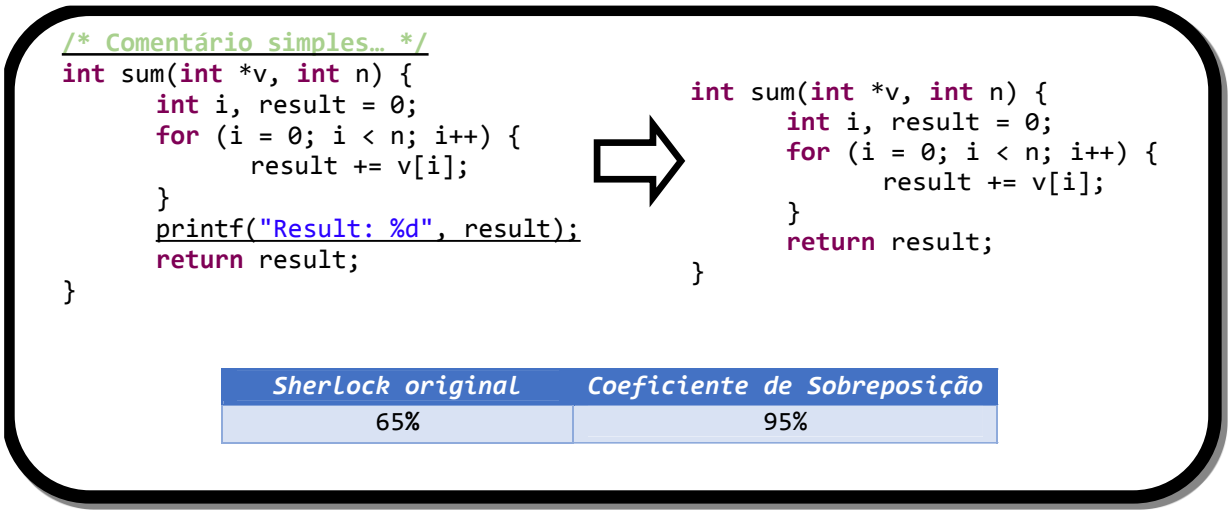


A utilização do coeficiente de Sorensen-Dice proporciona um ganho principalmente para valores de similaridade baixos. Esse ganho aumenta numericamente os valores de similaridade obtidos pelo Sherlock, mas não significa uma melhoria efetiva na detecção de similaridade. O coeficiente de Sorensen-Dice equaliza o espectro de similaridade, trazendo as similaridades mais baixas para um patamar mais próximo ao das similaridades altas.

4.2.2. *Coeficiente de Sobreposição*

O Coeficiente de Sobreposição exprime o quanto a interseção de dois conjuntos sobrepõem o menor dos conjuntos. Essa medida representa uma melhoria efetiva na detecção de similaridade para casos em que o código-fonte original é modificado com a adição ou remoção de código, independentemente da quantidade de código.

Figura 9: Robustez do Coeficiente de Sobreposição para código nulo.



