****

**UNIVERSIDADE FEDERAL DO CEARÁ**

**CENTRO DE TECNOLOGIA**

**DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA**

**ENGENHARIA DE TELEINFORMÁTICA**

**FRANCISCO ANDERSON DE ALMADA GOMES**

**IMPLEMENTAÇÃO DE TÉCNICAS PARA A ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE BASEADAS EM CONTAGEM DE ATRIBUTOS ESTRUTURAIS E SEMÂNTICOS**

**FORTALEZA**

**2014**

FRANCISCO ANDERSON DE ALMADA GOMES

IMPLEMENTAÇÃO DE TÉCNICAS PARA A ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE BASEADAS EM CONTAGEM DE ATRIBUTOS ESTRUTURAIS E SEMÂNTICOS

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

2014

FRANCISCO ANDERSON DE ALMADA GOMES

IMPLEMENTAÇÃO DE TÉCNICAS PARA A ANÁLISE DE SIMILARIDADE ENTRE CÓDIGOS-FONTE BASEADAS EM CONTAGEM DE ATRIBUTOS ESTRUTURAIS E SEMÂNTICOS

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)

Ao grande mestre, Sobral.

**AGRADECIMENTOS**

*“A inovação distingue os líderes dos seguidores.”*

(Steve Jobs)

**RESUMO**

**Palavras-chave:**

**ABSTRACT**

**Keywords:**

**LISTA DE ILUSTRAÇÕES**

Figura 1 – legenda ....... 2

**LISTA DE TABELAS**

Tabela 1 – legenda ..... 3

**LISTA DE ABREVIATURAS E SIGLAS**

ABNT Associação Brasileira de Normas Técnicas

**SUMÁRIO**

[**1 INTRODUÇÃO** 1](#_Toc397786810)

[**1.1** **Contextualização** 1](#_Toc397786811)

[**1.2** **Motivação** 1](#_Toc397786812)

[**1.3** **Objetivos** 1](#_Toc397786813)

[**1.4** **Estrutura do documento** 2](#_Toc397786814)

[**2 FUNDAMENTAÇÃO TEÓRICA** 3](#_Toc397786815)

[**2.1 Técnicas de plágio** 3](#_Toc397786816)

[**2.2 Detecção de plágio em código-fonte** 5](#_Toc397786817)

[**2.2.1 Técnicas de detecção de plágio** 5](#_Toc397786818)

[**2.2.1.1 Técnicas baseadas em texto** 5](#_Toc397786819)

[**2.2.1.2 Técnicas baseadas em *tokens*** 6](#_Toc397786820)

[**2.2.1.3 Técnicas baseadas em árvores** 6](#_Toc397786821)

[**2.2.2.4 Técnicas baseadas em grafos** 7](#_Toc397786822)

[**2.2.1.5 Técnicas baseadas em métricas** 8](#_Toc397786823)

[**2.2.2 Algoritmos usados na detecção de plágio** 8](#_Toc397786824)

[**2.2.2.1 *Winnowing*** 8](#_Toc397786825)

[**2.2.2.2 *Greedy String Tiling*** 9](#_Toc397786826)

[**2.2.2.3 *Running Karp Rabin Greedy String Tiling*** 11](#_Toc397786827)

[**2.2.3.1 YAP3** 12](#_Toc397786828)

[**2.2.3.2 JPlag** 12](#_Toc397786829)

[**2.2.3.3 MOSS** 13](#_Toc397786830)

[**2.2.3.4 SIM** 15](#_Toc397786831)

**1 INTRODUÇÃO**

* 1. **Contextualização**

O acompanhamento de alunos em turmas numerosas de disciplinas voltadas ao aprendizado de técnicas de programação é uma dificuldade encontrada pelos professores e/ou monitores, que buscam usar a tecnologia para facilitar o gerenciamento de turmas grandes. Um problema encontrado em laboratórios de programação é a cópia total ou parcial de soluções entre colegas. Em cenários de turmas numerosas, a detecção deste tipo de conduta se torna bastante complicada.

A análise de similaridade em códigos-fonte é objetivo de estudo de diversos trabalhos e algumas ferramentas foram desenvolvidas para auxiliar na avaliação.

* 1. **Motivação**

Apesar da grande quantidade de ferramentas disponíveis, poucas delas são capazes de identificar de maneira eficaz todas as semelhanças léxicas e semânticas entre pares de códigos.

