



UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

**Introdução de Métricas de Complexidade em Algoritmos de
Análise Léxica no Contexto da Detecção de Plágio**

Edson Augusto Bezerra Lopes

Fortaleza, Ceará
2013/2

EDSON AUGUSTO BEZERRA LOPES

INTRODUÇÃO DE MÉTRICAS DE COMPLEXIDADE EM ALGORITMOS DE
ANÁLISE LÉXICA NO CONTEXTO DA DETECÇÃO DE PLÁGIO

Projeto de Final de Curso apresentado à
Coordenação do Curso de Graduação em
Engenharia de Teleinformática da Universidade
Federal do Ceará como requisito parcial para a
obtenção do diploma de Engenheiro de
Teleinformática.

Orientador: Prof. Dr. José Marques Soares

Fortaleza, Ceará
2013/2

EDSON AUGUSTO BEZERRA LOPES

INTRODUÇÃO DE MÉTRICAS DE COMPLEXIDADE EM ALGORITMOS DE
ANÁLISE LÉXICA NO CONTEXTO DA DETECÇÃO DE PLÁGIO

Projeto de Final de Curso apresentado à
Coordenação do Curso de Graduação em
Engenharia de Teleinformática da Universidade
Federal do Ceará como requisito parcial para a
obtenção do diploma de Engenheiro de
Teleinformática.

Aprovado em: / /

BANCA EXAMINADORA:

Prof. Dr. José Marques Soares (orientador)
Universidade Federal do Ceará

Prof. Dr. George André Pereira Thé
Universidade Federal do Ceará

Prof. Dr. Giovanni Cordeiro Barroso
Universidade Federal do Ceará

Fortaleza, Ceará
2013/2

Ao meu avô, José Olivar Pereira Bezerra, por ter me ensinado a nunca desistir.

AGRADECIMENTOS

Primeiramente à minha família. Meus pais, avós e irmãos, especialmente minha mãe, tiveram um papel fundamental na minha formação, são os maiores motivadores do meu sucesso e foram fundamentais em momentos decisivos da vida.

À Narcisa, uma segunda mãe que pouco sabe sobre números e letras, mas foi provavelmente a pessoa com quem passei mais tempo até hoje e que me ensinou a importância da honestidade e da generosidade.

Aos bons professores que eu tive ao longo da vida, pela dedicação e portado o conhecimento compartilhado.

Ao Prof. Marques, orientador do projeto, e ao Danilo Leal, um de seus orientandos de mestrado, por terem sido sempre solícitos e dado uma contribuição essencial ao trabalho.

Aos colegas do curso de Engenharia de Teleinformática, especialmente aqueles que acabaram se tornando grandes companheiros fora do meio acadêmico ou que tiveram um papel fundamental em algum momento da minha graduação, como Felipe Barbosa, Ênio Rabelo, Rodrigo Cavalcante, Victor Farias, George Casé, Lana Beatriz, Ronaldo Milfont, Victor Borba, Daniel Rocha, Bianca Matos, Manoel Amora e vários outros.

Aos meus demais amigos, pelos fantásticos momentos de lazer, que me deram fôlego para continuar trabalhando duro, e pela ajuda em momentos de dificuldade.

Por fim, a todos aqueles que não fazem parte de nenhum dos grupos acima, mas colaboraram, direta ou indiretamente, na minha formação.

RESUMO

Entre as diversas aplicações das ferramentas de análise de similaridade em códigos-fonte, se destaca a detecção de plágio em disciplinas de programação, foco deste estudo. Com o aumento do número de alunos nas universidades, a identificação manual de irregularidades nas avaliações dessas disciplinas se tornou uma prática extremamente complicada para os docentes e contribui negativamente na formação de milhares de alunos.

Visando a compreensão e a melhoria de algumas dessas ferramentas, este trabalho analisa diversas técnicas utilizadas atualmente, como o método *Sherlock*, discute a aplicação de normalizações em códigos-fonte e se inspira em métricas de complexidade e *parsers* para introduzir um novo método.

Palavras-chave: **Detecção de plágio em software, Detecção de plágio em algoritmos, Similaridade de códigos-fonte.**

ABSTRACT

Among all source-code similarity analysis applications, the plagiarism detection in programming classes stands out and is the focus of this study. With the growing number of university students, the manual identification of plagiarism in the evaluation of these classes has become extremely complicated for teachers and contributes negatively to the formation of thousands of students

Targeting the understanding and the improvement of some of these tools, this work analyses several techniques used nowadays, such as the Sherlock Algorithm, discusses about source-code normalizations and presents a new technique based on parsers and complexity measures.

Keyword: *Software plagiarism detection, Algorithm plagiarism detection, Source-code similarity*

LISTA DE ILUSTRAÇÕES

Figura 1 - Componentes da complexidade na detecção de plágio	16
Figura 2 - Código para exemplificar grafo de fluxo de controle	21
Figura 3 - Grafo de fluxo de controle	21
Figura 4 - Exemplo de classificação em métodos de "tokenização"	23
Figura 5 - Exemplo de falha na análise sintática (parte 1)	24
Figura 6 - Exemplo de falha na análise sintática (parte 2)	24
Figura 7 - Exemplo de falha na análise semântica (parte 1)	25
Figura 8 - Exemplo de falha na análise semântica (parte 2)	26
Figura 9 - Exemplo de código básico de tokenizer	28
Figura 10 - Esquema de normalização pré-aplicação da ferramenta de detecção	32
Figura 11 - Representação gráfica da ferramenta SIM	33
Figura 12 - Comparação entre o método de Ottenstein e a ferramenta Plague	34
Figura 13 - Representação gráfica resumida do método Plague	38
Figura 14 - Exibição dos resultados do MOSS (parte 1)	38
Figura 15 - Exibição dos resultados do MOSS (parte 2)	39
Figura 16 - Exibição dos resultados do Jplag	44
Figura 17 - Execução do programa Sherlock.exe	47
Figura 18 - Experimento I, Código 1	48
Figura 19 - Experimento I, Código 2	49
Figura 20 - Experimentos II e III, Código 1	50
Figura 21 - Experimento III, Código 2	51
Figura 22 - Experimento IV, Código 1	53
Figura 23 - Experimento IV, Código 2	54
Figura 24 - Experimento V, Código 1	55
Figura 25 - Experimento V, Código 2	56
Figura 26 - Experimento VI, Código 1	57
Figura 27 - Experimento VI, Código 2	58
Figura 28 - Experimento VII, Código 1	59
Figura 29 - Experimento VII, Código 2	60
Figura 30 - Experimento VIII, Código 1	62
Figura 31 - Experimento VIII, Código 2	63

LISTA DE TABELAS

Tabela 1 - Comparação entre as diferentes versões do YAP	36
Tabela 2 - Comparação qualitativa das ferramentas	40
Tabela 3 - Resultados do Experimento I	48
Tabela 4 - Resultados do Experimento II	50
Tabela 5 - Resultados do Experimento III	52
Tabela 6 - Resultados do Experimento IV	53
Tabela 7 - Resultados do Experimento V	56
Tabela 8 - Resultados do Experimento VI	59
Tabela 9 - Resultados do Experimento VII	60
Tabela 10 - Resultados do Experimento VIII	61
Tabela 11 - Resumo dos Resultados	64

SUMÁRIO

1 INTRODUÇÃO	12
1.1Objetivos	13
1.2Organização do Trabalho	13
2 SIMILARIDADE EM CÓDIGOS-FONTE	14
2.1 Conceitos Gerais	14
2.2 Métodos Clássicos de Análise de Similaridade	16
2.2.1 Métricas	17
2.2.1.1 Métrica de Haslthead	18
2.2.1.2 Complexidade Ciclomática de McCabe	19
2.2.2 Tokens e Similaridade Sintática	21
2.2.2.1 Conceitos Gerais	21
2.2.2.2 O Algoritmo de Sherlock e as Normalizações	25
2.2.3 Análise Semântica	27
2.3 Ferramentas de Detecção de Plágio	
2.3.1 SIM	29
2.3.2 Plague	30
2.3.3 YAP	31
2.3.4 MOSS	34
2.3.5 JPlag	36
2.3.6 Resumo	38
3. METODOLOGIA	39
4. RESULTADOS TEÓRICOS E PRÁTICOS	42
4.1 Análise dos Algoritmos Utilizados	42
4.1.1 Algoritmo de Sherlock	42
4.1.2 Inserção de normalizações	44

4.1.3 Inserção de métricas de complexidade	44
4.1.4 Medida final de grau de similaridade após inserção da métricas	45
4.2 Experimentos	46
4.2.1 Experimento I – Informações pouco relevantes	46
4.2.2 Experimento II – Nomes de variáveis	48
4.2.3 Experimento III – Tipos de variáveis	49
4.2.4 Experimento IV - Mistura dos experimentos I, II e III	51
4.2.5 Experimento V – Presença de variáveis e operadores	53
4.2.6 Experimento VI – Posição de variáveis e expressões	56
4.2.7 Experimento VII – Estrutura semântica	58
4.2.8 Experimento VIII – Caso de alta complexidade	60
4.3 Resumo dos Resultados	63
5. CONCLUSÃO	64
5.1 Trabalhos futuros	65
6. REFERÊNCIAS BIBLIOGRÁFICAS	66
APÊNDICE A – ALGORITMO DE SHERLOCK	60
APÊNDICE B – CÓDIGOS NORMALIZADOS	78

1. INTRODUÇÃO

A classificação e a análise de similaridade em *softwares* são assuntos recorrentes nos meios acadêmico e industrial, dada a enorme variedade de aplicações e a complexidade que tais áreas de estudo atingem. Para comprovar tal afirmação, basta observar a quantidade de artigos científicos relacionados a estas práticas e a vasta bibliografia utilizada neste projeto e referenciada diversas vezes ao longo do capítulo.

Entre as utilizações mais comumente discutidas, se destacam as seguintes:

- Otimização de sistemas, através da releitura dos códigos componentes e da eliminação de trechos repetidos ou redundantes. As técnicas utilizadas nesse tipo de aplicação têm alto grau de similaridade com as que discutiremos nesse trabalho.
- Detecção de arquivos maliciosos, baseada na identificação e classificação de propriedades únicas desses programas, tais como características invariantes do código-fonte, e na utilização de técnicas de *Machine Learning*.
- Detecção de plágio em aplicações, seja ele nos códigos-fonte, caso que mais se enquadra no escopo deste trabalho, na versão binária dos arquivos ou no funcionamento dos algoritmos.

Essas aplicações são apenas alguns dos exemplos de extrema importância de tais técnicas, já que uma grande quantidade de novos *malwares*, quebras de patente e outras situações de sinistro são relatadas com frequência nas mais diversas indústrias e áreas do conhecimento (CESARE E XIANG, 2011).

De uma forma geral, a extração de características-chave únicas e o processamento destas para criação de uma métrica que define o nível de similaridade entre dois códigos-fonte são os métodos mais importantes e mais utilizados para identificar plágio, mas muitos estudos recentes se aproximam de bons resultados também na análise de desempenho de algoritmos para este fim, apesar da grande dificuldade técnica e da instabilidade dos resultados obtidos nos estudos mais recentes.

No contexto desse projeto, se destaca a aplicação dessas ferramentas na detecção de plágio em aulas de programação, já que a tarefa de encontrar eventuais fraudes em exames de alunos pode se tornar bastante complicada, principalmente quando mais de um professor é responsável pela correção ou quando o número de alunos ou a complexidade dos problemas aumenta, o que comprova que a automatização do processo de verificação de integridade das soluções propostas pelos estudantes é uma necessidade nos dias de hoje. Muito se pode fazer para avançar e obter bons resultados no curto prazo.

Terminada a discussão teórica que segue essa introdução, serão analisados estudos de caso, todos baseados em situações reais, o que ajuda consideravelmente a validar os métodos estudados.

Apresentam-se em seguida os objetivos do projeto e a organização do trabalho, bem como os maiores desafios esperados no processo de elaboração desse material.

1.1 Objetivos

O trabalho tem como objetivo principal a comparação e a melhoria dos resultados de diversos métodos de análise de similaridade em códigos-fonte escritos em C, além da implementação dos mesmos nessa linguagem.

