# 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 tipos de similaridade, assim como as técnicas de normalização. A Seção 2.3 apresenta as principais técnicas para detectar plágio. Na Seção 2.4 e Seção 2.5 é apresentado os algoritmo e as ferramenta mais comuns para análise de similaridade, respectivamente. 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*).

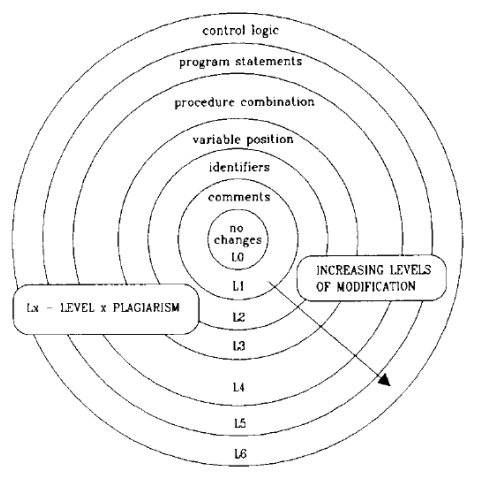


Figura 2.1 - Componentes da complexidade na detecção de plágio.

Fonte: FAIDHI e ROBINSON

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

1. Alteração de comentários e/ou formatação;
2. Modificação de nomes de identificadores;
3. Alteração da ordem de operandos e expressões;
4. Alteração de tipos de dados;
5. Substituição de expressões por equivalentes;
6. Adição de instruções redundantes ou variáveis;
7. Alteração na ordem de instruções que não alteram o funcionamento;
8. Alteração das estruturas de loop;
9. Alteração das estruturas das instruções de seleção;
10. Substituição de chamadas a funções pelo respectivo conteúdo;
11. Adicionar instruções que não influenciam o fluxo do programa;
12. Combinação de código copiado com código original.

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

1. Alteração de comentários;
2. Alteração de espaços em branco e layout;
3. Modificação de nomes de identificadores;
4. Reordenação de blocos de código;
5. Reordenação de instruções dentro de blocos de códigos;
6. Alteração na ordem de operadores/operandos em expressões;
7. Mudança de tipos de dados;
8. Adição de instruções redundantes ou variáveis;
9. Substituição de estruturas de controle por equivalentes;
10. 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 Tipos de similaridade

De acordo com Walenstein (2007), antes de classificar a similaridade, o desafio é responder o que será avaliado como similar. Walenstein (2007) separou a similaridade em dois grupos: sintática e semântica.

# 2.2.1 Similaridade sintática

Para Walenstein (2007), a similaridade sintática refere-se ao programa como sendo uma sequência de caracteres formando uma estrutura de texto completa. A similaridade pode ser definida em termos de suas formas, propriedades ou características.

Apesar da grande utilidade da análise de similaridade sintática (baseada em elementos textuais do código-fonte e da linguagem), as técnicas baseadas nessa análise, apresentadas na Seção 2.3, não são suficientes para identificar plágio.

Na Figura 2.2 de (MOTA; GOYA, 2010), é possível verificar as falhas. Os trechos de códigos são praticamente idênticos no nível textual, mas no momento da execução eles apresentam comportamentos completamente distintos. A variável sum é global no primeiro bloco e local no segundo bloco, o que remete a outra diferença significativa no momento da execução. Para resolver essas falhas é importante introduzir os conceitos semânticos, como será feito na Seção 2.2.2.

Outra problemática na analise puramente sintática é a utilização de elementos léxicos que não contém informação importante para a comparação, assim a detecção pode ser facilmente contornáveis se não houver a eliminação de trechos inúteis e eventuais redundâncias que possam ter sido inseridas com essa intenção. Esse processo chama-se normalização e será discutido a seguir.

|  |
| --- |
| **int** sum = 0 ;  **void** foo (Iterator iter){  **for** ( item = first (iter) ; has more (iter) ; item = next (iter) ) {  sum = sum + value (item) ;  }  }  **int** bar (Iterator iter) {  **for** ( item = first (iter) ; has more (iter) ; item = next (iter) ) {  sum = 0 ;  sum = sum + value (item) ;  }  } |

Figura 2.2 - Semelhança textual com comportamento distinto

# 2.2.1.1 Técnicas para normalização de códigos

De acordo com Danilo et al (2012), a normalização é um pré-processamento que atua principalmente na identificação e remoção de trechos de código sem relevância, além de padronizar os códigos para facilitar a comparação. Essas modificações são limitadas a mudanças que não alteram o funcionamento do programa. Para Mahmoud (2009) essa técnica vem colaborando como um suporte imprescindível na busca por melhorias em ferramentas de comparação textual.