Sherlock original	*Coeficiente de Sobreposição*
65%	95%

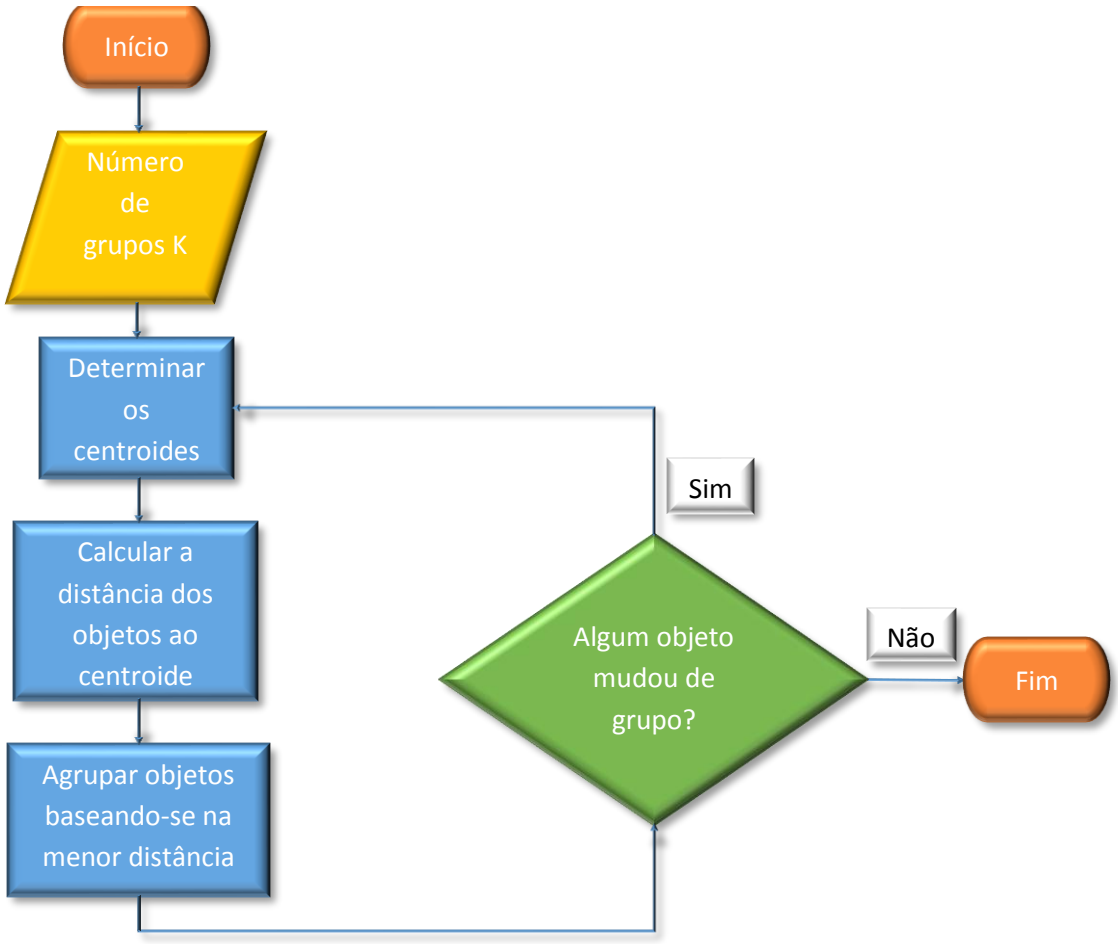
Na figura acima, os resultados da comparação dos dois códigos-fonte utilizando o Sherlock original e o Sherlock modificado com o coeficiente de sobreposição são expostos, evidenciando a melhoria alcançada com o segundo. A adição de código nulo afeta demasiadamente os resultados do Sherlock original. Isso acontece porque todas os as assinaturas são levadas em consideração no cálculo de similaridade do Sherlock original, diferentemente do Sherlock modificado com o coeficiente de sobreposição. Quando se parte do pressuposto que o tamanho do código-fonte plagiado é maior que o do código-fonte base, o coeficiente de sobreposição tem demonstrando-se mais apropriado.

4.3. Pós-processamento: Agrupamento dos códigos-fonte semelhantes

O objetivo dessa etapa é agrupar os códigos-fonte semelhantes, de modo a facilitar a identificações de soluções semelhantes. O problema do agrupamento de objetos é recorrente na mineração de dados. Dentre os algoritmos que se propõem a resolver esse problema, o k-médias[4] (MacQueen, 1967) é um dos mais simples.

O algoritmo k-médias particiona um conjunto de objetos em k subgrupos, baseando-se nos atributos ou características dos objetos. A ideia principal desse algoritmo é encontrar o centroide de cada grupo, sendo esse centroide definido como o elemento mais representativo do grupo. O fluxograma abaixo ilustra as etapas do algoritmo.

Figura 10: Fluxograma ilustrativo do algoritmo k-médias.