* 1. **Objetivos**

Tendo em vista as dificuldades apresentadas por essas ferramentas, esse trabalho propõe uma ferramenta que extrai as características-chave únicas da linguagem e o processamento dessas para criação de uma métrica que define o nível de similaridade entre dois códigos-fonte e a comparação dos resultados encontrados com as outras ferramentas de análise de similaridade em códigos-fonte.

Os objetivos específicos são:

* Efetuar um estudo teórico sobre as principais técnicas de detecção de plágio em código-fonte e estudar os resultados dos principais algoritmos relacionados.
* Analisar técnicas de normalização de códigos-fonte que melhorem o desempenho dos algoritmos de análise sintática no contexto de analise de similaridade.
* Introduzir métricas de complexidade a ferramenta.
  1. **Estrutura do documento**

O trabalho está organizado em cinco capítulos. O presente capítulo descreve uma breve introdução ao tema, contextualizando o assunto abordado, a motivação, os objetivos e as contribuições deste trabalho.

No Capítulo 2 são abordados os principais conceitos teóricos necessários para compreensão deste trabalho. Entre eles estão os principais conceitos relativos a técnicas de plágio, ao funcionamento de algoritmos e de ferramentas para análise de similaridade e detecção de plágio, as técnicas de normalização e as métricas de complexidade.

No Capítulo 3 apresenta a ferramenta desenvolvida, bem como o método de conformidade que foi desenvolvido para a análise dos resultados de múltiplas ferramentas.

No Capítulo 4, apresentam-se os resultados obtidos com códigos gerados manualmente.

Por fim, no Capítulo 5 são realizadas as considerações finais, bem como as possíveis melhorias a serem consideradas em trabalhos futuros.

**2 FUNDAMENTAÇÃO TEÓRICA**

Neste capítulo, são apresentados os principais conceitos teóricos necessários para compreensão deste trabalho. A Seção 2.1 apresenta os meios mais comuns empregados para realizar o plágio. A Seção 2.2 apresenta os fundamentos da detecção de plágio em código-fonte. A Seção 2.3 apresenta os algoritmos usados na detecção de plágio. A Seção 2.4 mostra as ferramentas mais comuns para análise de similaridade. Na Seção 2.5, as técnicas de normalização são abordadas. Por fim, as métricas de complexidade são discutidas na Seção 2.6.

**2.1 Técnicas de plágio**

Nos primeiros laboratórios de programação é fácil encontrar códigos com nomes de variáveis que são pouco usuais e idênticos, revelando certa ingenuidade na tentativa da cópia. Com o passar das aulas de programação, são apresentados problemas mais complexos, também se tornando mais difícil identificar o nível de similaridade entre pares de código dos alunos, mas não são esperadas modificações muito elaboradas em códigos plagiados.

As modificações utilizadas na cópia de um código-fonte podem ser bastante simples, como mudar comentários e nomes de variáveis, ou atingir níveis maiores complexidade, no caso de alterar uma estrutura de controle ou utilizar comandos similares (e.g., *for* no lugar de *while*), o que requer um conhecimento maior da linguagem de programação. Faidhi e Robinson estão entre os primeiros autores a caracterizar essas modificações, ilustrando-as conforme a Figura 2.1, que mostra, de dentro para fora, o aumento da dificuldade na detecção do plágio (FAIDHI e ROBINSON, 1987):

L0: Corresponde a códigos sem modificação;

L1: Corresponde a modificação de comentários no código;

L2: Corresponde a alteração de identificadores (e.g., os nomes das variáveis);

L3: Corresponde a mudança das posições de variáveis (e.g., tornar uma variável global em local);

L4: Equivale a modificações de combinação de procedimentos (e.g., trocar trechos de código por funções);

L5: Equivale a alterações nas instruções (e.g., substituir um operador por outro similar);

L6: Equivale a mudança na lógica de controle (e.g., *if* por *switch*).

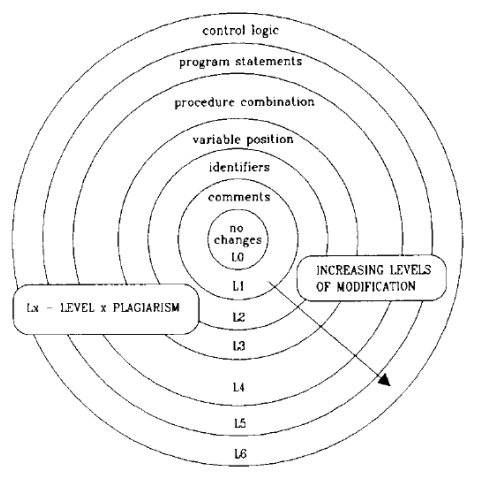


Figura 2.1 - Componentes da complexidade na detecção de plágio.