Na Tabela 1, adaptada de Danilo et al (2012), são apresentadas quatro abordagens de normalização, com graus de intervenção crescentes. O código original, apresentado a Figura 2.3, é comparado com os códigos gerados com o uso das técnicas, sendo que o código gerado pela primeira técnica é o que mais se assemelha ao da versão original e o código gerado com a quarta técnica o que realiza alterações mais expressivas.

|  |
| --- |
| // Imprime resposta  for (i=1;i<= n;i++){  if (i %3== 1){  printf("Resp %d", v[i]/x+z);  }  } |

Figura 2.3 - Código original para ser aplicado a normalização

|  |  |  |
| --- | --- | --- |
| Técnica | Modificações | Código |
| 1 | Remoção de linhas e espaços vazios;  Remoção de todos os comentários;  Remoção das referências aos arquivos externos (bibliotecas);  Inclusão (ou remoção) de espaços em branco entre expressões, declaração de variáveis e outras estruturas.  Regras específicas para aproximar e afastar caracteres para normalização 1. | for( i=1; i<=n; i++ ) {  if( i % 3 ==1 ) {  printf ( " Resp %d ", v[ i ] / x +z );  }  } |
| 2 | Aplicação da normalização 1;  Remoção de todos os caracteres situados entre aspas.  Regras específicas para aproximar e afastar caracteres para normalização 2 | for( i=1; i<=n; i++ ) {  if( i % 3 ==1 ) {  printf( , v[ i ] / x +z );  }  } |
| 3 | Aplicação da normalização 2;  Remoção de todos os valores literais e variáveis.  Regras específicas para aproximar e afastar caracteres para normalização 3. | for( = ; <= ; ++ ) {  if( % == ) {  printf( , [] / +);  }  } |
| 4 | Aplicação da normalização 3;  Remoção de todas as palavras reservadas.  Regras específicas para aproximar e afastar caracteres para normalização 4 | ( =; <=; ++ ) {  ( %== ) {  ( , [] / +);  }  } |

Tabela 2.1 - Descrição das técnicas de normalização

Como pode ser visto na Tabela 1, cada normalização tem modificações específicas e herda da anterior algumas propriedades, que podem ser diferenciadas nas regras que levam a aproximar ou afastar determinados caracteres (e.g., na normalização 3 os resultados são melhores quando “% ==” ficam separados, enquanto que na normalização 4 os resultados são melhores com “%==” juntos). Apesar de colaborar bastante no resultado, como mostrado em Danilo et al (2012), essa metodologia ainda não é capaz de lidar com mudanças de ordem de operadores e outras modificações que são detectáveis através de análise semântica, assunto discutido a seguir.

# 2.2.2 Similaridade semântica

A semelhança semântica está relacionada ao comportamento do programa, que é definido pelas funções que estão implementadas (MOTA; GOYA, 2010), e não aos elementos textuais que compõem o código-fonte, como na Seção 2.2.1. A dificuldade está em definir medidas de similaridade semântica. Mota et al (2010), define dois tipos de representação que pode ser utilizado neste tipo de comparação:

* **Semelhança funcional:** Dois programas podem ser considerados análogos se implementarem uma função similar.
* **Semelhança de execução:** É observada a sequência de execução das instruções do programa (e.g., *bytecode* Java*)*. Nesse caso é necessário encontrar uma correspondência entre as instruções dos programas executáveis.

Existe um consenso quanto à dificuldade de obter medidas de similaridade semântica, pois é possível implementar de diversas formas o mesmo código. Na Figura 2.4, adaptada de (MOTA; GOYA, 2010), os códigos são semanticamente idênticos. Um programador experiente pode tornar muito difícil a tarefa de detecção de plágio e simples modificações podem ser consideradas complicadas para algumas ferramentas. Apesar das dificuldades, existem técnicas eficientes de combate ao plágio baseadas nessa abordagem, como será visto na próxima seção.

|  |
| --- |
| while ((x = pi[t-1]) != ‘(‘ && x != ‘+’ && x != ‘-‘) {  posf[j++] = x;  --t;  }  while(1) {  x = pi[t-1];  if (x == ‘(‘ || x == ‘+’ || x == ‘-‘)  break;  --t;  posf[j++] = x;  } |

Figura 2.4 - Códigos semanticamente idênticos, com diferenças sintáticas.