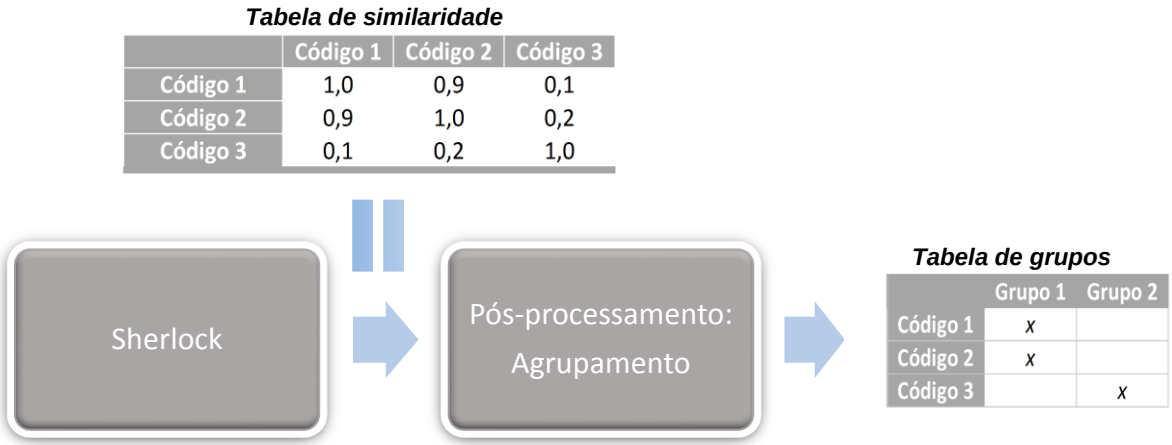


[4] *K-means no inglês.*

Uma das limitações do k-médias está associada à necessidade de se definir o valor de k a priori, o que é inviável quando analisamos um conjunto de objetos no qual não se sabe inicialmente o número de grupos real. Uma abordagem simples é executar o k-médias repetidamente com diferentes valores de k, e escolher o k que minimize algum critério pré-definido.

Uma customização do k-médias foi utilizada neste trabalho, de modo a tornar o algoritmo capaz de agrupar os códigos-fonte. A ideia é usar os resultados obtidos na execução do Sherlock como entrada do algoritmo, removendo a necessidade de se calcular as distâncias entre os objetos analisados a cada iteração do algoritmo. A relação entre similaridade e distância $dist(A, B) = 1 - sim(A, B)$, remove a necessidade de se calcular essas distâncias repetidamente, pois os valores de similaridade já são obtidos da execução do Sherlock.

Figura 11: Entrada e saída do pós-processamento.