Fonte: FAIDHI e ROBINSON

De acordo com Whale (1990), as técnicas de modificações mais empregadas são:

i. Alteração de comentários e/ou formatação;

ii. Modificação de nomes de identificadores;

iii. Alteração da ordem de operandos e expressões;

iv. Alteração de tipos de dados;

v. Substituição de expressões por equivalentes;

vi. Adição de instruções redundantes ou variáveis;

vii. Alteração na ordem de instruções que não alteram o funcionamento;

viii. Alteração das estruturas de loop;

ix. Alteração das estruturas das instruções de seleção;

x. Substituição de chamadas a funções pelo respectivo conteúdo;

xi. Adicionar instruções que não influenciam o fluxo do programa;

xii. Combinação de código copiado com código original.

Para Mozgovoy (2006), as transformações para esconder o plágio são:

i. Alteração de comentários;

ii. Alteração de espaços em branco e layout;

iii. Modificação de nomes de identificadores;

iv. Reordenação de blocos de código;

v. Reordenação de instruções dentro de blocos de códigos;

vi. Alteração na ordem de operadores/operandos em expressões;

vii. Mudança de tipos de dados;

viii. Adição de instruções redundantes ou variáveis;

ix. Substituição de estruturas de controle por equivalentes;

x. Substituição da chamada a uma função pelo conteúdo da mesma.

Verifica-se que não existe uma grande variação entre as principais técnicas defendidas pelos autores. Para este trabalho, foram reunidas algumas dessas técnicas para a elaboração de um conjunto de códigos que representem as principais modificações. As ações utilizadas para a produção dos códigos são apresentadas na subseção 4.1.1.

**----------------------------------------------------------------------------------------------------------------**

**2.2.1 Contagem de atributos e comparação de estruturas**

Segundo Ji, Woo e Cho (2007), duas metodologias para detecção de plágio em código-fonte são utilizadas: contagem de atributos e comparação de estruturas. Contagem de atributos é uma metodologia que extrai e calcula informações (e.g., a freqüência de *keywords*). De acordo com Kleiman (2007) e Cornic (2008), essa abordagem é eficiente apenas para casos em que ocorreram poucas tentativas de alteração no código. Segundo Verco e Wise (2006), os primeiros sistemas automatizados para a detecção de plágio em código-fonte utilizavam a técnica de contagem de atributo em suas comparações. Segundo Ji, Woo e Cho (2007), a metodologia baseada 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.

**Análise sintática**

Como o Sherlock faz análise de elementos léxicos, temos dois problemas: incapacita a detecção de similaridade em códigos que possuam equivalência semântica (Figura 2.12) e 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.

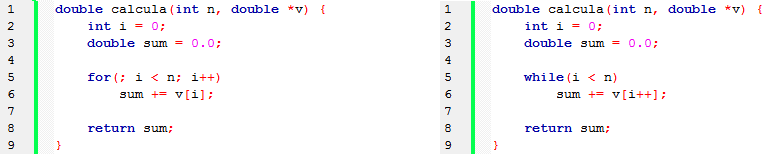


Figura 2.12 - Equivalência semântica em duas funções

**----------------------------------------------------------------------------------------------------------------**

**2.2 Detecção de plágio em código-fonte**

Esta seção apresenta as principais técnicas, algoritmos e ferramentas utilizadas na detecção de plágio de código-fonte.

**2.2.1 Técnicas de detecção de plágio**

De acordo com Roy e Cordy (2007), são classificados as técnicas de detecção de plágio da seguinte forma:

**2.2.1.1 Técnicas baseadas em texto**

Segundo Roy e Cordy (2007), essas técnicas são baseadas puramente em texto, também são conhecidas como abordagens léxicas, em que o código-fonte é considerado como uma sequência de linhas, sendo cada linha uma sequência de caracteres. Os algoritmos de comparação buscam por sequências de caracteres iguais em dois fragmentos de código. Algumas alterações são comumente aplicadas como: remoção de qualquer tipo de comentário, remoção dos espaços em branco e normalizações básicas. Essas técnicas são incapazes de detectar modificações mais complexas, que envolvem alterações na lógica do programa.