# 2.3 Técnicas de detecção de plágio

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

# 2.3.1 Técnicas baseadas em texto

Essas técnicas são baseadas puramente em texto, também conhecidas como abordagens léxicas, sendo então uma análise sintática, em que o código-fonte é considerado como uma sequência de linhas, sendo cada linha uma sequência de caracteres (ROY; CORDY, 2007). 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.

# 2.3.2 Técnicas baseadas em métricas

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 (ROY; CORDY, 2007) e é uma medida de similaridade sintática. Existem muitas técnicas de detecção de plágio que utilizam várias métricas de software para detecção de similaridade de código. O método é simples de programar e é um ótimo complemento nos algoritmos utilizados hoje em dia. Essa técnica é detalhada na Seção 2.6, pois é a utilizada na ferramenta proposta.

# 2.3.3 Técnicas baseadas em *tokens*

Essas técnicas são baseadas em *tokens* (palavras, frases, símbolos ou qualquer outro elemento significativo que possa vir a ter utilidade no processo de análise de similaridade), 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 (ROY; CORDY, 2007) e é considerada uma medida de similaridade sintática. Essa abordagem é normalmente mais robusta do que a baseada em texto, sendo a base para as ferramentas mais complexas.

# 2.3.4 Técnicas baseadas em árvores

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 (ROY; CORDY, 2007), então é uma medida de similaridade sintática. A detecção de plágio utilizando AST consiste em encontrar sub-árvores comuns de dois programas. Para exemplificar, na Figura 2.5 é mostrado um simples código e na Figura 2.6 a respectiva AST desse código.

|  |
| --- |
| x = 2;  y = x – 1;  while(y < 5) {  y = y + 1;  } |

Figura 2.5 - Simples código-fonte em C

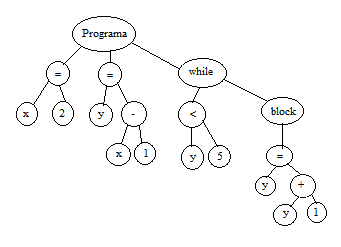


Figura 2.6 - AST do código-fonte

# 2.3.5 Técnicas baseadas em grafos

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

# 2.4 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.

# 2.4.1 *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 osvalores para ser a assinatura do documento. Na Figura 2.7 é apresentado um código que executa o W*innowing*.

|  |
| --- |
| void winnow(int w /\*window size\*/) {  // circular buffer implementing window of size w  hash\_t h[w];  for (int i=0; i<w; ++i) h[i] = INT\_MAX;  intr=0; //window right end  int min = 0; // index of minimum hash  // At the end of each iteration, min holds the  // position of the rightmost minimal hash in the  // current window. record(x) is called only the  // first time an instance of x is selected as the  // rightmost minimal hash of a window.  while (true) {  r = (r + 1) % w; // shift the window by one  h[r] = next\_hash(); // and add one new hash  if (min == r) {  // The previous minimum is no longer in this  // window. Scan h leftward starting from r  // for the rightmost minimal hash. Note min  // starts with the index of the rightmost  // hash.  for(int i=(r-1)%w; i!=r; i=(i-1+w)%w)  if (h[i] < h[min]) min = i;  record(h[min], global\_pos(min, r, w));  } else {  // Otherwise, the previous minimum is still in  // this window. Compare against the new value  // and update min if necessary.  if (h[r] <= h[min]) { // (\*)  min=r;  record(h[min], global\_pos(min, r, w));  }  }  }  } |

Figura 2.7 - Código para W*innowing*

Fonte: Schleimer (2003).

# 2.4.2 *Greedy String Tiling*

O algoritmo *Greedy String Tiling* (GST) foi introduzido por Wise (1993), ele compara duas cadeias de caracteres 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 composta de duas fases. Na primeira fase, as máximas-correspondências acima de 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 for tratada, um comprimento menor é escolhido e começa novamente a busca da primeira fase. O algoritmo para quando todo o texto estiver marcado. Na Figura 2.8 é apresentado um código que executa o GST.

|  |
| --- |
| **Greedy-String-Tiling** (String A, String B) {  tiles = {};  do {  maxmatch = MinimumMatchLength;  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 \oplus  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 > MinimumMatchLength);  return tiles;  } |

Figura 2.8 - Algoritmo *Greedy String Tiling*

Fonte: Prechelt, Malpohl e Philippsen (2002).