5. RESULTADOS

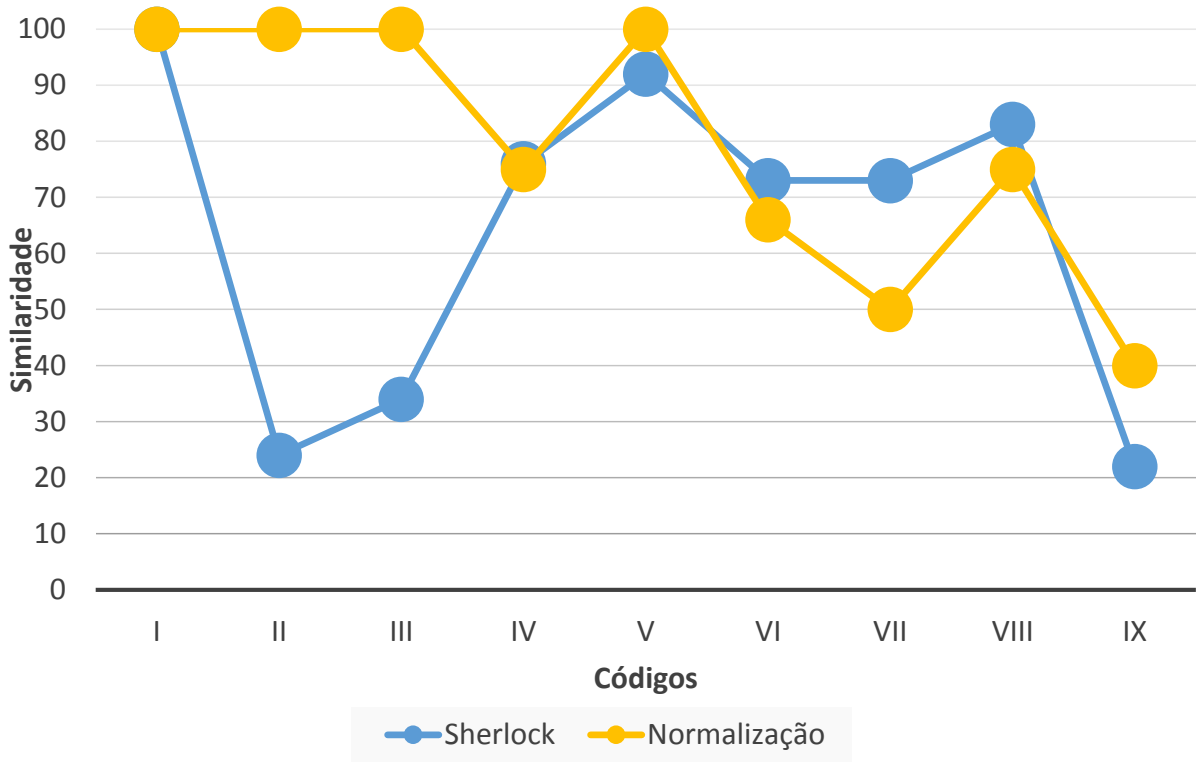
O resultados apresentados nas próximas seções foram obtidos da análise de similaridade em um conjunto de testes formado por códigos-fonte intencionalmente modificados seguindo alguns padrões de plágio mais frequentemente encontrados, adaptando-se aqueles que são elencados por Ahmadzadeh *et al.* (2011). As alterações inseridas no código foram realizadas da seguinte maneira: (I) Cópia do código base; (II) Inclusão/edição de comentários; (III) Mudança de nomes de variáveis; (IV) Troca de posição de variáveis e funções; (V) Mudança de escopo de variável; (VI) Alteração na identação; (VII) Inclusão de informação inútil: incluir bibliotecas, variáveis, comentários; (VIII) Rearranjo de expressões; (IX) Todas as alterações combinadas. As alterações realizadas não ocorrem de forma incremental, ao contrário, são originadas do mesmo código base, com exceção da última, que reúne alterações de todos os itens anteriores.