**2.2.1.2 Técnicas baseadas em *tokens***

Segundo Roy e Cordy (2007), essas técnicas são baseadas em *tokens*, em que o código-fonte analisado é transformado em uma sequência de *tokens*, posteriormente as comparações são feitas entre as sequências obtidas a fim de encontrar a maior subsequência comum. Essa abordagem é normalmente mais robusta do que a baseada em texto, sendo a base para as ferramentas mais complexas.

**2.2.1.3 Técnicas baseadas em árvores**

Segundo Roy e Cordy (2007), essas técnicas são baseadas em uma representação hierárquica de um programa, *Abstract Syntax Trees* (AST), em que consiste em uma árvore de *tokens* e segue um conjunto de regras sintáticas próprias de cada linguagem. A detecção de plágio utilizando AST consiste em encontrar sub-árvores comuns de dois programas. A utilização de AST tornasse mais robusta contra renomeação quando são eliminados nomes de variáveis e constantes. Contudo, segundo Cornic (2008), as técnicas baseadas em AST são vulneráveis a inserção de código ou códigos reestruturados. Na Figura 2.2, tem um simples código-fonte e na Figura 2.3 é mostrada AST desse código.

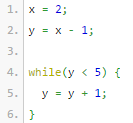


Figura 2.2 - Simples código-fonte em C

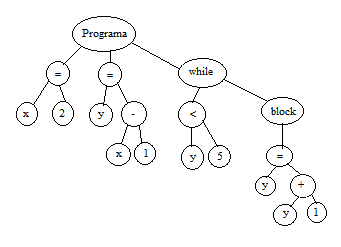


Figura 2.3 - AST do código-fonte

**2.2.2.4 Técnicas baseadas em grafos**

Segundo Roy e Cordy (2007), essas técnicas são baseadas em uma representação em grafo de uma função ou procedimento do código-fonte, *Program Dependence Graph* (PDG). PDGs podem mostrar a estrutura profunda dos programas, pois contém informações sobre o fluxo de controle e o fluxo de dados do software analisado. Diferente das outras representações utilizadas na detecção de plágio, essa não armazena informação sintática.

Modificações sem qualquer efeito sobre a execução do programa (e.g., alterar os nomes das variáveis) não alteram ou modificam um PDG. Assim, para modificá-lo, necessitaria modificar o funcionamento do programa, o que exigirá uma boa compreensão do mesmo. Na Figura 2.4 é apresentado um código-fonte para exemplificar o uso do PDG e na Figura 2.5 mostra o PDG desse código:

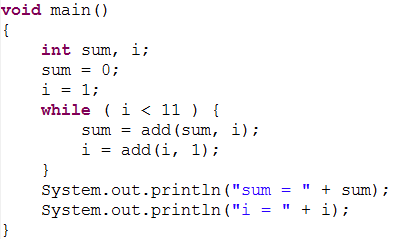


Figura 2.4 - Código-fonte

Fonte: http://www4.comp.polyu.edu.hk/~cscllo/teaching/SDGAPI/background.htm

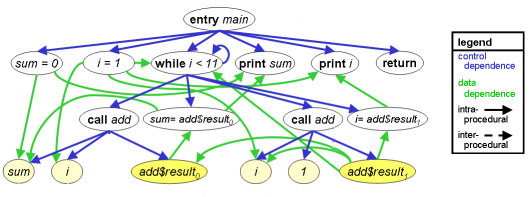


Figura 2.5 - PDG do código

Fonte: http://www4.comp.polyu.edu.hk/~cscllo/teaching/SDGAPI/background.htm

**2.2.1.5 Técnicas baseadas em métricas**

Segundo Roy e Cordy (2007), essas técnicas são baseadas em reunir diferentes métricas de fragmentos de código e compara esses vetores de métricas em vez de comparar o código diretamente. Existem muitas técnicas de detecção de plágio que utilizam várias métricas de software para detecção de similaridade de código. O método é simples de implementar e é um ótimo complemento nos algoritmos utilizados hoje em dia. Na seção 2.3 são apresentados algumas métricas de software.

**2.2.2 Algoritmos usados na detecção de plágio**

Kleiman (2007) apresentou alguns algoritmos para a detecção de plágio em código-fonte, os principais são descritos nas próximas linhas.