Os objetivos específicos são:

- Efetuar um estudo teórico sobre as principais técnicas de detecção de plágio em código-fonte, além de implementar 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 que concerne à detecção de plágio.
- Introduzir métricas de complexidade ao algoritmo de Sherlock, demonstrando a melhoria considerável dos resultados obtidos. Entre as métricas que serão utilizadas, se destacam as de Halstead (Halstead, 2007) e as que foram utilizadas na elaboração da ferramenta CLAN (Merlo, 2007).

Entre os maiores desafios e dificuldades do projeto, se destacam a complexidade das técnicas existentes para obtenção de características-chave em algoritmos, a obtenção e compressão de detalhes sobre o funcionamento de algumas ferramentas, como o *Sherlock*, e eventuais complicações com a implementação de códigos.

1.2 Organização do Trabalho

O texto é dividido, a partir desse ponto, em quatro capítulos.

A seção 2 apresentará o embasamento teórico necessário para a realização dos experimentos que produzirão os resultados do projeto, sendo estudadas técnicas de análise sintática e semântica no contexto da detecção de plágio em códigos-fonte.

Dando sequência ao estudo, o terceiro capítulo apresenta a metodologia utilizada e os capítulos 4 e 5 apresentam, respectivamente, os resultados e a conclusão.

2. ANÁLISE DE SIMILARIDADE EM SOFTWARES

Este capítulo introduz a teoria necessária para compreensão dos resultados e é dada uma visão geral dos métodos mais utilizados na análise de similaridade em códigos-fonte, discorrendo sobre o funcionamento e os pontos fortes e fracos de alguns modelos clássicos de algoritmo, além de comparar as ferramentas existentes, sejam elas de código aberto ou não, e discute a necessidade de normalizar códigos-fonte para melhorar os resultados obtidos.

2.1 Conceitos Gerais

O primeiro questionamento ao estudar similaridades em *softwares* para as aplicações que interessam a essa pesquisa é sem dúvidas sobre o que o caracteriza exatamente o plágio.

Os meios digitais potencializam a cópia, portanto, estudos para proteger propriedade intelectual na internet ganham destaque. Os litígios relacionados a infrações de direitos autorais têm crescido, e com isto a necessidade de desenvolver técnicas que auxiliam na prevenção e detecção do roubo da propriedade intelectual (MAIA E GOYA, 2010).

Para o contexto deste trabalho, a definição se encaixa na ideia de Parker e Hamble, que o definem como um programa que foi produzido através de outro programa com um número pequeno de mudanças de rotina (CLOUGH, 2000).

Podemos também dizer que ele ocorre, no contexto de uma classe de programação em uma universidade, quando avaliações de diferentes estudantes foram copiadas ou transformadas de uma mesma versão de solução com esforço irrelevante na modificação de alguns parâmetros (JOY, 2001) ou quando características-chave do código não sofrem alterações suficientemente importantes para perder um caráter único de identificação do mesmo.

Segundo Culwin e Lancaster, o plágio é o roubo de propriedade intelectual, onde se inclui a utilização de código ou de programa sem permissão ou referência. Entende-se que os métodos desenvolvidos na luta contra o plágio podem ser divididos em prevenção e detecção, sendo necessária a combinação de ambos para obter sucesso.

Os métodos de prevenção extrapolam o esforço de trabalho do universo científico e envolvem a sociedade como um todo na mudança de atitude. Prevenir inclui fazer com que as pessoas entendam o plágio como um ato imoral, anti-ético e, acima de tudo, criminoso.

A prevenção, portanto, recai em ações e implantação de estratégias de longo prazo, que, além de demandarem tempo, dependem também de políticas públicas que amparem a proteção da propriedade intelectual.

De outro lado, há os métodos de detecção, foco desse trabalho, que apresentam resultados de curto prazo e que vêm sendo amplamente usados no auxílio às questões de litígio (CULWIN E LANCASTER, 2000).

Para justificar a viabilidade do estudo de ferramentas que combatam esse tipo de irregularidade e a possibilidade de obter bons resultados, basta observarmos o fato de encontrarmos mais facilmente referências bibliográficas que tratam sobre plágio em códigos-fonte do que em textos.

Outra motivação é o fato de partirmos da premissa de que as linguagens de programação seguem mais padrões e tem níveis de complexidade e vocabulário mais simples que os idiomas utilizados na comunicação convencional, além de terem uma quantidade bastante limitada de regras de sintaxe, sofrerem menos adaptações ao longo do tempo e não permitirem neologismos, o que explica as afirmações anteriores e chama atenção para a necessidade de conhecer intensamente a estrutura das mesmas para analisar a similaridade de códigos e até mesmo de algoritmos (CLOUGH, 2000).

As transformações utilizadas na cópia de um código-fonte podem ser bastante simples, como mudar comentários e nomes de variáveis, ou atingir níveis maiores de discrição, no caso de alterar uma estrutura de controle ou utilizar comandos similares (*switch* no lugar de *if*, por exemplo), o que requer um conhecimento maior da linguagem de programação e um entendimento do funcionamento do programa.

Faidhi e Robinson têm uma visão interessante das diferentes técnicas utilizadas na modificação de um código, como podemos observar na Figura 1, que ilustra, de dentro para fora, o aumento da dificuldade na detecção do plágio (FAIDHI e ROBINSON, 1987):

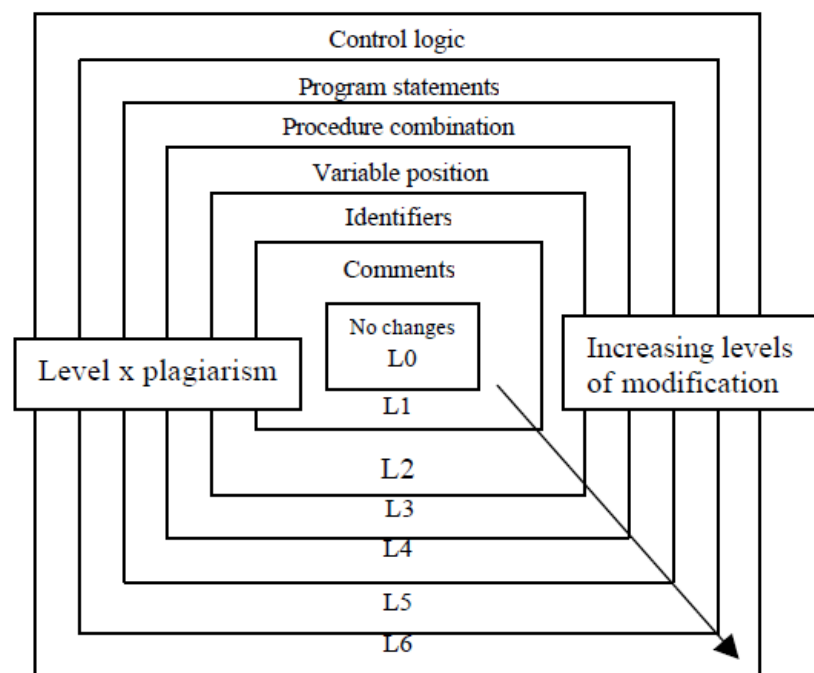


Figura 1: Componentes da complexidade na detecção de plágio (FAIDHI e ROBINSON, 1987)

Traduzindo o conteúdo da figura, observamos que as partes do código que podem eventualmente ser alteradas são, respectivamente e em ordem crescente de dificuldade de percepção:

- Comentários
- Identificadores (em geral, nomes de variáveis)
- Posição de variáveis no código, como no caso de tornar global uma variável local ou inverter a ordem de declaração das mesmas
- Combinação de procedimentos, como reposicionar trechos de código que podem ser trocados de ordem
- Comandos, como quando são adicionados elementos (parênteses ou chaves, por exemplo) ou se substitui um operador por outro similar
- Controle lógico, no caso de fazer alterações como trocar uma rotina que utiliza *for* por uma que utiliza *while*. Esse nível de dificuldade torna ainda mais importante a análise de similaridades em algoritmos, tema de um próximo capítulo.

Whale afirma que as modificações mais comumente realizadas são mudanças em comentários, identificadores e tipos de variável, seguidas de perto pela adição de trechos redundantes de código (WHALE, 1990).

Para concluir os conceitos gerais do capítulo, podemos classificar então as modificações em lexicais e estruturais. As primeiras não mudam o *parsing* do programa (ou seja, não alteram a forma com que ele é analisado por um compilador ou interpretador) e as outras, bem mais complicadas de se detectar, têm influência no modo com que a leitura do código é feita (JAMAL, 2012).

2.2 Métodos Clássicos de Análise de Similaridade

As ferramentas mais conhecidas para combater o plágio em códigos-fontes e baseiam essencialmente em cinco grupos de técnicas (JAMAL, 2012):

- Similaridade textual: Simples e rápido, esse método se baseia em comparações entre o texto propriamente dito do código. Incapaz de detectar modificações mais complexas, que envolvem alterações na lógica do programa, deve ser complementado por outras técnicas para obter resultados mais precisos.

- Métricas: Estabelece parâmetros e medidas calculadas a partir de propriedades dos diferentes trechos de um código, comparando os mesmos com outro código “suspeito”. Apesar de ter muitas falhas, como veremos na próxima seção, o método é simples de implementar e é um ótimo complemento nos algoritmos utilizados hoje em dia.
- Tokens: Consideradas como o maior avanço nesse domínio de estudo, as técnicas de “tokenização”, que transformam um código-fonte em *strings* de operadores, identificadores e outros tipos de elemento, são amplamente utilizadas hoje em dia, são a base para as ferramentas mais complexas e funcionam através da análise de similaridade no posicionamento dos componentes básicos de um código. Mais detalhes serão discutidos posteriormente.
- Árvores de Análise Sintática: Corresponde a uma das mais recentes técnicas. O conceito principal é baseado em uma representação em forma de árvore dos *tokens*, seguindo um conjunto de regras sintáticas próprias de cada linguagem.
- PDG (*Program Dependency Graphs*): A mais recente, complexa e eficiente ferramenta de combate ao plágio em códigos-fonte. Se baseia principalmente no estudo do fluxo de um programa através de sua representação em forma de grafo de dependência, uma maneira de apresentar os laços existentes entre os diferentes componentes de um *script*.

Abordaremos alguns desses tópicos nas próximas seções, buscando seguir a ordem cronológica em que foram desenvolvidos.

2.2.1 Métricas

As primeiras ferramentas utilizadas para detecção de plágio em *softwares* surgiram na década de 70 e se baseavam em técnicas de contagem de atributos, análise de grafos de fluxo de controle e medidas de complexidade, que consistiam basicamente em criar métricas para caracterizar diferentes elementos componentes de um código, como operadores, sendo Ottenstein, pesquisador da Universidade de Purdue, o pioneiro na aplicação de medidas já existentes, como a métrica de Halstead (CLOUGH, 2000).

Nesse último caso, o objetivo principal é criar uma série de parâmetros para diferentes códigos escritos na mesma linguagem e obter um grau de similaridade através da comparação dos valores obtidos, de forma que duas rotinas com alto grau de proximidade sejam fortes candidatos ao plágio.

Essas técnicas são também amplamente utilizadas para diagnosticar programas com níveis de complexidade acima do necessário (NICKERSON, 2006).

A ferramenta CLAN (MERLO, 2007), que foi desenvolvida para detectar plágio em classes de C/C++, é um ótimo exemplo. Apesar de não utilizar métricas mais complexas, como as que discutiremos posteriormente, ela tem como ideia central a utilização de sete contagens, é facilmente implementável e, segundo Merlo, teve um bom desempenho, desencorajando irregularidades entre os alunos (MOTA E GOYA, 2010). Os contadores utilizados no referido trabalho 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

Apresentaremos a seguir outras técnicas mais complexas, como as que foram utilizadas por Ottenstein na década de 70.

2.2.1.1 Métrica de Halstead

As medidas de complexidade de Halstead foram introduzidas pelo cientista da computação que dá nome à teoria no ano de 1977, como parte de um estudo que tinha como objetivo estabelecer medidores empíricos de eficiência e similaridade de diferentes códigos em uma mesma linguagem (CLOUGH, 2000).

Obtidas estatisticamente a partir do código, as medidas trazem informações importantes sobre o funcionamento do programa e foram baseadas em conceitos da física clássica, como volume, massa e pressão (HALSTEAD, 1977).