# 2.4.3 *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 ideia 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)*. Na Figura 2.9 é apresentado o algoritmo RKR-GST.

|  |
| --- |
| **Algoritmo** RKR-GST  **Entrada:** Cadeias A e B  **Saída:** ladrilhos  HA ← pares (assinatura,pos) da cadeia A  HB ← pares (assinatura,pos) da cadeia B  ladrilhos ← {}  emparelhmax ← M  **para todo** (hB, pos(hB)) em HB e pos(hB) não está marcada **faça**  **se** hB está em HA e sua posição em A não está marcada **então**  {Verifique o casamento}  pA ← pos(hA)  pB ← pos(hB)  i ← M - 1  **enquanto** ApA+i = BpB+i ˄ i ≤ 0 **faça**  i ← i - 1  **fim enquanto**  **se** i ≤ 0 **então**  Continue para próxima iteração  **fim se**  i ← 1  j ← emparelhmax  **enquanto** ApA+i = BpB+i e não atingimos o final de A ou B **faça**  i ← i + 1  j ← j + 1  **fim enquanto**  **se** j > emparelhmax **então**  emparelhmax ← j  emparelhamentos ← emparelhamento(pA,pB,j)  Marca temporariamente o ladrilho em A e B  **senão se** j = emparelhmax **então**  **se** ladrilho em A e B não está marcado temporariamente **então**  emparelhamentos ← emparelhamentos\oplus  emparelhamento(pA,pB,j)  Marca o ladrilho em A e B temporariamente  **fim se**  **fim se**  **fim se**  **para todo** (a,b,j) em emparelhamentos **faça**  **se** j = emparelhmax **então**  Marca o ladrilho com início na posição a de comprimento j na cadeia A  Marca o ladrilho com início na posição b de comprimento j na cadeia B  ladrilhos = ladrilhos ∪ emparelhamento(a,b,j)  **fim se**  **fim para**  **fim para** |

Figura 2.9 - Algoritmo *Running Karp-Rabin Greedy String Tiling*

Fonte: KLEIMAN, 2007

# 2.5 Ferramentas para detecção de plágio

Existem diversas ferramentas especificamente construídas para realizar a análise de similaridade (ou detecção de plágio) entre códigos-fonte A seguir são apresentadas as algumas dessas ferramentas.

# 2.5.1 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 o desempenho 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.

# 2.5.2 JPlag

O JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2000) é uma ferramenta de código fechado desenvolvida em Java para detecção de plágio e atualmente está disponibilizadano 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 2.10 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 eatualmente suporta várias linguagens de programação, tais como: Java, C#, C, C++e Scheme. No JPlag é possível comparar arquivos de texto em linguagem natural.

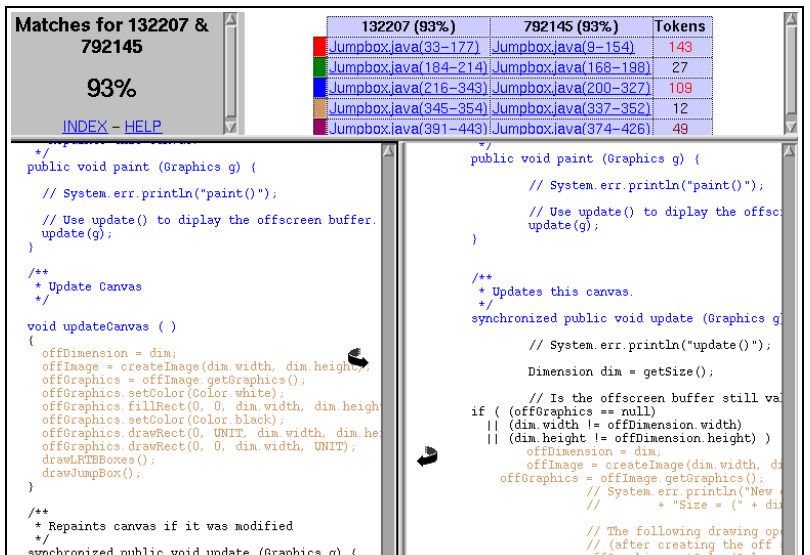


Figura 2.10 - Comparação entre dois códigos no JPlag

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

# 2.5.3 MOSS

*Measureof 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 deque, ao conhecer o funcionamento do sistema, poderiamburlar o sistema.