Em cada gráfico apresentado nas seções seguintes, os códigos-fonte do conjunto de testes são comparados um a um ~~ao~~ código-fonte base. Mais especificamente, o eixo horizontal do gráfico exibe de forma ordenada as modificações do código-fonte base, enquanto que os pontos associados indicam a similaridade obtida da comparação dessas modificações com o código-fonte base.

5.1. Resultados da utilização de pré-processamento

A melhoria alcançada com a normalização dos códigos-fonte antes da execução do Sherlock foi significante para as primeiras modificações. A padronização dos códigos-fonte, somada a eliminação dos elementos léxicos irrelevantes do ponto de vista da análise de similaridade, contribuíram majoritariamente para a obtenção desses resultados mais expressivos.

Figura 12: Sherlock vs Normalização.

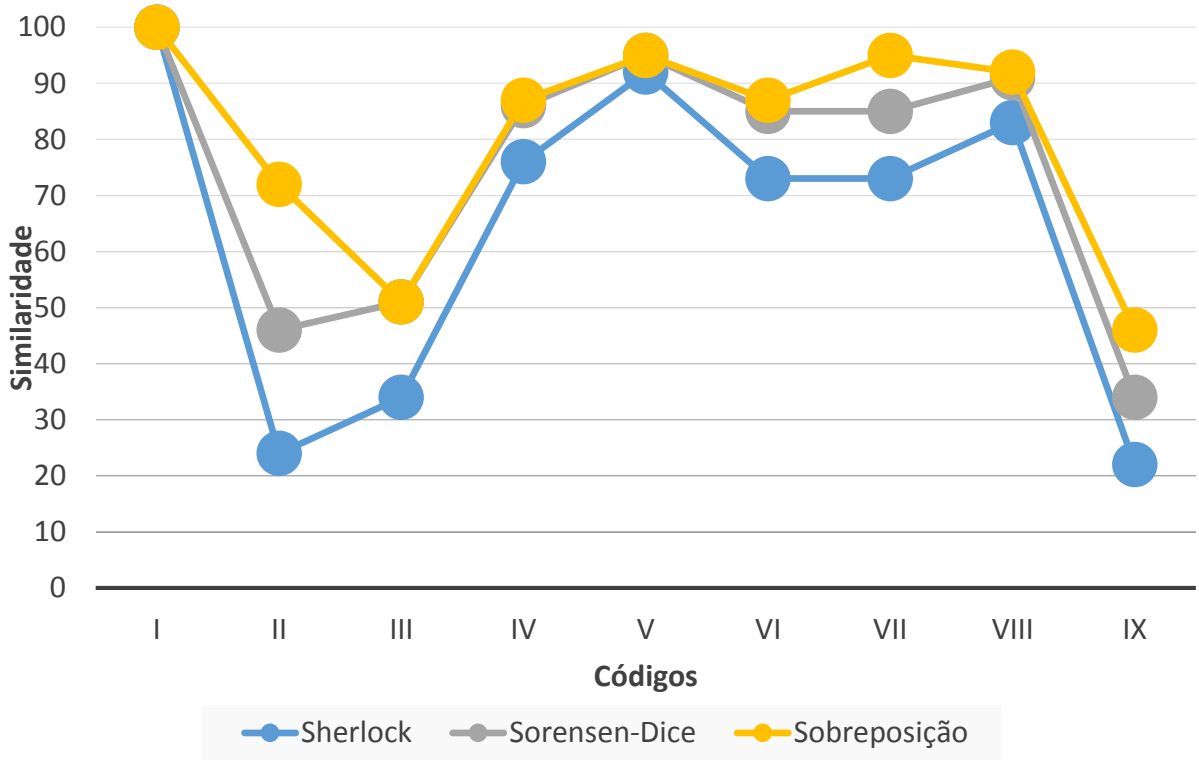


5.2. Resultados das alterações no Sherlock

As implementações do Coeficiente de Sorensen-Dice e o Coeficiente de Sobreposição obtiveram melhores resultados que o Sherlock original. Ambas as alterações apresentam melhoria com relação aos resultados do Sherlock, com destaque para o Coeficiente de Sobreposição que chegou a obter melhoria de 300% para o código II.

A alta similaridade apontada para código II (Inclusão/edição de comentários no código original) comprova que o coeficiente de sobreposição é especialmente imune a adição de código nulo, aspecto de grande importância para o escopo de práticas laboratoriais.

Figura 13: Sherlock original vs Sorensen-Dce vs Sobreposição.



5.3. Resultados preliminares do pós-processamento

O conjunto e testes usado nessa seção difere das demais. O conjunto de testes foi composto por 6 códigos base acrescidos de suas 8 modificações executadas conforme descrito no início desse capítulo (página 22), totalizando 48 códigos. O objetivo é verificar se o pós-processamento usando o k-médias é capaz de agrupar esses códigos de forma correta. O resultado esperado é que cada grupo seja formado pelo código base e suas modificações, em um total de 6 grupos. ~~A tabela abaixo expõem o resultado obtido~~.

Tabela 2: Resultados preliminares do pós-processamento.

	Código base 1	*Código base 2*	*Código base 3*	*Código base 4*	*Código base 5*	*Código base 6*
Modificação 1	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 2	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 3	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 4	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 5	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 6	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 7	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6
Modificação 8	Grupo 2	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6

Exceto pela modificação 8 do código base 1 (*c1m8*), os resultados foram conforme o esperado. Esse agrupamento aparentemente incorreto do *c1m8* foi devido ao baixo valor de similaridade encontrado pelo Sherlock para o par (*c1*, *c1m8)*, levando o k-médias a um equívoco no agrupamento. É de se esperar que falsos resultados oriundos da execução do Sherlock gerem falsos agrupamentos no pós-processamento, pois as etapas estão dispostas sequencialmente.