Para um dado *script*, sejam:

n_1 = número de operadores distintos

n_2 = número de operandos distintos

N_1 = número total de operadores

N_2 = número total de operandos

Definimos então as seguintes medidas:

Vocabulário: $n = n_1 + n_2$

Tamanho: $N = N_1 + N_2$

Volume: $V = N \log_2 n$

Dificuldade: $D = \frac{n_1 N_2}{2n_2}$

Esforço: $E = D \times V$

A explicação detalhada da motivação de cada uma dessas expressões foge do escopo deste trabalho, mas versões aprimoradas dos conceitos introduzidos por Halstead são utilizadas até hoje como complemento em métodos de análise de similaridade em programas (NICKERSON, 2006).

2.2.1.2 Complexidade ciclomática de McCabe

O método desenvolvido por Ottenstein se mostrou ineficiente para programas curtos ou que utilizassem modificações complexas, como as que foram sugeridas na primeira seção deste capítulo, o que ocasionou o surgimento de técnicas relacionadas a grafos de fluxo de controle (NICKERSON, 2006).

Complexidade ciclomática, ou condicional, é uma métrica de *software* usada para indicar a complexidade de um programa de computador. Desenvolvida por Thomas J. McCabe na década de 70, ela mede a quantidade de caminhos de execução independentes a partir de um código-fonte.

Essa complexidade é calculada através de um grafo que toma como nós os diferentes grupos de comandos que não podem aparecer desconexos no código e cria arestas direcionais indicando as diferentes possibilidades de ordem em que tais comandos podem ser executados para produzir o mesmo resultado. A métrica final é então a quantidade de diferentes caminhos que podem ser seguidos, mas diversas outras propriedades podem ser obtidas ao analisar as características dessa estrutura (MCCABE, 1976).

Sendo $G(n, e, p)$ o grafo, onde n representa o número de vértices, e representa o número de arestas e p representa o número de grafos desconexos da configuração, a complexidade ciclomática é definida como $V(G) = e - n + 2p$ (MCCABE, 1976).

Para exemplificar a construção de um grafo de fluxo de controle e o cálculo da complexidade do mesmo, consideramos o seguinte código e seu respectivo grafo (NICKERSON, 2006):

```

if (x < 0)
do {
if (y){

        b();
        m = c() * m;
    }
    } while (m < k);
else if (x == 0)
do{
if (y ==0){

b();
        c();
    }
    } while (x == 0);
else
do {
if (j){

        b();
        m = c() * 2m;
    }
    }while (m <= k);

```

Figura 2: Código para exemplificar grafo de fluxo de controle

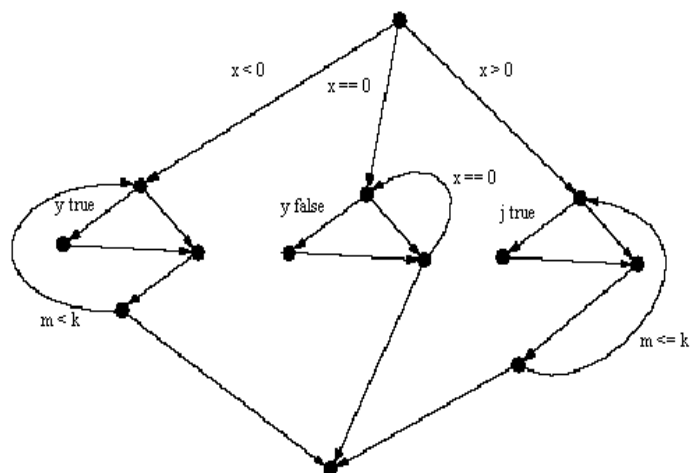


Figura 3: Grafo de fluxo de controle

Nesse caso, temos 20 arestas, 13 vértices e apenas um elemento completamente conexo, o que indica que o valor da complexidade, segundo McCabe, é 9.

Essa informação, por si só, não é muito significativa, mas a distância entre os valores para dois programas distintos, bem como a forma do grafo, pode ser fundamental para decidir qual o nível de risco de plágio (NICKERSON, 2006).

Esse método de análise tem diversas limitações, como o fato de diferentes expressões aritméticas no interior de comandos de controle de fluxo ou loops (como `if`, `for` e `while`) adicionarem o mesmo nível de complexidade ao grafo, mas a complexidade ciclomática de McCabe é, assim como a métrica de Halstead, uma ferramenta extremamente útil como complemento na busca por similaridades em códigos.

A primeira vez em que os dois métodos vistos até agora foram combinados para detectar plágio foi ainda na década de 70, quando Donaldson *et al.* propuseram um sistema chamado ACCUSE para Fortran (CLOUGH, 2000).

2.2.2 Tokens e Similaridade Sintática

2.2.2.1 Conceitos Gerais

No contexto do projeto, a “tokenização” pode ser considerada como o processo de quebrar um trecho de texto em palavras, frases, símbolos ou qualquer outro elemento significativo que possa vir a ter utilidade no processo de análise de similaridade. A lista de *tokens* obtidos se torna então a entrada para os algoritmos de processamento que serão utilizados na detecção de plágio (HUANG E SIMON, 2007).

No caso de códigos-fonte, o interesse pode ser concentrado em identificar e eliminar trechos irrelevantes para obter informação a partir de operadores e identificadores, mas a utilização de conteúdo textual pode também ter utilidade, o que indica que muitas vezes é necessário criar mais de um *token* para maximizar as chances de encontrar características particulares em um programa.

Esse método é útil tanto em linguística quanto em ciência da computação, sendo amplamente aplicável em diversos contextos e tendo diferentes níveis de dificuldade de implementação, dependendo da linguagem utilizada.

No caso de aplicar essa técnica com idiomas não ocidentais, o trabalho pode se tornar bastante complexo, como com o chinês, o grego e o russo, casos em que alfabetos diferentes são utilizados e são adotadas regras sintáticas completamente diferentes das do inglês. Esse problema não ocorre com a programação, mas existem diversos outros que são compartilhados entre as duas abordagens (CLOUGH, 2000).

Encarar situações mais complicadas em que é utilizada a “tokenização” pode envolver o desenvolvimento de heurísticas mais complexas, o tratamento exaustivo de exceções através de matrizes ou a criação de relações injetivas entre a linguagem foco e outras mais simples (HUANG E SIMON, 2007).

Em detecção de plágio, não é necessário que haja uma relação de igualdade completa entre os diferentes *tokens* de dois códigos, pois o objetivo principal é ter uma noção do quão “suspeitas” são as rotinas, deixando para algoritmos mais complicados (que também usam *tokens* para construir árvores e grafos), como os que mencionamos no início do capítulo, uma análise mais apurada (CLOUGH, 2000)

Para simplificar a compreensão, pode-se observar um exemplo simples de algoritmo de detecção de plágio usando “tokenização” (PARKER E HAMBLEN, 1989):

- 1) Remoção de todos os comentários
- 2) Remoção de espaços em brancos ou linhas extras
- 3) Obtenção de tokens e comparação dos seus caracteres (usando o comando *diff* no UNIX, por exemplo)
- 4) Armazenamento dos dados obtidos
- 5) Repetição do procedimento para todos os pares de códigos que devem ser analisados
- 6) Criação de um sumário em ordem decrescente de correlação
- 7) Análise manual dos pares mais relacionados

Dado um conjunto de códigos-fonte que executam a mesma tarefa, esse algoritmo é apenas uma forma de selecionar os pares com maior probabilidade de conter uma irregularidade, e não uma solução completa e próxima do determinismo, como as que já existem hoje.

Uma forma de melhorar essa técnica seria a utilização de diferentes *tokens* para diferentes tipos de elemento e a análise de *substrings* de mesmo tamanho de cada um deles, mas existem diversas outras, como as que discutiremos a seguir.

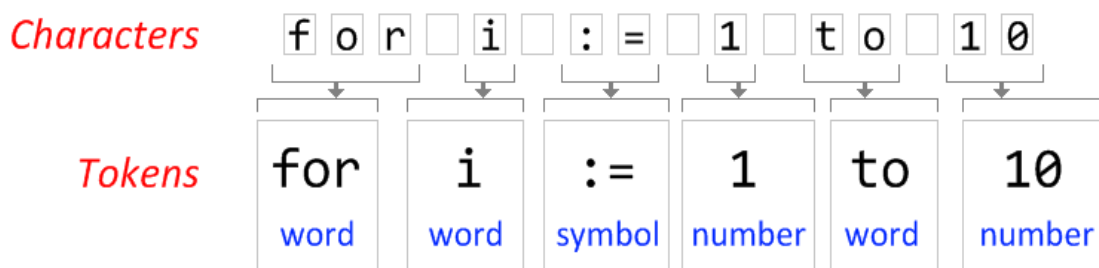


Figura 4: Exemplo de classificação em métodos de “tokenização”

Apesar da grande utilidade da análise de similaridade sintática (baseada nos elementos textuais do código-fonte e da linguagem, e não nos seus componentes lógicos, funcionais e de execução), podemos listar inúmeros exemplos em que a técnica de “tokenização” não é suficiente para identificar plágio. Observemos os seguintes trechos de código em C (MOTA E GOYA, 2010):

Código 1:

```
sum = 0 ;
void foo (Iterator iter){
    for(item=first(iter);has more(iter);item = next(iter)){
        sum = sum + value ( item ) ;
    }
}
```

Figura 5: Exemplo de falha na análise sintática (parte 1)

Código 2:

```
int bar (Iterator iter){
    for(item=first(iter);has more(iter); item = next(iter)){
        sum = 0 ;
        sum = sum + value ( item ) ;
    }
}
```

Figura 6: Exemplo de falha na análise sintática (parte 2)

Como pode se observar, os códigos apresentam grande similaridade sintática e executam tarefas completamente diferentes, o que ressalta a importância de introduzir conceitos semânticos, como faremos nas próximas sessões.

Além dessa diferença comportamental, a variável *sum* é global no primeiro bloco, enquanto no segundo é de escopo local, o que remete a outra diferença significativa no momento da execução.

Por outro lado, há casos em que trechos de código com pouca semelhança textual podem reproduzir exatamente o mesmo comportamento, como podemos observar nos códigos das figuras 7 e 8 (MOTA E GOYA, 2010):

Código 1:

```
while ((x = pi[t-1]) != '(' && x != '+' && x != '-') {
    vec[j++] = x;
    --t; }
```

Figura 7: Exemplo de falha na análise semântica (parte 1)

Código 2:

```
while(1){  
  
    x = pi[t-1];  
    if (x == '(' || x == '+' || x == '-') break;  
    --t;  
    vec[j++] = x;  
}
```

Figura 8: Exemplo de falha na análise semântica (parte 2)

Um programador experiente que quisesse dificultar o trabalho dos detectores de plágio poderia facilmente gerar o segundo a partir do primeiro e maximizar as chances de passar despercebido.

Nesse caso, mantivemos o nome das variáveis, mas trocá-los tornaria ainda mais complicada a análise, já que afetariamos também o *token* que contém os elementos textuais não componentes da linguagem.

Os exemplos acima estão em dois extremos dos problemas que uma ferramenta de detecção automática teria para ser bem sucedida, casos em que a observação humana e a experiência do analisador seriam imprescindíveis (MOTA E GOYA, 2010).

Como já vimos, o princípio básico da “tokenização” consiste em separar os diferentes tipos de elemento de um código. A ideia é percorrer o código-fonte e organizar as partes componentes (identificadores, comentários e operadores, por exemplo) em diferentes listas. Para exemplificar, podemos observar o código em C da Figura 9, que separa palavras de vírgulas em uma string.

Esse caso é obviamente simplista, mas introduz algumas funcionalidades da biblioteca de strings padrão que utilizaremos (*string.h*) e mostra a importância de utilizar a função *strtok*, que retorna um vetor contendo os diferentes elementos separados por um delimitador que é introduzido como parâmetro.

Em uma aplicação real, é necessário conhecer a sintaxe da linguagem e obter listas contendo, para cada tipo de elemento, todas as possibilidades. Uma vez definidos os grupos que desejamos separar e as listas com os possíveis elementos de cada um, a “tokenização” consiste na aplicação recorrente e levemente adaptada do algoritmo descrito na Figura 9 (HUANG E SIMON, 2007)..


```

#include<string.h>
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
    char *exemp[5];
    const char *frase="Ola,Como,voce,esta,?";
    int n=0,i;

    char *ds=strdup(frase);

    exemp[n]=strtok(ds, ",");
    while(exemp[n] && n<4) exemp[++n]=strtok(NULL, ",");

    for(i=0;i<=n;++i) printf("%s.", a[i]);
    putchar('\n');

    free(ds);

    return 0;
}

```

Figura 9: Exemplo de código básico de *tokenizer*

2.2.2.2 O Algoritmo de Sherlock e as Normalizações

Entre os algoritmos que se baseiam em análise léxica para detectar plágio, destaca-se o Sherlock (PIKE E WHITE, 2012), que é uma alternativa de código aberto, implementada originalmente em C, e que, portanto, pode ser objeto de estudos.