**2.2.2.1 *Winnowing***

O *Winnowing* é um algoritmo que tem como objetivo melhorar a eficiência do processo de comparação de documentos com base em assinatura única por documento. Segundo Schleimer et. al (2003), esse 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. Segundo Kleiman (2007), os k-gramas de uma cadeia S são as sub-cadeias de comprimento k contiguas e sobrepostas da cadeia S. Os 3-gramas da frase “o gato morreu" são: ‘oga’, ‘gat’, ‘ato’, ‘tom’, ‘omo’, ‘mor’, ‘orr’, ‘rre’, ‘reu’, 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 *winnowing* do super-conjunto de todos os valores para ser a assinatura do documento. Na Figura 2.6 é apresentado um código que executa o w*innowing*.

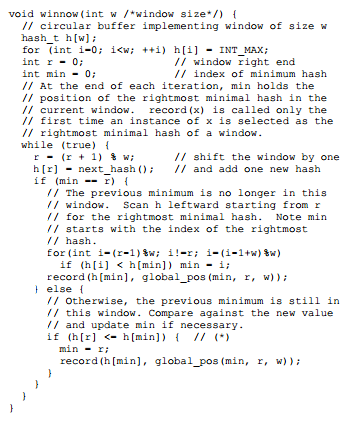


Figura 2.6: Código para w*innowing*

Fonte: Schleimer (2003).

**2.2.2.2 *Greedy String Tiling***

O algoritmo *Greedy String Tiling* (GST) foi introduzido por Wise (1993), ele compara duas strings e determina o seu grau de similaridade. Para compreensão, é importante entender algumas definições. A cadeia de caracteres padrão é uma referência à menor das cadeias de caracteres comparados, enquanto que a maior é referenciada por cadeia de caracteres texto. Sendo P uma cadeia de caracteres padrão e T uma cadeia de caracteres texto, a máxima-correspondência acontece quando uma sub-cadeia *Pp* de uma cadeia padrão iniciada em p, corresponde elemento a elemento com uma sub-cadeia *Tt* da cadeia texto iniciada em t.

O algoritmo executa múltiplas passagens nos dados, cada uma delas é composto de duas fases. Na primeira fase, as máximas-correspondências acima de um certo comprimento são coletadas e armazenadas em listas, de acordo com seus comprimentos. A segunda fase constrói *tiles* (associação permanente e única de uma sub-cadeia a partir de P com uma sub-cadeia correspondente em T) com máxima-correspondência da primeira fase, começando com a mais longa. Para cada correspondência, o algoritmo testa se ele está marcado. Se não, um *tile* é criado com esta correspondência e os textos correspondentes em P e T são marcados. Quando a correspondência de comprimento considerado forem tratados, um comprimento menor é escolhido e começa novamente a busca da primeira fase. O algoritmo pára quando o texto completo estiver marcado.

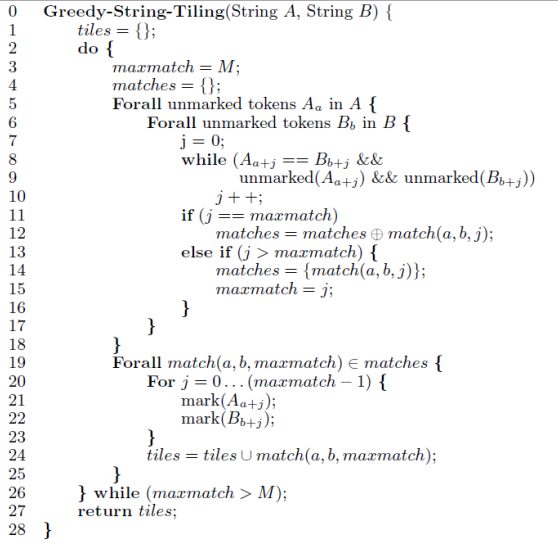


Figura 2.7 – Algoritmo *Greedy String Tiling*

Fonte: Prechelt, Malpohl e Philippsen (2002).