Da mesma forma que o JPlag, o MOSS também é acessado exclusivamentepor meio de um webservice, em que os código são submetidos utilizando umscript disponibilizado pelo autor e os resultados são processados imediatamente, em seguida esses resultados são apresentados em uma página HTML, que permanece disponível paraconsulta por 14 dias. 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.

# 2.5.4 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 softwaree detectar fragmentos duplicadas em grandes projetos de software. O algoritmo básico da ferramenta é (CLOUGH, 2000):

1. Cada arquivo é lido por um *tokenizer* apropriado as entradas e os *tokens* de um byte são armazenados em um vetor.
2. 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.
3. O algoritmo percorre todos os arquivos e encontra a melhor correspondência entre as sub-cadeias de caracteres (chamado de *run*).
4. Encontra o número da linha de início e fim de cada bloco pela posição do *token*.
5. 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.

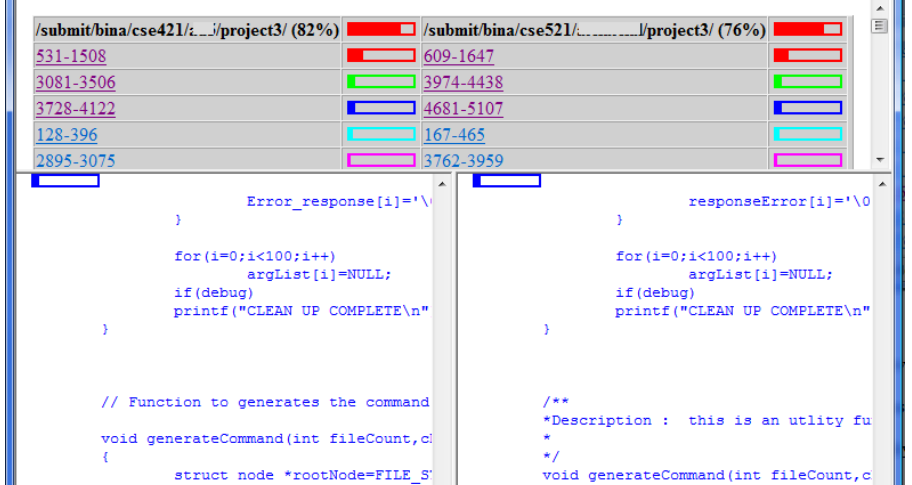


Figura 2.11 - Comparação entre dois códigos no MOSS.

Fonte: Adaptado de Ramarurthy e Settembre (2008)

# 2.5.5 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 apresentabom 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 umarquivo. O programa Comp usa esse arquivo de assinaturas para calcular emostrar o percentual de similaridade. Assim, a detecção de plágio do Sherlock se resume em duas etapas: geração de assinaturas e comparação de assinaturas. A seguir são descritas essas etapas.

**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, tabulaçõesou novas linhas. Na Figura 2.12 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 que𝑛 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 deassinaturas. A próxima etapa utilizará esses vetores de assinaturas para comparações.

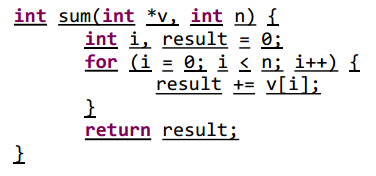


Figura 2.12 – Trecho de código com os *tokens* destacados

Fonte: Camargos (2013)

**Comparação de assinaturas**

A comparação das assinaturas dos códigos-fonte é 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.

# 2.5.5.1 Parâmetros do Sherlock

O Sherlock possui três parâmetros de configuração, são eles:

1. *Zerobits* (z): controla a granularidade da comparação. Esse valor pode ser qualquer número no intervalo [0-31]. Quanto menor o número, mais exata a comparação, porém mais lenta. Quando *z* = 0, todas as assinaturas geradas serão comparadas. O valor padrão para este parâmetro é 4;
2. *Chain length* (n): define a quantidade de *tokens* que forma 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 definido como padrão é 3;
3. *Threshold* (t): indica um valor de corte, em que expressa o nível de similaridade mínimo que um par de códigos deve apresentar para ser exibido. O valor definido como padrão é 20%.