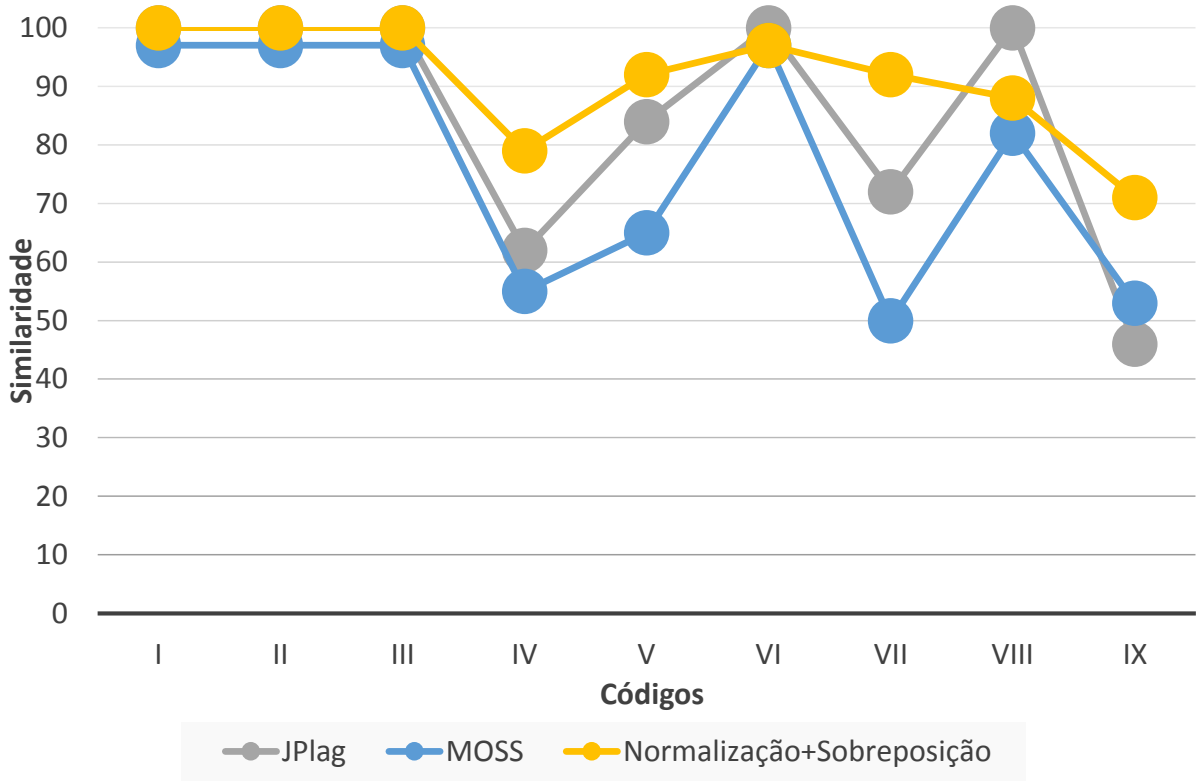
Os resultados preliminares obtidos com o pós-processamento evidenciam sua eficácia para o conjunto de testes, entretanto ainda é necessário avaliar a aplicação dessa etapa em códigos gerados por alunos.

5.4. Resultado geral das alterações propostas

O gráfico ~~abaixo~~ expõe~~m~~ os resultados obtidos na avaliação das alterações propostas em comparação com as duas ferramentas para detecção de plágio mais citadas: JPlag e MOSS. A implementação do coeficiente de sobreposição no Sherlock supera as outras duas ferramentas na maioria dos casos, exceto ~~por~~ VIII.

O código VIII contém um rearranjo das expressões matemáticas do código original. O JPlag trata a expressões matemáticas de maneira diferenciada, de modo a aumentar a robustez para o caso de plágio por rearranjo. Esse tratamento para expressões matemáticas não existe no Sherlock, de forma que simples alterações na ordem dos operadores e operandos, por exemplo, são capazes de iludir a ferramenta.

Figura 14: JPlag vs MOSS vs Normalização + Sobreposição.



6. CONCLUSÂO

A detecção de plágio em laboratórios de programação não é uma tarefa fácil, devido ao universo de alterações possíveis que podem ser aplicadas para plagiar um código-fonte, além do fato de que até mesmo a ocorrência de alta similaridade em códigos pode ser decorrente de práticas irrepreensíveis conforme (Maciel, 2012). Dependendo da perspectiva e do contexto, a semelhança pode ser indicativa de parceria, de trabalho colaborativo, de referência ou apoio sobre solução encontrada em livros ou exemplos fornecidos pelo professor, entre outros motivos.

Os resultados obtidos neste trabalho com a implementação do pré-processamento e das alterações do Sherlock revelam melhorias significativas para o módulo de suporte à detecção de plágio do BOCA-LAB, principalmente no sentido de obter maior eficiência na análise de similaridade. Entretanto, a automatização completa da detecção de plágio é desaconselhável, devido à incapacidade de identificação de práticas irrepreensíveis por parte das ferramentas.