Para representar um fragmento do documento a ser comparado com outro, é gerada uma assinatura digital que calcula os valores de *hash* (ou seja, uma relação injetiva entre *strings* e números) para palavras e sequências de palavras, havendo ao final uma comparação para indicar o grau de semelhança (MACIEL, SOARES, FRANÇA E GOMES, 2012).

As maiores vantagens dessa técnica são a simplicidade de implementação, a velocidade de execução e a flexibilidade, já que a ideia por trás é realizar uma análise de semelhança léxica bem geral. É possível utilizá-la para detecção de plágio em documentos textuais e códigos-fonte em qualquer linguagem.

O método Sherlock funciona por meio da comparação de palavras separadas por espaços. Por isso, expressões que diferem na escrita (por ter um espaço a mais ou a

menos) e possuem o mesmo significado para o código são tratadas de forma diferente, o que é uma restrição que motiva a utilização de normalizações (MACIEL, SOARES, FRANÇA E GOMES, 2012).

Desenvolvida na Universidade de Warwick, na Inglaterra, essa técnica vem sendo utilizada e aperfeiçoada há vários anos, principalmente na utilização com códigos-fonte escritos em Java.

Outras ferramentas serão discutidas na próxima seção, mas o destaque dado ao método *Sherlock* é explicado pelo fato de ser ele o algoritmo que analisaremos e melhoraremos no projeto.

As técnicas puramente sintáticas, podem ser facilmente contornáveis se não houver um trabalho precedente de “limpeza” do código-fonte, eliminando-se trechos inúteis e eventuais redundâncias que possam ter sido inseridas com essa intenção. Esse processo, representado na Figura 10, chama-se normalização e vem colaborando como um suporte imprescindível na busca por melhorias em ferramentas de comparação textual (MAHMOUD, 2009).

Podemos exemplificar o processo de normalização com um *framework* sugerido em (MACIEL, SOARES, FRANÇA E GOMES, 2012) e identificado como “Normalização 4”. São quatro etapas, que devem ser executadas nessa ordem:

- Parte 1: Eliminação das linhas e espaços vazios, assim como referências a bibliotecas, havendo então a inclusão de espaços em branco entre estruturas, como identificação de variáveis e expressões.
- Parte 2: Remoção de todos os caracteres entre aspas
- Parte 3: Exclusão de valores literais e variáveis
- Parte 4: Supressão de todas as palavras reservadas

Uma padronização dos espaçamentos entre os diferentes elementos componentes que restam é feita ao final de cada uma das etapas da normalização e as regras seguidas podem variar, dependendo da situação.

Apesar de colaborar bastante no resultado do algoritmo *Sherlock*, como demonstrado no artigo de origem, 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 na próxima seção.

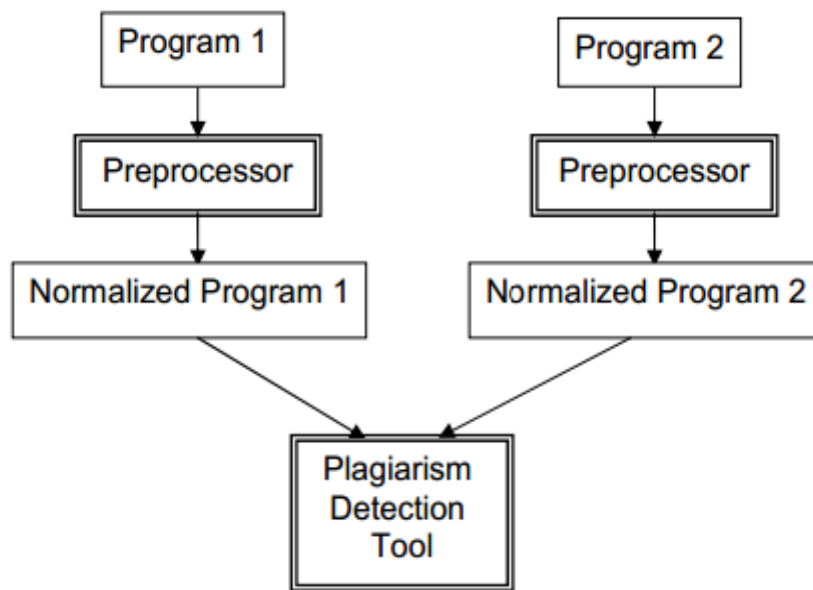


Figura 10: Esquema de normalização que precede aplicação da ferramenta de detecção

2.2.3 Análise Semântica

A semelhança semântica, como vimos anteriormente, está relacionada ao comportamento do programa e às funções que ele implementa, e não só aos elementos textuais que compõem o código-fonte, como nas seções anteriores.

O que separa essas técnicas das que estudam plágio em algoritmos, e não em códigos-fonte, é o fato de elas se basearem unicamente no conteúdo textual e na forma das ligações entre variáveis, operadores e estruturas de controle de fluxo, não havendo portanto uma análise da lógica do programa, o que é algo bem mais complexo e envolve uma análise completa do tempo de execução de cada um dos componentes fundamentais de um programa. Esse é um tema recente e não existem ainda muitas referências bibliográficas falando do assunto, o que dificulta a realização de experimentos para validar os poucos modelos existentes (ZHANG E JHI, 2012).

No contexto da análise semântica, observamos que, ao contrário do algoritmo de Sherlock, por exemplo, temos aqui uma forte dependência da linguagem em que é escrito o programa que será analisado. Outra grande dificuldade é encontrar métricas semânticas que sejam eficientes em diferentes tipos de contexto.

Existem basicamente dois tipos de similaridade no contexto da análise semântica (MOTA E GOYA, 2010):

- Semelhança funcional: Dois programas podem ser considerados análogos se implementarem uma função similar. Consiste basicamente em analisar a

similaridade entre as respostas de ambos quando submetidos a um conjunto de possíveis entradas.

- Semelhança de execução: É observada a sequência de execução do programa, em código *Assembly* ou em outras linguagens de baixo nível, como *BytecodeJava*. Nestes casos é necessário encontrar uma correspondência entre as intruções dos programas executáveis

Há um consenso quanto à dificuldade de obter medidas de similaridade semântica, o que é justificado pelo fato de podermos implementar de diversas formas o mesmo código. Um programador experiente pode tornar muito difícil a tarefa de detecção de plágio e até modificações simples, como as que foram mostradas nas figuras 7 e 8, podem ser consideradas complicadas para algumas ferramentas.

Apesar das dificuldades, existem ferramentas bastante eficientes de combate ao plágio baseadas nessa abordagem. As ideias mais comumente utilizadas para obter indícios de plágio são (CLOUGH, 2000):

- Análise da curva de execução: Traços de execução podem ser representados por uma curva. É possível obter informações importantes sobre similaridade semântica ao comparar curvas de diferentes programas em um dado tempo e espaço.
- Análise *input-output*: É observada a relação entre as saídas de dois programas distintos quando fazemos variar o conjunto de entradas.
- Análise de distância semântica: É calculado o custo de mutação de um programa para o outro através de um complexo método matemático.
- Análise da distância na equivalência de abstração: São removidos alguns componentes de um código com baixo nível de importância no mesmo e é feita uma comparação, verificando-se então o nível de abstração que um programa precisa alcançar antes que dois programas se tornem idênticos.
- Análise de similaridade nos grafos de fluxo de controle e dependência: Uma vez criados os grafos, é feita uma comparação. É a técnica por trás da maioria dos algoritmos recentes de detecção de plágio em software e tem inúmeras aplicações.

Como não serão utilizadas neste trabalho, as técnicas de análise semântica não serão discutidas em detalhes na parte teórica. Algumas ideias particulares serão eventualmente citadas nos capítulos 4 e 5, que descrevem a metodologia e os resultados obtidos no trabalho.

2.3 Ferramentas de Detecção de Plágio

Discutiremos aqui algumas das ferramentas mais importantes já desenvolvidas no combate ao plágio. Serão mostradas as principais ideias, vantagens e desvantagens de cada uma delas.

Antes de apresentá-las individualmente e falar de resultados, podemos estabelecer alguns critérios de avaliação que permitem ter uma visão mais clara das principais diferenças qualitativas existentes (HAGE, RADEMAKER E VUGT, 2010):

- Número de linguagens suportadas: Quanto mais flexíveis são as ferramentas, melhor, pois cresce consideravelmente o número de aplicações possíveis.
- Adaptabilidade: Algumas ferramentas podem ser facilmente alteradas para se tornar utilizáveis com outras linguagens.
- Qualidade de apresentação dos resultados: Diversas aplicações apresentam apenas o grau de similaridade, sem maiores detalhes nos critérios utilizados ou observações específicas para cada caso.
- Facilidade de utilização: Interfaces intuitivas são valorizadas e tornam mais prática a análise de similaridade.
- Exclusão de *templates*: Muitas vezes são fornecidos modelos de soluções para os alunos e a não exclusão de trechos desse tipo pode implicar em um falso grau elevado de similaridade.
- Manipulação de arquivos: Quando os códigos são divididos em vários arquivos, é necessário que haja uma técnica eficiente para balancear os pesos de cada um na análise de similaridade, bem como uma forma de avaliá-los individualmente e em conjunto.
- Local de hospedagem da ferramenta: Ao rodar localmente o programa de detecção de plágio, não somos obrigados a expor informações que podem ser confidenciais, como no caso em que a submissão é feita online (caso de algumas ferramentas gratuitas e de código-fechado, como o *JPlag*).
- Disponibilidade do código-fonte: Métodos *open source* são preferidos, tendo em vista que melhorias e adaptações podem ser realizadas, além da possibilidade realizar estudos mais detalhados dos resultados.

Tendo em vista que as primeiras ferramentas desenvolvidas, como a *Accuse*, acabaram caindo em desuso, discutiremos apenas descobertas mais recentes e que ainda são utilizadas. São elas: *SIM*, *Plague*, *YAP*, *MOSS* e *JPlag*, nessa ordem. Como *Sherlock* já foi apresentado anteriormente e ainda serão analisados diversos aspectos nas seções de metodologia e resultados, não haverá um sub-capítulo dedicado a essa ferramenta.

2.3.1 SIM (Software Similarity Tester)

Desenvolvida em 1989 na universidade de Vrije, em Amsterdam, essa é uma ferramenta de análise léxica que suporta as linguagens de programação *C*, *Java*, *Pascal*, *Modula-2* e *Miranda*, além de suportar linguagem natural (GRUNE E HUNTJENS, 1989).

Além de detectar plágio em códigos-fonte, é amplamente utilizada para encontrar fragmentos duplicados e redundantes em grandes sistemas, aplicação mais comum nos dias de hoje.

Em 2008, quando estava na versão 2.26, parou por um bom tempo de ser modificada pelos autores, razão pela qual o método caiu um pouco em desuso. Outro fato importante foi que a apresentação dos resultados, que necessitava de *Shell Script* e utilizava histogramas, era consideravelmente inferior a de outras ferramentas (HAGE, RADEMAKER E VUGT).

Independente das desvantagens, *SIM* é um ótimo exemplo de utilização de *tokenizers* na análise de similaridade. Seu algoritmo pode ser descrito basicamente pelos seguintes passos (CLOUGH, 2000):

- 1) Cada arquivo é lido por um *tokenizer* apropriado para a linguagem do *input* e *tokens* de um *byte*, representando os caracteres, são criados e armazenados em um vetor.
- 2) Cada *substring* do vetor é comparada com todas de mesmo tamanho que a seguem e é associada em uma tabela *hash* ao índice que aponta para o início da próxima *string* igual a ela, sendo considerado o valor 0 se não há correspondência.
- 3) O algoritmo percorre todos os arquivos e encontra as melhores correspondências entre as diferentes tabelas obtidas (chamados de *run*).
- 4) São observados e comparados os valores da linha inicial e da linha final de cada *run*
- 5) Os *runs* são ordenados em ordem de importância na obtenção de similaridade e os resultados são apresentados na forma de histogramas.