# 2.5.5.2 Sherlock N-Overlap

Danilo (2014) afirma que o Sherlock utiliza métrica de similaridade análoga ao coeficiente de similaridade de Jaccard (JACCARD, 1901). Danilo (2014) estudou outras métricas a fim de verificar possíveis melhorias nos indicadores de similaridade e desenvolveu o Sherlock N-Overlap, que utiliza o coeficiente de similaridade Overlap como métrica de similaridade, pois torna o código-fonte mais robusto ao enxerto de código inútil.

# 2.6 Métricas de complexidade

Diversas ferramentas de analise de similaridade utilizam a técnica de contagem de atributos para identificar similaridades entre códigos-fonte. Essa técnica consiste basicamente em criar métricas para caracterizar diferentes elementos de um código (e.g., a frequência de *keywords*), extraindo dos códigos analisados uma série de parâmetros para obter um grau de similaridade através da comparação dos valores obtidos. A seguir são apresentadas algumas métricas de software.

# 2.6.1 Métricas de Halstead

As medidas de complexidade de Halstead tem por objetivo estabelecer medidores empíricos de eficiência e similaridade de diferentes códigos de uma mesma linguagem (CLOUGH, 2000).

De acordo com Halstead (1977), os algoritmos têm características mensuráveis​​, porque eles obedecem as leis da física. As métricas são baseadas em duas características: operadores e operandos. Operadores são símbolos que afetam o valor ou a ordenação de operandos. Os operadores podem consistir de símbolos adjacentes (tais como as letras que formam uma palavra), ou símbolos que estão fisicamente separados, mas que operam como uma unidade. Operandos são as variáveis ​​ou constantes em que os operadores atuam. Embora essas definições fornecem condições necessárias para a identificação de operadores ou operandos, eles não são suficientes.

Antes de aplicar métricas de Halstead, os operadores e operandos devem ser claros e precisamente definidos para a contagem de cada um. Estes contadores serão utilizados nas fórmulas que determinam o comprimento de um programa, o volume de um programa, a sua complexidade e outros atributos e são definidos da seguinte forma:

* η1: O número de operadores distintos,
* N1: O número total de operadores,
* η2: O número de operandos distintos,
* N2: O número total de operandos.

Halstead (1977) define as seguintes métricas:

1. Vocabulário: palavras
2. Tamanho: palavras
3. Tamanho estimado: palavras
4. Volume: *bits*
5. Volume potencial: *bits*
6. Nível:
7. Nível estimado:
8. Dificuldade:
9. Conteúdo inteligente: *bits*
10. Esforço de programação: discriminações
11. Tempo estimado de programação: segundos
12. Nível da linguagem:
13. Número de *bugs*: *bugs*

A explicação detalhada de cada fórmula pode ser encontrada em (DOT/FAA/CT-91/1). Nickerson (2006) afirma que versões aprimoradas dos conceitos introduzidos por Halstead são utilizadas até hoje como complemento em métodos de análise de similaridade em programas.

# 2.6.2 Complexidade ciclomática de McCabe

Thomas J. McCabe em 1976 criou a teoria da complexidade ciclomática, que consiste em uma métrica de *software* que é utilizada para indicar a complexidade de um programa de computador.

O resultado desta métrica define o número de caminhos independentes, que consiste em qualquer caminho ao longo do módulo que apresenta pelo menos uma nova condição ou um novo conjunto de comandos de processamento no conjunto base de um programa, e fornece um número mínimo de testes que devem ser realizados para que todos estes caminhos tenham sido testados pelo menos uma vez. Estes caminhos podem ser gerados a partir da representação de um grafo fortemente conectado com uma única entrada e uma única saída.

Pressman (2006) define a complexidade ciclomática (V) de três formas:

1. O números de regiões do grafo de fluxo;
2. , em que:

G = representa o grafo de fluxo.

E = representa a quantidade de arestas no grafo.

N = representa a quantidade de ramos no grafo.

1. , em que:

G = representa o grafo de fluxo.

P = representa a quantidade de ramos predicativos.