As melhorias obtidas garantem ao professor um instrumental valioso para o acompanhamento pedagógico de laboratórios de programação, contribuindo para evitar práticas indesejáveis como o plágio, ao mesmo tempo que facilita a identificação manual de soluções similares.

REFERÊNCIAS BBLIOGRÁFICAS

PIKE, R. **The Sherlock Plagiarism Detector**. Disponível em: "http://sydney.edu.au/engineering/it/~scilect/sherlock/". Acesso em: 08 de abril de 2013.

CESARE, S.; XIANG, Y. **Software Similarity and Classification**, ISBN: 978-1-4471-2908-0, Springer, 2012.

Maciel, D. L.; SOARES, J. M; Bonetti, A.; Gomes, D.G. **Análise de Similaridade de Códigos-Fonte como Estratégia para o Acompanhamento de Atividades de Laboratório de Programação**. RENOTE. Revista Novas Tecnologias na Educação, v. 10, p. 1-10, 2012.

FRANÇA, A. B.; Soares, J. M. **Sistema de apoio a atividades de laboratório de programação via Moodle com suporte ao balanceamento de carga**. In: XXII Simpósio Brasileiro de Informática na Educação, Aracaju - SE. Anais do XXII SBIE, 2011. p. 710-719.

DOUGIAMAS, M.; TAYLOR, P. Moodle: **Using Learning Communities to Create an Open Source Course Management System**. In D. Lassner & C. McNaught (Eds.), Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2003 (pp. 171-178).

KLEIMAN, A. B. **Analise e comparação qualitativa de sistemas de detecção de plagio em tarefas de programação**. Dissertação (mestrado), Universidade Estadual de Campinas. Campinas, SP, p. 94. 2007.

CORNIC, P. **Detection using model-driven software development in eclipse platform**. University of Manchester (dissertação de mestrado). Manchester, p. 101. 2008.

VERCO, K.L.; WISE, M.J. **Software for detecting suspected plagiarism: comparing structure and attribute-counting systems**. ACM International Conference Proceeding Series, 1:81–88, 1996.

OTTENSTEIN, K. J. **Further Investigation into a Software Science Relationship**. ACM SIGCSE Bulletin, New York, NY, USA, v. 8, n. 4, p. 195~198, 1976.

HALSTEAD, M. H. **Elements of Software Science**. New York: Elsevier North-Holland, 1977.

FAIDHI, J. A. W.; ROBINSON, S. K. **An Empirical Approach for Detecting Program Similarity within a University Programming Environment**. Computers and Education 11(1), pp. 11-19, 1987.

DONALDSON, J. L.; LANCASTER, A.; SPOSATO, P. H. **A Plagiarism Detection System**. ACM SIGCSE Bulletin - Proceedings of the 12th SIGCSE symposium on Computer science education, New York, NY, USA , v. 13, n. 1, p. 21-25, 1981.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. **Finding Plagiarisms among a Set of Programs with JPlag**. Journal of Universal Computer Science, 8(11), 2002.

MOSS. Disponível em “http://theory.stanford.edu/~aiken/moss/”. Acessado em 04/09/2007.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. **Winnowing: Local Algorithms for Document Fingerprinting**. In SIGMOD 2003, 2003.

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

ANEXO A – PSEUDOCÓDIGO DO ALGORITMO GST

Pseudocódigo do algoritmo Greedy String Tiling

```
Greedy-String-Tiling(String A, String B) {
    tiles = {};
    do {
        maxmatch = M;
        matches = {};
        Forall unmarked tokens Aa in A {
            Forall unmarked tokens Bb in B {
                j= 0;
                while (Aa+j == Bb+j && unmarked(Aa+j) && unmarked(Bb+j))
                    j ++;
                if (j == maxmatch)
                    matches = matches ⊕ match(a, b, j);
                else if (j >maxmatch) {
                    matches = {match(a, b, j)};
                    maxmatch = j;
                }
            }
        }
        Forall match(a, b, maxmatch) ∈ matches {
            For j = 0. . . (maxmatch− 1) {
                mark(Aa+j);
                mark(Bb+j );
            }
            tiles = tiles ∪ match(a, b, maxmatch);
        }
    } while (maxmatch > M);
    return tiles;
}
```