**2.2.2.3 *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 ocorrências em cadeia de caracteres. A idéia principal deste algoritmo gira em torno de uma função hash que gera uma assinatura para as sub-cadeias de P (cadeia padrão) e T (cadeia texto), essas assinaturas são utilizadas para encontrar ocorrências de uma cadeia de caracteres em outras. Wise (1993) aplicou a ideia do algoritmo *Karp-Rabin* ao algoritmo GST e criou o algoritmo *Running Karp-Rabin Greedy String Tiling* (RKR-GST). Este algoritmo é um dos mais usados nos sistemas detectores de plágio e, segundo Wise (1993), sua complexidade no pior caso é *𝑂(𝑛3)*.

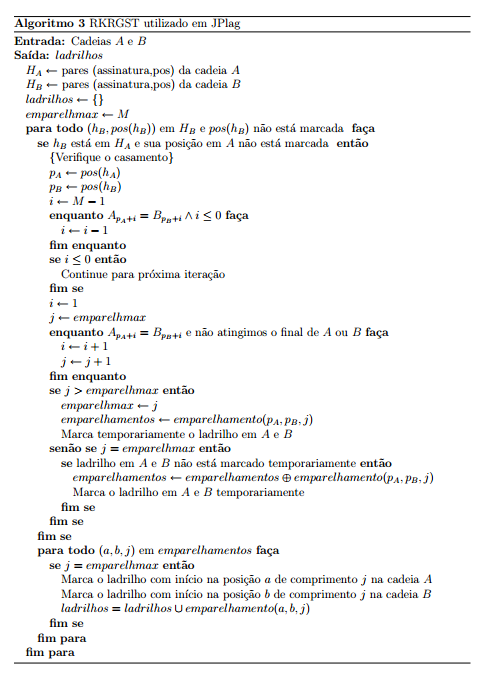


Figura 2.8 - Algoritmo *Running Karp-Rabin Greedy String Tiling*

Fonte:Kleiman (2007)

**2.2.3 Ferramentas para detecção de plágio**

As ferramentas especificamente construídas para realizar a análise de similaridade (ou detecção de plágio) entre códigos-fonte, permitem encontrar semelhanças no caso de alterações de nomes de variáveis, nomes de funções, comentários ou, ainda, alterando-se a ordem de partes do código. Diversas ferramentas podem ser utilizadas, tais como: YAP, JPlag, MOSS, SIM e Sherlock.

**2.2.3.1 YAP3**

O *Yet Another Plague* (YAP) (WISE, 1992) concentra-se principalmente na detecção de plágio em código-fonte. O Yap3 (WISE, 1996) é a terceira versão do YAP que utiliza RKR-GST como algoritmo. O sistema YAP foi desenvolvido visando corrigir falhas e melhorar a performance dos métodos de análise léxica e contagem de atributos de outras ferramentas, tendo como principais diferenças a excelente capacidade de tratar sequências transpostas, o maior número de linguagens de programação aceitas e o processo de normalização que precede o estudo das estruturas sintáticas e léxicas das entradas (WISE, 1996).

Segundo Clough (2000), o YAP é versátil pois é imune a grande maioria das técnicas utilizadas por estudantes (e.g., mudanças de comentários, alteração de identificadores e tipos de variáveis). Como desvantagem da ferramenta destaca-se a não sensibilidade a troca de ordem dos elementos não semânticos que compõem o código.

**2.2.3.2 JPlag**

O JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2000) é uma ferramenta de código fechado desenvolvida em Java para detecção de plágio e atualmente está disponibilizada no endereço: https://jplag.ipd.kit.edu/. É bem utilizada para desestimular a cópia não permitida de programas de exercícios de estudantes no ensino de programação. De acordo com Kleiman (2007), o sistema funciona como uma aplicação Java WebStart que opera via uma interface gráfica. O usuário envia os arquivos que serão comparados dentro de um diretório e os resultados são processados imediatamente, em seguida esses resultados são apresentados em arquivos HTML, que pode ser baixado ou visualizado online. A Figura 2.9 mostra parte de uma página de exibição de resultados do JPlag para um par de códigos.

De acordo com Prechelt, Malpohl e Philippsen (2000), o JPlag utiliza RKR-GST com otimizações como algoritmo que funciona assim: primeiro os códigos-fontes enviados para comparação são submetidos a um analisador que interpreta as estruturas da linguagem e gera *tokens*, em seguida compara esses *tokens* par a par, sendo, na sequência, calculado um índice de similaridade por par.

A ferramenta é robusta contra muitos tipos de plágio e atualmente suporta várias linguagens de programação, tais como: Java, C#, C, C++ e Scheme. No JPlag é possível comparar arquivos de texto em linguagem natural.