Como quase todas as ferramentas que utilizam unicamente análise sintática, *SIM* se torna menos eficiente quando modificações complexas são feitas na estrutura do programa, especialmente no caso em que há muitas alterações também no texto do código-fonte (ao trocarmos os comandos de controle de fluxo ou os *loops*, o que é bastante comum e não exige alto grau de conhecimento em programação).

Mesmo estando praticamente fora do mercado, é uma excelente fonte de estudo, já que é *Open Source*, fácil de usar e adaptável a outras linguagens. O algoritmo envolve diversos detalhes relacionados à flexibilidade dos *inputs* e a implementação original contém 1783 linhas em C++, mas a ideia geral pode ser facilmente compreendida, como o diagrama a seguir indica (GITCHELL E TRAN, 1997):

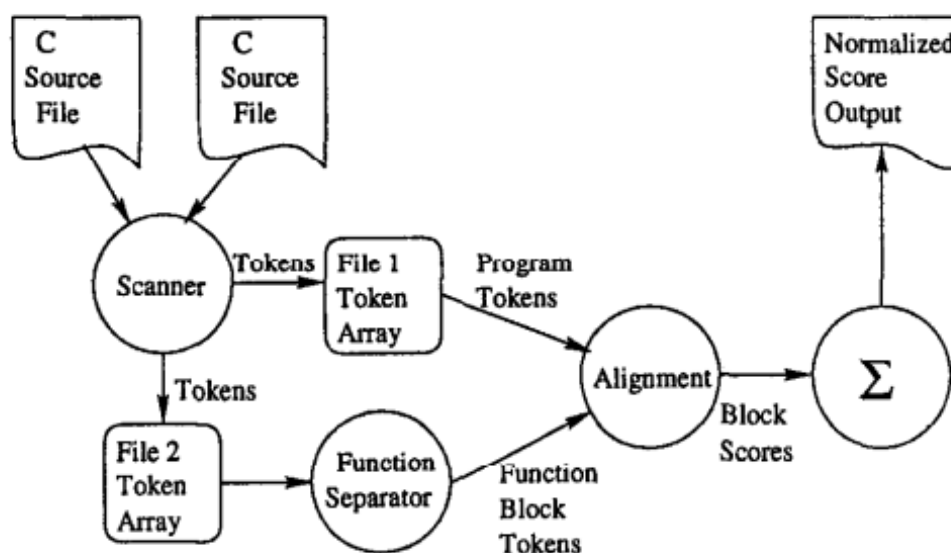


Figura 11: Representação gráfica da ferramenta SIM

2.3.2 Plague

O método *Plague* de detecção de plágio (WHALE, 1990) foi uma das primeiras soluções com bons resultados a incorporar os conceitos de análise semântica e obter características de similaridade a partir da estrutura do programa, e não apenas do seu código-fonte (CLOUGH, 2000).

Um educador que utilize o sistema pode introduzir códigos-fonte em Pascal, *Modula-2* e *Prolog*, linguagens bastante utilizadas na época da invenção do método, e observar como resultado o agrupamento de perfis de estrutura, criados a partir da análise de similaridade semântica entre cada par de entradas. A partir desses grupos, é possível realizar um estudo mais refinado, fazendo a comparação de *tokens* e utilizando técnicas convencionais de análise sintática (WHALE, 1990).

O *output* final do programa é uma lista organizada de pares de códigos em ordem decrescente de similaridade e contendo informações sobre os resultados obtidos, como os graus de semelhança em cada uma das etapas do processo (WHALE, 1990).

Na época do desenvolvimento da ferramenta, foram testados 245 programas em *Pascal*, com uma média de 200 linhas cada um, 24 deles contendo irregularidades dos mais diversos tipos e níveis de complexidade. Foram encontrados 22 códigos-fonte, um

resultado marcante para a época e excelente até os dias de hoje, o que comprova a eficiência da combinação das análises léxica e semântica.

Para efeito de comparação, foi feito um estudo entre os resultados do *Plague* e do clássico método de Ottenstein, que utiliza as medidas de complexidade de Halstead, havendo uma vantagem considerável da primeira ferramenta quando aumentamos o número de códigos-fonte no conjunto de entrada e o número de irregularidades presentes entre eles, como pode ser observado na Figura 12 (Sendo *Alternative* a linha relativa ao *Plague*) (CLOUGH, 2000):

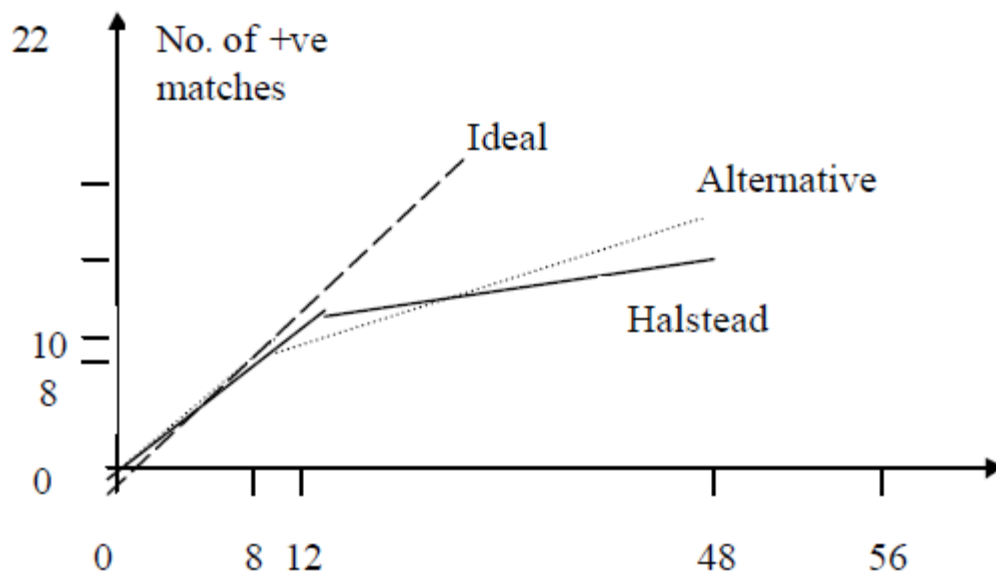


Figura 12: Comparação entre o método de Ottenstein e a ferramenta *Plague*

Além dos bons resultados, Whale introduziu quatro conceitos utilizados até hoje na comparação entre ferramentas de detecção de plágio (WHALE, 1990):

- Desempenho: Razão entre o número de documentos irregulares encontrados e o número total entre as entradas
- Precisão: Razão entre o número de documentos realmente irregulares e o número total encontrado pelo programa
- Sensibilidade: Grau mínimo de similaridade para que seja considerada a existência de plágio (esse valor é uma das entradas do *Plague*)
- Seletividade: Capacidade de manter alta precisão quando a sensibilidade aumenta

Para resumir, podemos dividir o algoritmo *Plague* em três etapas (CLOUGH, 2000):

- Criação de perfis de estrutura semântica, através da “tonenização” e da organização das diferentes estruturas de controle dos programas em algo que seria equivalente graficamente a um diagrama de blocos
- Um algoritmo com tempo $O(n^2)$ compara as diferentes estruturas e realiza um agrupamento, utilizando um conceito regulável de similaridade semântica e percorrendo as árvores que representam cada uma delas
- Utiliza-se uma métrica conhecida como *Longest Common Subsequence* (Maior sub-sequência em comum) para identificar similaridade entre os diferentes *tokens* de cada grupo de estruturas obtido na etapa anterior, havendo então um ordenamento e a exibição dos resultados finais

Na Figura 13 é apresentado o resumo do funcionamento da ferramenta (WHALE, 1990):

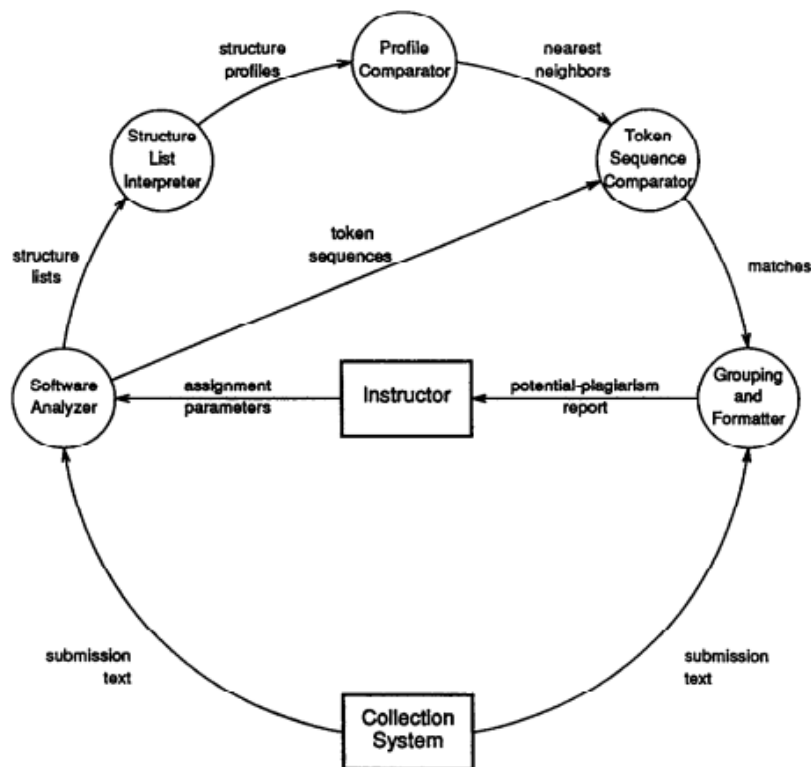


Figura 13: Representação gráfica resumida do método *Plague*

Apesar dos excelentes resultados, a técnica *Plague* não é mais utilizada com frequência para detecção de plágio no meio acadêmico, pois é muito complicado adaptá-la a outras linguagens de programação, a exibição dos resultados não é intuitiva (apesar de bastante completa) e existe uma forte dependência de algumas ferramentas *UNIX*, o que causa alguns problemas de portabilidade.

2.3.3 YAP

Sigla para *Yet Another Plague*, essa ferramenta é composta de uma série de técnicas desenvolvidas entre 1992 e 1996 por Michael Wise, pesquisador da Universidade de Sidney, sendo o *YAP3* a última versão, ainda utilizada com bastante frequência hoje em dia, dados os excelentes resultados que apresenta (JONES, 2007).

Baseados no *Plague*, como o nome sugere, os algoritmos *YAP* foram desenvolvidos visando corrigir falhas e melhorar a performance dos métodos de análise léxica e contagem de atributos que o precedem, 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).

Apesar de o processo de “tokenização” ser diferente para cada linguagem aceita, todas as versões do *YAP* utilizam a mesma ideia de base e apenas os algoritmos de triagem, alguns métodos de normalização e as métricas de similaridade sofreram alteração ao longo do tempo. Os principais pontos em comum antes da fase de comparação são as seguintes etapas (CLOUGH, 2000):

- Remoção de comentários, *strings* impressas na tela e letras não encontradas em identificadores
- Alteração de letras maiúsculas para minúsculas
- Formação de *tokens* primitivos após a primeira etapa de normalização
- Unificação de sinônimos, como no caso de funções diferentes que executam trabalhos similares (*strncmp* e *strcmp*, por exemplo)
- Identificação, organização e divisão de blocos de estruturas de controle das diferentes entradas do programa, havendo eliminação de blocos repetidos (que são, entretanto, indicados através de uma variável numérica que acompanha cada tipo diferente de *token*).

Uma vez que o problema está estruturado, cada versão do *YAP* trata a comparação de uma forma diferente. Com o *YAP1*, por exemplo, um simples algoritmo de análise sintática de tempo $O(n^2)$ era utilizado, o que foi acelerado no *YAP2* com a utilização do algoritmo de Heckel e melhorado mais uma vez no *YAP3*, onde é usado o algoritmo RKS-GST (WISE, 1996).

O grau de similaridade dos diferentes pares de entradas, representado em uma escala de 0 a 100, passa ainda por um estudo estatístico, o que é justificado pelo fato de pares com o indicador distante da média terem maior probabilidade de apresentar irregularidades (WISE, 1996).

Para comprovar a eficiência do método, Wise efetuou em 1992 um estudo comparativo entre o *YAP1* e a ferramenta *Plague*, introduzindo 167 programas escritos em Pascal por alunos de uma classe de programação. Sua ferramenta obteve 654 pares de códigos-fonte suspeitos, contra 640 do *Plague*. A grande surpresa foi o fato de mais de metade dos pares obtidos por cada método não terem sido encontrados pelo outro, apesar de o conjunto final de códigos suspeitos ter sido razoavelmente similar e o *YAP1* ter ficado mais próximo da solução perfeita. Essa diferença motivou a inserção de uma análise com certos aspectos ainda mais próximos do *Plague* na segunda versão, principalmente no que diz respeito a organização das estruturas similares (CLOUGH, 2000).