Para exemplificar a construção de um grafo de fluxo de controle e o cálculo da complexidade do mesmo é apresentado um código na Figura 2.13 e o respectivo grafo na Figura 2.14. Através dos grafos de fluxo, é possível verificar que:

1. Tem Cinco regiões.
2. V(G) = 13 arestas – 10 ramos + 2 = 5.
3. V(G) = 4 ramos predicativos + 1 = 5.

Assim, a complexidade ciclomática é 5. A tabela 2.2 classifica os códigos com base no V(G).

|  |
| --- |
| /\*1\*/ import javax.swing.JOptionPane;  /\*1\*/ public class PotencialTeste {  /\*1\*/ public static void main( String args[] ) {  /\*1\*/   String PrimeiroNumero,SegundoNumero;  /\*1\*/     float Base, Expoente,Resultado, Potencial;  /\*1\*/     PrimeiroNumero = JOptionPane.showInputDialog("Entra com a base:" );  /\*1\*/     SegundoNumero = JOptionPane.showInputDialog("Entra com o expoente:" );  /\*1\*/     Base = Integer.parseInt( PrimeiroNumero );  /\*1\*/     Expoente = Integer.parseInt( SegundoNumero );  /\*1\*/     if (Expoente < 0 ) {  /\*2\*/     Potencial=0-Expoente;  /\*3\*/     }  /\*3\*/     else {  /\*3\*/     Potencial=Expoente;  /\*4\*/     }  /\*4\*/     Resultado=1;  /\*4\*/     while (Potencial !=0 ){  /\*5\*/     Resultado = Resultado \* Base;  /\*5\*/       Potencial=Potencial-1;  /\*5\*/     }  /\*6\*/     if (Expoente<0 && Base !=0){  /\*7\*/     Resultado=1/Resultado;  /\*8\*/     }  /\*8\*/     else{  /\*8\*/     if(Base == 0){  /\*9\*/        JOptionPane.showMessageDialog(null,”A potencia é finita”);} /\*10\*/     }  /\*10\*/ JOptionPane.showMessageDialog(null,”A potencia é “ +Resultado, "Resultado", JOptionPane.PLAIN\_MESSAGE );  /\*10\*/ System.exit(0);  /\*10\*/  }  /\*10\*/ } |

Figura 2.13 - Código para exemplificar grafo de fluxo de controle

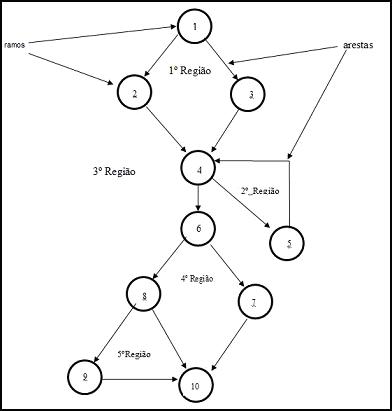


Figura 2.14 - Grafo de fluxo de controle do código

Fonte: http://www.devmedia.com.br/uma-visao-da-tecnica-de-teste-de-caixa-branca/15610

|  |  |
| --- | --- |
| **Complexidade** | **Classificação** |
| Entre 1 e 10 | Baixa complexidade. Fácil entender e testar. |
| Entre 11 e 20 | Média complexidade. Relativamente difícil entender e testar. |
| Entre 21 e 50 | Alta complexidade. Difícil entender e testar. |
| Maior que 51 | Altíssima complexidade. Impossível entender e testar. |

Tabela 2.2 - Classificação da complexidade de um código

A forma do grafo e a distância entre os valores da complexidade ciclomática para dois programas distintos, pode ser fundamental para decidir se existe plágio nos códigos (NICKERSON, 2006). A complexidade ciclomática de McCabe é, assim como as métricas de Halstead, uma ferramenta extremamente útil como complemento na busca por similaridades em códigos.

# 2.6.3 Outras métricas

Merlo (2007) criou uma ferramenta para detectar plagio entre códigos C/C++ de alunos que utiliza a técnica baseada em métricas. Apesar de não utilizar métricas mais complexas, ela tem como ideia central a utilização de sete contagens facilmente implementável. Mota et al (2010) afirma que os resultados obtidos com o CLAN foram suficientes para desencorajar a pratica de plágio pelos alunos. As sete métricas são:

1. Número de chamadas de funções
2. Número de variáveis locais usadas ou definidas
3. Número de variáveis não-locais usadas ou definidas
4. Número de parâmetros
5. Número de sentenças
6. Número de desvios
7. Número de loops