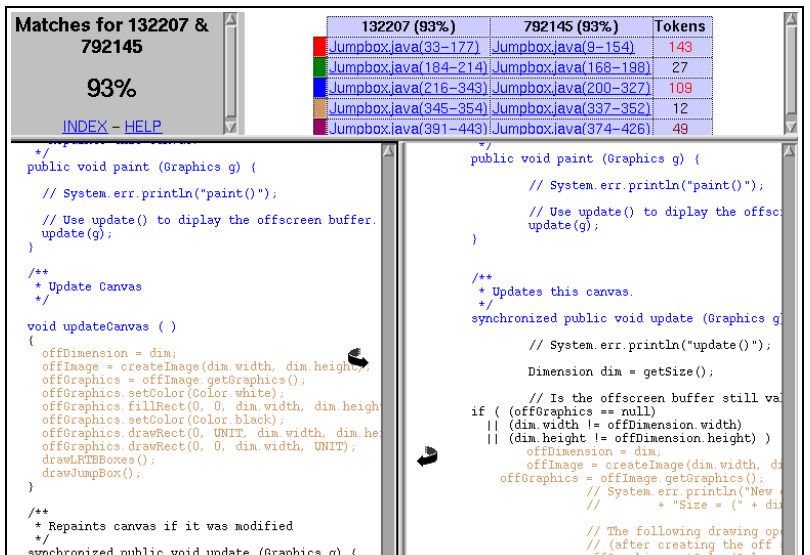


Figura 2.9 - Comparação entre dois códigos no JPlag

Fonte: Adaptado de Prechelt, Malpohl e Philippsen (2000).

**2.2.3.3 MOSS**

*Measure of Software Similarity* (MOSS) é um sistema para detectar similaridade entre programas e atualmente está disponível no endereço: http://theory.stanford.edu/~aiken/moss/. Seu desenvolvimento iniciou-se em 1994 pelo professor Alex Aiken da Universidade da Califórnia em Berkeley.

Schleimer, Wilkerson e Aiken (2003) apresentaram alguns detalhes sobre o sistema que utiliza o algoritmo *Winnowing* para efetuar a comparação de documentos. Existem poucos detalhes sobre o funcionamento do MOSS, mas de acordo com Kleiman (2007), detalhes da implementação não foi liberados ao público, sob o risco de que, ao conhecer o funcionamento do sistema, poderiam burlar o sistema.

Da mesma forma que o JPlag, o MOSS também é acessado exclusivamente por meio de um webservice, em que os código são submetidos utilizando um script disponibilizado pelo autor e os resultados são processados imediatamente, em seguida esses resultados são apresentados em uma página HTML, que permanece disponível para consulta por 14 dias. A Figura 2.10 mostra o resultado de uma submissão, em que é listado em ordem decrescente de similaridade os pares de código e a Figura 2.11 mostra a comparação entre dois códigos.

Atualmente o sistema suporta as seguintes linguagens de programação: C, C#, C++, Java, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, HCL2. O MOSS também efetua comparações sobre textos comuns.

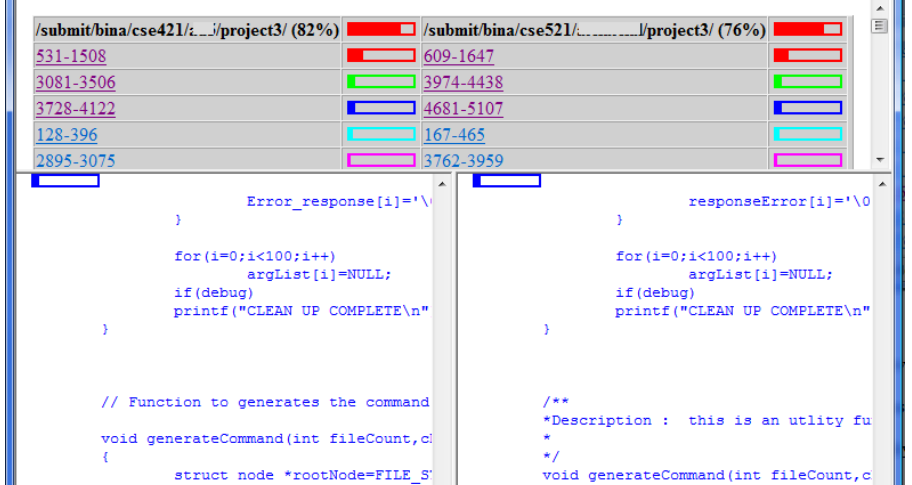


Figura 2.11 - Comparação entre dois códigos no MOSS.