A principal novidade da terceira versão foi o novo algoritmo de detecção de similaridade (RKS-GST), capaz de lidar com sequências transpostas e de detectar trechos de códigos que foram fundidos pelos alunos, e não apenas transformações de trechos livres em funções, como nas versões anteriores (WISE, 1996).

O grande sucesso dos resultados foi comprovado em 1996, quando Wise decidiu estudar a evolução entre as três versões da sua ferramenta. Considerando 3 conjuntos de códigos-fonte (Q1, Q2 e Q3), com grau crescente de dificuldade de detecção de plágio, ficou nítido o progresso, como pode ser observado na Tabela 1, que mostra o percentual de irregularidades detectadas em cada grupo por cada versão (WISE, 1996).

	Q1	Q2	Q3
YAP1	73%	55%	65%
YAP2	98%	56%	62%
YAP3	100%	75%	75%

Tabela 1: Comparação entre as diferentes versões do *YAP*

A principal vantagem do *YAP*, quando comparado ao *Plague*, é a menor dependência da linguagem de programação utilizada, o que permite adaptá-lo aos dias de hoje, onde *C* e *Java*, por exemplo, são bem mais utilizadas.

Segundo Clough, a versatilidade da ferramenta vem do fato de ela ser imune a grande maioria das técnicas utilizadas por estudantes, tais como (CLOUGH, 2000):

- Mudanças de comentários e formatação
- Alteração de identificadores e tipos de variáveis
- Troca na ordem dos operadores
- Substituição por expressões equivalentes semanticamente (com leves falhas para alterações mais complexas)
- Adição de partes redundantes

- Mudança na ordem de estruturas de controle de fluxo, seleção e *loop*

Entre os pontos negativos, se destacam as falhas encontradas ao trocarmos a ordem dos elementos não semânticos que compõem o código e a ausência de uma ferramenta que indique similaridades parciais, como no caso da combinação de trechos plagiados e trechos originais.

2.3.4 MOSS

Desenvolvido em 1994 por Alex Aiken, pesquisador da Universidade de Berkeley, MOSS (*Measure of Software Similarity*) é gratuito e disponível online, mas seu código é fechado e o desenvolvedor nunca fez um estudo oficial sobre seus resultados, que foram amplamente discutidos por outros pesquisadores e educadores ao redor do mundo (HALL, 2010).

Apesar de ser extremamente eficiente no contexto da detecção de plágio em classes de programação, não apresenta um bom desempenho quando utilizada em grandes sistemas, principalmente os que são compostos por vários arquivos, o que é justificado pelos fatos de apresentar tempo de execução $O(n^3)$, segundo testes realizados por terceiros, e de não realizar análise de similaridade *cross-language* (MOTA E GOYA, 2010).

A ferramenta suporta códigos em diversas linguagens (*C*, *C++*, *Java*, *Javascript*, *Visual Basic*, *Pascal*, *Ada*, *ML*, *Lisp*, *Scheme* e até mesmo *Assembly*, por exemplo), mas não é possível determinar o grau de dificuldade de adaptação, já que não se sabe ao certo como é feito o seu código. A única coisa que é divulgada é o fato de se utilizar uma versão adaptada do método *Winnowing*, um complexo algoritmo de janelamento que utiliza propriedades estatísticas para obter “impressões digitais” de arquivos de códigos-fonte (SCHLEIMER E WILKERSON, 2002).

Dado um conjunto de entradas (no caso, soluções de alunos para um problema de programação), o MOSS, disponível como uma aplicação web, retorna uma página em *HTML* listando os diferentes pares em ordem decrescente de similaridade, como pode ser observado nas figuras 14 e 15 (RAMAMURTHY E SETTEMBRE, 2008).

Sobre o algoritmo utilizado, o que se sabe também é que existe uma normalização antes da aplicação do *Winnowing*, onde são eliminados trechos irrelevantes e são feitas padronizações distintas para cada linguagem. Depois disso, há o processo de janelamento, onde cada sub-string do texto é endereçada em uma tabela *hash*. Essas são as únicas etapas discutidas pelo autor da ferramenta (CLOUGH, 2000).

Moss Results - Windows Internet Explorer

http://moss.stanford.edu/results/376397665/

Moss Results

Wed Dec 17 19:05:51 PST 2008

Options -l c -d -m 40

[[How to Read the Results](#) | [Tips](#) | [FAQ](#) | [Contact](#) | [Submission Scripts](#) | [Credits](#)]

File 1	File 2	Lines Matched
/submit/bina/cse421/...h/project3/ (99%)	/submit/bina/cse521/...project3/ (99%)	4425
/submit/bina/cse421/...5/project3/ (82%)	/submit/bina/cse521/...1/project3/ (76%)	3874
/submit/bina/cse421/...1/project3/ (80%)	/submit/bina/cse521/...1/project3/ (83%)	3587
/submit/bina/cse521/...1/project3/ (97%)	/submit/bina/cse521/...1/project3/ (97%)	3284
/submit/bina/cse421/...1/project3/ (96%)	/submit/bina/cse521/...1/project3/ (96%)	3677
/submit/bina/cse521/...1/project3/ (62%)	/submit/bina/cse521/...1/project3/ (70%)	2917
/submit/bina/cse521/...1/project3/ (92%)	/submit/bina/cse521/...1/project3/ (93%)	2527
/submit/bina/cse521/...1/project3/ (87%)	/submit/bina/cse521/...1/project3/ (89%)	2399
/submit/bina/cse521/...1/project3/ (87%)	/submit/bina/cse521/...1/project3/ (90%)	2487
/submit/bina/cse521/...1/project3/ (89%)	/submit/bina/cse521/...1/project3/ (88%)	2421
/submit/bina/cse521/...1/project3/ (89%)	/submit/bina/cse521/...1/project3/ (87%)	2472
/submit/bina/cse521/...1/project3/ (88%)	/submit/bina/cse521/...1/project3/ (89%)	2438
/submit/bina/cse521/...1/project3/ (89%)	/submit/bina/cse521/...1/project3/ (89%)	2484
/submit/bina/cse421/...1/project3/ (55%)	/submit/bina/cse421/...1/project3/ (67%)	2207
/submit/bina/cse521/...1/project3/ (84%)	/submit/bina/cse521/...1/project3/ (85%)	2327
/submit/bina/cse521/...1/project3/ (84%)	/submit/bina/cse521/...1/project3/ (82%)	2352

http://moss.stanford.edu/results/376397665/matches/

Internet | Protected Mode: On

Figura14: Exibição dos resultados do MOSS(parte 1)

Matches for /submit/bina/cse421/au5/project3/ and /submit/bina/cse521/arkarkal/project3/ - Windows Internet Explorer

http://moss.stanford.edu/results/376397665/match1.html

Moss Results

Matches for /submit/...

/submit/bina/cse421/au5/project3/ (82%)	/submit/bina/cse521/arkarkal/project3/ (76%)
531-1508	609-1647
3081-3506	3974-4438
3728-4122	4681-5107
128-396	167-465
2895-3075	3762-3959

```

Error_response[i]='\
}

for(i=0;i<100;i++)
    argList[i]=NULL;
if(debug)
    printf("CLEAN UP COMPLETE\n")
}

// Function to generates the command
void generateCommand(int fileCount, c
{
    struct node *rootNode=FILE_S
    char totalCommand[SIZE], temp
    struct node *temp=rootNode;
    char *finalCommand;
    while(temp!=NULL)
    {
        sprintf(totalCommand
        strcat(totalCommand,

```

Internet | Protected Mode: On

Figura 15: Exibição dos resultados do MOSS (parte 2)

2.3.5 JPlag

Tida como uma das técnicas mais bem sucedidas no contexto da detecção de plágio em classes de programação, o *JPlag* é outro exemplo de ferramenta de código fechado e disponível gratuitamente *online*.

Desenvolvida em 1996 por Tichy e Malpohl, pesquisadores do departamento de computação da Universidade de Karlsruhe, a plataforma suporta apenas códigos em *Java*, *C*, *C++* e *Scheme*, que são provavelmente as linguagens mais utilizadas hoje em dia no meio estudantil (CLOUGH, 2000).

Apesar de não serem divulgadas muitas informações sobre as ideias por trás do *JPlag*, sabemos que são utilizados conceitos de análise sintática e léxica, como nos casos de *MOSS*, *YAP* e *Plague*, e que existe um processo de normalização antes da criação dos *tokens* (HAGE, RADEMAKER E VUGT, 2010).

O algoritmo utilizado para a comparação de *strings* é uma versão otimizada e mais eficiente do *Greedy String Tiling*, de Michael Wise, que é o algoritmo usado no *Plague* e que tem complexidade linear, apresentando excelentes resultados mesmo para grandes conjuntos de entradas (HAGE, RADEMAKER E VUGT, 2010).

Assim como o *MOSS*, a apresentação dos resultados é feita em páginas *HTML*, mas dessa vez estas são enviadas ao cliente e armazenadas localmente, o que representa uma vantagem no quesito segurança da informação. A página principal mostra uma lista de pares de entradas em ordem decrescente de similaridade com representações gráficas da distribuição dos valores obtidos para os graus de semelhança, além de pares de código similares selecionados, como podemos observar na figura16 (BABAR, 2012).

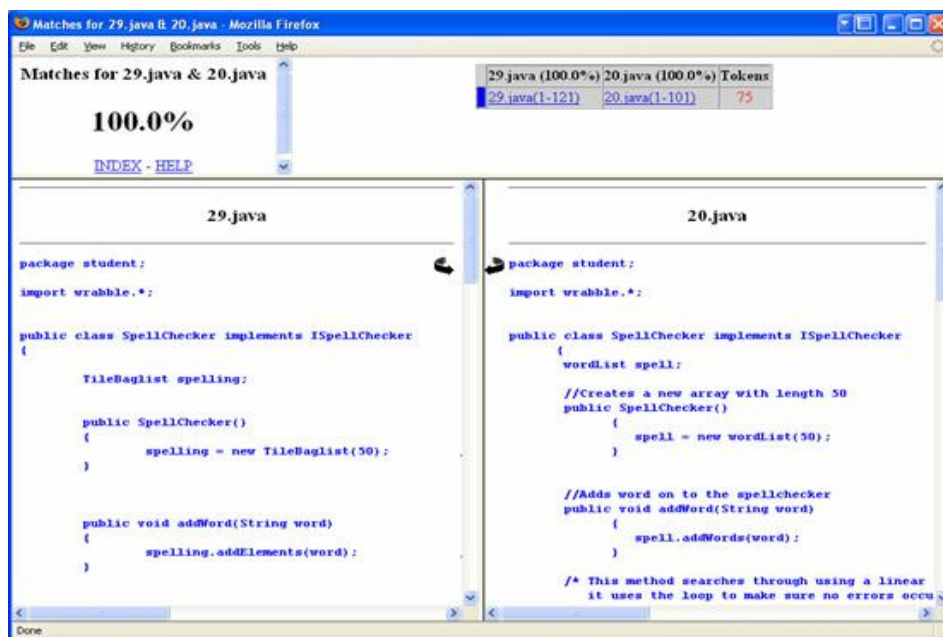


Figura 16: Exibição dos resultados do *Jplag*

Fácil de usar, o *JPlag* tem sido amplamente utilizado em universidades e as críticas encontradas *online* são extremamente positivas.

2.3.6 Resumo

Na Tabela 2 são apresentadas as principais informações qualitativas da seção 2.3, além de alguns conceitos discutidos anteriormente sobre o algoritmo de *Sherlock*.

	Sherlock	SIM	Plague	YAP3	MOSS	JPlag
Número de linguagens suportadas	Todas	5	4	3	23	6
Adaptabilidade a outras linguagens	Sim	Sim	Complexa	Sim	Não	Não
Qualidade e apresentação dos resultados (1 – 5)	4	2	2	3	4	5
Facilidade de utilização (1 – 5)	5	2	2	4	4	5
Exclusão de <i>Templates</i>	Não	Não	Não	Não	Sim	Sim
Manipulação de arquivos	Não	Não	Não	Não	Sim	Sim
Local de hospedagem	Local	Local	Local	Local	WEB	WEB
Código-fonte disponível	Sim	Sim	Sim	Sim	Não	Não

Tabela 2: Comparação qualitativa das ferramentas

3. METODOLOGIA

Visando a obtenção dos resultados necessários para comprovar a utilidade das métricas de complexidade e das técnicas de normalização na melhoria de algoritmos de análise sintática, foram realizadas comparações de desempenho entre os três métodos, na seguinte ordem de evolução:

- 1) Algoritmo Sherlock
- 2) Algoritmo Sherlock com códigos normalizados (utilizando as técnicas de normalização desenvolvidas em um artigo de MACIEL, SOARES, FRANÇA E GOMES)
- 3) Mistura do algoritmo de Sherlock com códigos normalizados e análise de métricas de complexidade em códigos-fonte (CLAN e Haslthead)

Cada ferramenta foi estudada e implementada individualmente e nessa ordem em linguagem C, sempre tomando como base a versão precedente. Detalhes sobre o funcionamento de cada uma delas serão apresentados ao longo do capítulo 4 e os respectivos códigos podem ser localizados nos apêndices.

Após estudar detalhadamente o funcionamento de cada método, foram realizados experimentos com a utilização de oito pares de códigos-fonte similares e escritos manualmente com inserção proposital de diferentes técnicas de plágio, conforme o seguinte roteiro:

- Par de códigos I: Adição ou eliminação de informação pouco importante (como bibliotecas e comentários), espaços e linhas em branco
- Par de códigos II: Mudança de nomes de variáveis
- Par de códigos III: Mudança no tipo de variáveis
- Par de códigos IV: Mistura das técnicas de plágio utilizadas nos três primeiros pares
- Par de códigos V: Inclusão e eliminação de variáveis e operadores inúteis
- Par de códigos VI: Mudanças na posição de variáveis ou expressões
- Par de códigos VII: Alterações na estrutura semântica do programa (principalmente operadores e controle de fluxo)
- Par de códigos VIII: Mistura de todas as técnicas de plágio utilizadas anteriormente

A análise de cada caso foi feita individualmente e foram sempre apresentados o par de códigos utilizados e um estudo comparativo resumindo os resultados obtidos, bem como identificando eventuais falhas e os principais pontos fracos de cada algoritmo no contexto do experimento.

O algoritmo de Sherlock necessita de algumas entradas (definidas no próximo capítulo) para definir a precisão e a sensibilidade dos resultados. Foram considerados diferentes cenários para essas variáveis e serão citadas na conclusão (capítulo 5) algumas sugestões de técnicas de detecção de plágio baseadas na variação desses parâmetros.

O acesso ao material utilizado nos experimentos do projeto foi gratuito e todas as referências bibliográficas utilizadas foram disponibilizadas pelo orientador, através da Universidade, ou são encontradas na *internet*. Segue a lista e a fonte de obtenção de cada um deles:

- Notebook Dell Vostro 1520
- Sistema Operacional Windows
- Compilador DEV C/C++: *Software* gratuito
- Base de códigos-fonte para testes: Desenvolvida no contexto do projeto
- Código original do algoritmo de Sherlock: Obtido no site da Universidade de Sidney
- Plataforma de testes do *Moodle*: Disponibilizada pelo orientador

4. RESULTADOS TEÓRICOS E PRÁTICOS

4.1 Análise dos Algoritmos Utilizados

O primeiro grande desafio do projeto foi a compreensão teórica dos dois primeiros algoritmos, que já foram obtidos em suas versões implementadas, e das métricas de complexidade de Merlo e Halstead, que serviram de inspiração para a criação da terceira ferramenta, principal resultado do trabalho.

4.1.1 Algoritmo de Sherlock

Base para as três técnicas utilizadas nos experimentos, a versão original do Algoritmo de Sherlock (encontrada no Apêndice A) consiste em um processo de “tokenização” que compara trechos diferentes de um dado par de códigos através de *hash*, o que permite que a sensibilidade da ferramenta seja controlada, como veremos mais na frente, e que o método seja menos sensível à mudanças nas posições de estruturas independentes.

As principais funções do código-fonte do algoritmo podem ser resumidas em:

- Função de leitura: É um dos elementos mais importantes do algoritmo. Com essa função (*read_word*) é possível ler um arquivo de texto (no nosso caso, um código .c) a partir de um certo ponto e retornar a primeira palavra delimitada por um vetor de operadores pré-definidos em um de seus parâmetros. A leitura do arquivo ignora espaços, quebras de linhas e caracteres também definidos em um parâmetro de entrada. Cada uma dessas palavras obtidas é um *token*.
- Função *Hash*: Outro elemento essencial. Dado um vetor de caracteres, a função retorna um número inteiro não negativo que é único para cada possibilidade de *string*. Valores próximos indicam alto grau de similaridade.
- Função de assinatura: Cada código-fonte tem uma assinatura, que é um vetor contendo os valores de hash de cada sub-vetor de N *tokens* consecutivos obtidos com a função de leitura, onde N é uma variável de entrada do programa, como veremos mais na frente.
- Função de comparação: Após a obtenção das assinaturas de cada código-fonte de entrada, o Algoritmo de Sherlock calcula o nível de similaridade de todos os pares (a proporção de elementos em comum entre duas assinaturas sobre o número total de elementos), ordena os resultados e expõe em um arquivo de saída.

Para inicializar o programa, comparar dois códigos-fonte em C e expor o resultado em um arquivo de texto, é necessário acessar a pasta em que o executável se encontra através do *cmd* do *Windows* e digitar um comando como o seguinte:

Sherlock.exe -t a% -z b -n c -o output.txt Arquivo1.c Arquivo2.c

As variáveis utilizadas durante a iniciação representam diferentes opções de configuração e têm o seguinte significado:

t: Vem da palavra *Threshold* e é o nível de similaridade mínimo que um par de códigos deve apresentar para ser exposto no arquivo de resultados. O padrão sugerido pelos autores é 20%, mas resolvemos utilizar 0% para ter uma análise mais precisa.

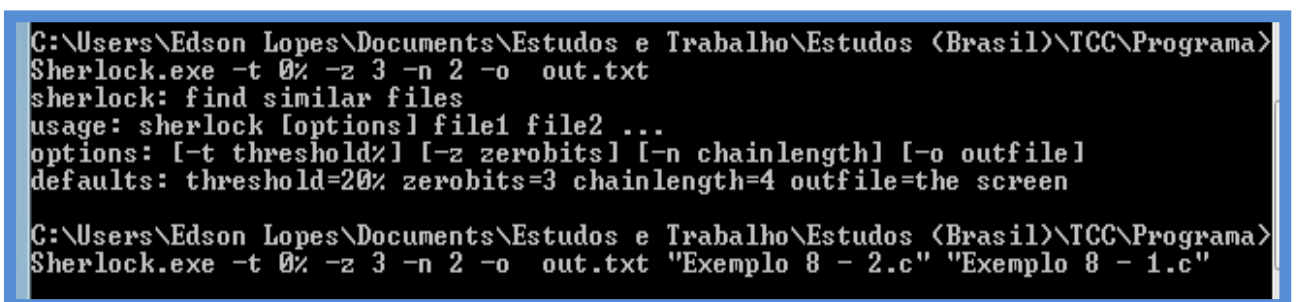
z: Significa *Zerobit* e está relacionado com a precisão do programa. Pode variar de 0 a 31 e quanto menor o valor, menor a rigidez na detecção de plágio. Essa variável decide quantas casas decimais eliminar no valor de *hash* associado aos vetores de *tokens*, o que determina o quão próximos dois deles tem que ser para serem considerados similares. Consideramos o valor dessa variável igual a 3.

n: É o número de *tokens* armazenados no vetor que gera os valores de *hash* da assinatura. Não existe uma relação precisa de crescimento ou decréscimo quando aumentamos o seu valor, mas números menores tendem a tornar o método mais rígido (ou seja, mais sensível ao plágio). O valor sugerido é 3, mas criamos cenários com 2 também.

o: É o nome do arquivo de saída que contem os resultados

Uma observação muito importante no contexto dos experimentos que serão realizados é o fato de valores pequenos de “n” (menores que 3) gerarem falsos níveis altos de similaridade para códigos grandes. Como são utilizados apenas códigos pequenos, como os que são produzidos por alunos de classes básicas de programação, o valor 2 também será utilizado, pois se mostrou mais eficiente nesse contexto.

Na Figura 17 é apresentado um exemplo de mensagem que é mostrada quando a entrada de dados não é feita corretamente e, logo abaixo, um exemplo correto de inserção de parâmetros.



```
C:\Users\Edson Lopes\Documents\Estudos e Trabalho\Estudos (Brasil)\TCC\Programa>
Sherlock.exe -t 0% -z 3 -n 2 -o out.txt
sherlock: find similar files
usage: sherlock [options] file1 file2 ...
options: [-t threshold%] [-z zerobits] [-n chainlength] [-o outfile]
defaults: threshold=20% zerobits=3 chainlength=4 outfile=the screen

C:\Users\Edson Lopes\Documents\Estudos e Trabalho\Estudos (Brasil)\TCC\Programa>
Sherlock.exe -t 0% -z 3 -n 2 -o out.txt "Exemplo 8 - 2.c" "Exemplo 8 - 1.c"
```

Figura 17: Execução do programa Sherlock.exe

4.1.2 Inserção de Normalizações

Os métodos utilizados foram basicamente aqueles que foram discutidos na seção 2.2.2.2. Foram testadas quatro possibilidades, sendo a n -ésima delas uma combinação das n primeiras etapas do *framework* sugerido naquele momento (de um total de quatro).

Apesar de o tema ter sido amplamente estudado durante a elaboração da parte teórica da dissertação, não foi necessária nenhuma implementação, pois o orientador do projeto disponibilizou uma plataforma *online* gratuita, imbutida no portal acadêmico *Moodle*, que permite a realização do processo. Bastava fazer o *upload* dos códigos-fonte e baixar as versões normalizadas.

Além dessa funcionalidade, a plataforma fornecida permite realizar testes com diversos algoritmos de detecção de plágio, personalizar normalizações e configurar detalhes bastante específicos, o que foi bastante útil para comparar o desempenho dos métodos estudados nesse projeto e de ferramentas existentes no mercado, como o *JPlag* e o *SIM*.

Consideramos como resultado de cada experimento, para o oito casos, a média dos dois maiores graus de similaridade obtidos ao testarmos o par de códigos-fonte com o Algoritmo de Sherlock após a aplicação de cada uma das quatro formas de normalização.

Podemos observar alguns exemplos de códigos normalizados no Apêndice B.

4.1.3 Inserção de Métricas de Complexidade

Como descrito nos capítulos anteriores, as métricas de complexidade tem como objetivo a extração de características do funcionamento de códigos-fonte que possam indicar similaridade.

Baseado nas técnicas utilizadas por Merlo e Haslthead em suas respectivas soluções, foram criados conjuntos formados por 10 elementos para cada código-fonte (antes da normalização). São eles:

- Número de operadores distintos
- Número total de operadores
- Número de palavras reservadas de declaração distintas
- Número total de palavras reservadas de declaração
- Número total estimado de estruturas semânticas

- Número total de palavras reservadas não contadas anteriormente
- Vocabulário (número de operadores distintos + número de operandos distintos)
- Tamanho (número total de operadores + número total de operandos)
- Volume (medida descrita no capítulo 2 que indica o quão grande é um código que executa uma determinada tarefa)
- Dificuldade (medida descrita no capítulo 2 que indica o nível de complexidade de análise de um dado programa)

Uma vez construídos os vetores (digamos A e B , com elementos a e b) e escolhida uma variável X , que vale entre 0 e 1 e está relacionada com o nível de precisão do método, podemos definir o grau de similaridade G , expresso em percentagem, como:

$$N = \text{número de posições dos vetores para as quais } \max\left(\frac{|b-a|}{a}, \frac{|b-a|}{b}\right) \leq X$$

$$G = \frac{N}{10}, \text{ se } N > 4, \text{ ou } 0, \text{ caso contrário}$$

Para exemplificar, suponhamos que os vetores tem 2 elementos, e não 10, que $X = 0.3$ e que $(A,B) = [(8,2),(10,3)]$. Nesse caso, a similaridade é de 50%, pois os valores 8 e 10 tem diferença percentual menor que 30%, valor de X , mas os valores 2 e 3 não satisfazem a mesma propriedade, resultando em $G = 1/2$.