Fonte: Adaptado de Ramarurthy e Settembre (2008)

**2.2.3.4 SIM**

Software Similarity Tester (SIM) é uma ferramenta *open-source* para análise léxica desenvolvida em 1989 na Universidade de Vrije em Amsterdam por Dick Grune. Atualmente está na versão 2.77 (2012) e está disponível para download dos códigos-fontes e dos binários no endereço: http://dickgrune.com/Programs/similarity\_tester/. A ferramenta é capaz de detectar plágio em projetos de software e detectar fragmentos duplicadas em grandes projetos de software.

O algoritmo básico da ferramenta é (CLOUGH, 2000):

i. Cada arquivo é lido por um *tokenizer* apropriado as entradas e *tokens* de um byte que são armazenados em um vetor.

ii. Cada sub-cadeia de caracteres do vetor é comparada com todas as outras e é associada em uma tabela hash ao índice que aponta para o início da próxima cadeia de caracteres igual a ela. Se não houver correspondência, o valor do índice é 0.

iii. O algoritmo percorre todos os arquivos e encontra a melhor correspondência entre as sub-cadeias de caracteres (chamado de *run*).

iv. Encontra o número da linha de início e fim de cada bloco pela posição do *token*.

v. Os *runs* armazenados são ordenados em ordem de importância e então são apresentados os resultados.

Atualmente as linguagens suportadas pela ferramenta são: C, Java, Pascal, Modula-2, Lisp, Miranda. A ferramenta também suportar linguagem natural.

**2.2.3.5 Sherlock**

O Sherlock (PIKE; LOKI, 2013) é uma ferramenta de código aberto, assim permite modificações e melhorias, desenvolvida em C para detectar plágio. A ferramenta apresenta bom desempenho e realiza a análise de semelhança léxica entre documentos de texto, assim é capaz de detectar plágio em códigos-fonte de qualquer linguagem de programação. Para encontrar a similaridade entre fragmentos dos documentos, é gerada uma assinatura digital calculando valores hash para sequência de palavras. Ao final, é feito uma comparação entre assinaturas geradas para indica o grau de similaridade.

A ferramenta é composta de dois programas: Sig e Comp. O primeiro gera as assinaturas digitais que representam cada documento analisado e armazena em um arquivo. O programa Comp usa esse arquivo de assinaturas para calcular e mostrar o percentual de similaridade. Assim, a detecção de plágio do Sherlock se resume em duas etapas: geração de assinaturas e comparação de assinaturas. As próximas subseções descrevem melhor essas etapas.

**2.2.3.5.1 Geração de assinaturas**

No inicio dessa etapa são extraído os *tokens,*em que o inicio e o fim de cada token é delimitado por espaços, tabulações ou novas linhas. Na Figura 2.13 temos os tokens destacados de um trecho de código. Esses *tokens* são mapeados em assinaturas por meio da função *hash*:

(2.1)

O inteiro K, que representa um token, é mapeado em um inteiro dentro do intervalo [0, 𝑀 − 1], onde 𝑀 = 232. O inteiro K é obtido do somatório:

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. A próxima etapa utilizará esses vetores de assinaturas para comparações.

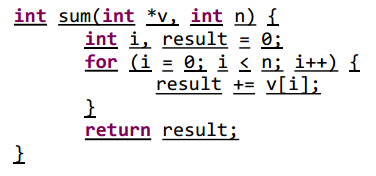


Figura 2.13 – Trecho de código com os *tokens* destacados

Fonte: Camargos (2013)

**2.2.3.5.2 Comparação de assinaturas**

A comparação das assinaturas dos códigos-fonte são feitas nessa etapa, exigindo um número de comparações igual à combinação sem repetição de n elementos 2 a 2. A similaridade é definida como a percentagem de semelhança entre as assinaturas correspondentes aos dois arquivos A e B que são comparados, como formulado na equação Equação 2.3.

em que a é o número de assinaturas similares encontradas em ambos os arquivos (A e B), b é o número de assinaturas encontradas exclusivamente em A e c o número de assinaturas encontradas exclusivamente em B.