Visando a obtenção de resultados coerentes, a variável X foi considerada individualmente para cada critério. Para as medidas de Halstead, foi utilizado o valor 0.1, para o número total de operadores o valor foi 0.2 e nos demais casos foi 0.15, escolhas estas que tinham como objetivo calibrar a o rigor de cada medida de acordo com seu funcionamento.

4.1.4 Medida final de grau de similaridade após inserção das métricas

Uma vez definidas as medidas de similaridade individuais do Algoritmo de Sherlock e das métricas mencionadas na seção anterior, é necessário unificar os resultados e obter um valor final para o terceiro método estudado no projeto.

Dado que as duas abordagens são bastante independentes, consideramos apenas uma média aritmética dos resultados obtidos com o segundo método (códigos normalizados + Algoritmo de Sherlock) e do valor de G que foi descrito anteriormente.

Foi observado que, apesar da simplicidade da fórmula, as respostas finais encontradas exprimem bem a contribuição de cada metodologia.

4.2 Experimentos

Serão apresentados aqui oito experimentos, cada um deles baseado em um par de códigos-fonte contendo as irregularidades listadas no capítulo 3. Os arquivos são analisados pelos três métodos estudados na sessão anterior e é feita uma análise dos resultados para dois valores possíveis de N (número de elementos do vetor de *tokens*).

4.2.1 Experimento I – Informações pouco relevantes

Nesse experimento foram utilizados dois códigos-fonte similares (Figuras 18 e 19) que realizam a inicialização de uma matriz 10x10 com algumas fórmulas e tem como saída a soma de todos os valores obtidos.

```
#include <stdio.h>
#include <string.h>

int main(void){

    int i,j ;

float matriz[10][10], s = 0;

    for (i = 0; i<10; i++){

        for(j = 0; j<10; j = j+1){

            /*Formula e incremento do valor de s    */

            matriz[i][j] = 1/(i+j+1); s = s+matriz[i][j];
            }
        }
    /*Fim do Loop*/

    printf("%f",s);

    return 0;

/*      Fim      */

}
```

Figura 18: Experimento I, Código 1

```

#include <stdio.h>

#include <stdio.h>

/*Soma de elementos da matriz base*/

int main(void){

    int i,j ;
    float matriz[10][10], s = 0;

    /*Loop*/

    for (i = 0; i<10; i++){
        for(j = 0; j<10; j = j+1){

matriz[i][j] = 1/(i+j+1);
            s = s+matriz[i][j];
        }

    }

    printf("%f", s);
    return 0;

}

```

Figura 19: Experimento I, Código 2

Foram alteradas as posições e o conteúdo de alguns comentários, além da inserção de espaços e linhas em branco. O conteúdo útil do programa é exatamente o mesmo.

O grau de similaridade obtido para cada método é apresentado a seguir:

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
66%	100%	95%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
50%	100%	95%

Tabela 3: Resultados do Experimento I

Podemos observar na tabela que o algoritmo de Sherlock é sensível à mudanças de comentários, já que não apresenta um grau de similaridade próximo de 100%. Isso pode

ser explicado pelo processo de formação dos *tokens*, que não descarta a presença de texto, mesmo que não haja influência no funcionamento do programa.

A presença de normalizações antes da utilização do algoritmo melhora totalmente a análise de similaridade, como pode ser observado, e isso vem do fato de uma das etapas do processo consistir justamente em eliminar comentários.

As métricas não são nem mesmo necessárias aqui, mas a estrutura idêntica dos códigos faz com que toda e qualquer medida de complexidade indique 90% de similaridade, fato que pode ser útil quando o Algoritmo de Sherlock não funciona bem (o que na prática está relacionado a um uso abusivo e desproporcional de comentários, situação que não ocorre frequentemente).

4.2.2 Experimento II – Nomes de variáveis

Foram utilizados aqui códigos praticamente idênticos e bem simples (Figuras 20 e 21). Em ambos os casos é calculada a soma dos números de 1 até o último elemento da sequência de Fibonacci menor do que 1000, sendo trocados os nomes de todas as variáveis.

```
#include <stdio.h>
#include <stdlib.h>

int main(void){

    int a = 0, b = 1;
    int i;
    int c;

    while(b < 1000){

c = b+a;

        a = b;
        b = c;

    }

    int s;
    s = 0;
    for (i = 0; i<b; i++) s = s+i;

    return 0;

}
```


Figura 20: Experimentos II e III, Código 1

```
#include <stdio.h>
#include <stdlib.h>

int main(void){

    int xx = 0, xy = 1;
    int loop;
    int count;

    while(xy < 1000){
        count = xx+xy;
        xx = xy;
        xy = count;
    }

    int sum;
    sum = 0;
    for (loop = 0; loop<xy; loop++) sum = sum+loop;

    return 0;

}
```

Figura 21: Experimento II, Código 2

Os resultados obtidos foram os seguintes:

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
40%	62,5%	81,25%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
12%	50%	75%

Tabela 4: Resultados do Experimento II

É fácil notar que o primeiro método não apresenta resultados consistentes e que o segundo, já bem mais eficiente, é satisfatório (principalmente para N = 3, onde a mudança é bastante expressiva).

A explicação é basicamente a mesma do Experimento I. O tratamento de conteúdo textual (no nosso caso, tudo que não é operador) pelo algoritmo de Sherlock não é eficiente no contexto da detecção de plágio em códigos-fonte, apesar de ser uma grande vantagem quando utilizamos linguagem natural, e a utilização de normalizações para eliminar esse problema tem um papel fundamental.

As métricas indicam 100% de semelhança quando consideradas independentemente e agregam bastante valor ao resultado nos casos apresentados, o que pode ser melhor observado no contexto de códigos mais complexos.

4.2.3 Experimento III – Tipos de variáveis

São utilizados aqui programas idênticos aos do Experimento II, mas com outra abordagem.

O primeiro código é exatamente igual (Figura 20), mas o segundo sofre agora alterações diferentes do caso anterior. É trocado o tipo de algumas variáveis, e não o nome delas, como podemos observar na Figura 22.

```
#include <stdio.h>
#include <stdlib.h>

int main(void){

    long int a = 0, b = 1;
    float i;
    int c;

    while(b < 1000){

        c = b+a;
        a = b;
        b = c;

    }

    unsigned int s;
s = 0;
    for (i = 0; i<b; i++) s = s+i;

    return 0;

}
```

Figura 22: Experimento III, Código 2

Os resultados obtidos nesse caso (Tabela 5) foram satisfatórios com todos os métodos, inclusive o *Sherlock* sem normalização, que se mostrou falho no último.

Podemos concluir com isso que as mudanças desse tipo são menos eficientes para plagiadores do que mudanças de nomes de variáveis. Isso ocorre pois o tipo tende a aparecer com menos frequência na proximidade de operadores, fazendo parte então de *tokens* maiores e tendo menos influência na composição dos valores de *hash*.

Códigos mais complexos poderiam exigir a presença de normalizações ou de métricas para a obtenção de uma boa análise de similaridade, mas nesse caso não foi preciso.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
100%	100%	75%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
66%	65%	57,5%

Tabela 5: Resultados do Experimento III

4.2.4 Experimento IV - Mistura dos experimentos I, II e III

Visando a análise de um caso mais complexo, foram misturadas todas as técnicas de plágio dos três primeiros textos (todas elas envolvendo alterações em elementos textuais do código).

Os programas utilizados nesse caso (Figuras 23 e 24) calculam o menor valor K tal que a soma dos fatoriais de 1 até K excede 10.000.000, além de apresentar o valor da soma.

Os resultados (Tabela 6) indicaram uma baixa eficiência do Algoritmo de Sherlock, como esperado. Essa é uma consequência natural da complexidade atingida ao misturar três métodos diferentes de plágio, dois deles com resultados razoáveis em experimentos anteriores.

Como podemos observar, o segundo método melhora consideravelmente a performance do primeiro e já se mostra suficiente para obter um alto grau de similaridade, o que é devido principalmente às alterações relativas ao Experimento II (modificação do nome de variáveis), que são “corrigidas” com a normalização.

Em códigos maiores, entretanto, as normalizações se tornam menos eficientes e é nesse contexto, quando uma grande quantidade de conteúdo textual foi modificado e a estrutura semântica permanece idêntica, que as métricas se mostram mais úteis (apesar de já agregarem valor ao resultado nesse caso).

As semelhanças entre pares de programas satisfazendo esses critérios podem ser facilmente detectáveis manualmente para códigos pequenos, mas se tornam bastante complexas em casos maiores, o que justifica a importância da inserção de medidas de complexidade para complementar os métodos de análise sintática.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
21%	61,5%	70,75%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
6%	60%	70%

Tabela 6: Resultados do Experimento IV

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*Funcao que calcula o fatorial*/

int fatorial(int n){

    if (n == 1) return 1;
    else return n*fatorial(n-1);

}

/*Main*/

int main(void){

    printf("%d", fatorial(10));
    int k = 0;
    int n = 1;

    /*Loop para calcular soma dos fatoriais que não excede
    10^10*/
    while (k < pow(10,7)){

        k = k + fatorial(n);
        n++;

    }

    printf("\nMin: %d\n",n);

    printf("Valor: %d", k);
    return 0;

}
```

Figura 23: Experimento IV, Código 1

```

#include <stdio.h>
#include <math.h>

int fat(int x){
    if (x == 1) return 1;

    else return x*fat(x-1);

/*Fim da funcao fatorial*/
}

void main(void){

printf("%d" , fat(10));

    long int sum = 0;
    unsigned int nn = 1;

    while (sum < pow(10,7)){

/*calculando...*/

        sum = sum + fat(nn);

        nn++;
    }

/*Impressao de resultados*/

printf("\nMinx: %d\n",nn);

printf("Valor: %d", sum);

return 0;

}

```

Figura 23: Experimento IV, Código 2

4.2.5 Experimento V – Presença de variáveis e operadores

Nesta etapa dos experimentos, são inseridas alterações um pouco mais complexas. A adição e eliminação de variáveis inúteis parece facilmente detectável em códigos pequenos, mas no contexto de um grande projeto pode se tornar um grande desafio, pois a detecção manual de plágio se torna praticamente inviável.

Além disso, é necessário entender que adicionar variáveis implica em adicionar operadores e tipos (Em C, por exemplo, temos vírgula, ponto-e-vírgula e asteriscos, no caso de utilização de ponteiros), o que altera completamente a estrutura dos *tokens* que são construídos nos algoritmos convencionais de análise léxica.

Os códigos-fonte para esse caso implementam o mesmo programa dos experimentos II e III, mas ambos contêm variáveis inúteis, caso que pode ocorrer quando dois códigos distintos foram copiados da mesma fonte (Figuras 24 e 25).

```
#include <stdio.h>
#include <stdlib.h>

int main(void){

    int a = 0, b = 1;
    int m;
    int i;
    char *c;

    while(b < 1000){
        float j, t;
m = b+a;

        a = b;
        b = m;
    }

    int s;
    s = 0;
    for (i = 0; i<b; i++) s = s+i;

    return 0;

}
```

Figura 24: Experimento V, Código 1

```

#include <stdio.h>
#include <stdlib.h>

int main(void){

    int a = 0, b = 1, l, d, m;
    int i, j, k;

    while(b < 1000){
        m = b+a;
        a = b;
        b = m;
    }

    int s;
    float u;
char r[10];
    s = 0;
    for (i = 0; i<b; i++) s = s+i;

    return 0;

}

```

Figura 25: Experimento V, Código 2

Observamos aqui (Tabela 7) que o algoritmo de Sherlock se mostra mais uma vez insuficiente para detectar o plágio, o que é justificando pelas razões já mencionadas.

O processo de normalização melhora consideravelmente os resultados obtidos e é aqui uma forma muito útil de se combater o plágio, principalmente quando não há adição abusiva de conteúdo inútil (o que pode ser detectado manualmente em casos extremos).

Nesse caso, onde há alteração na proporção entre o número de operadores e operandos, as métricas começam a se mostrar menos eficientes que o segundo método, pois, embora a estrutura semântica se mantenha idêntica, os valores das medidas de complexidade dos dois códigos são distintas. Ainda sim há um bom resultado.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
57%	83%	71,5 %
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
28%	60%	60%

Tabela 7: Resultados do Experimento V

4.2.6 Experimento VI – Posição de variáveis e expressões

São introduzidas aqui as primeiras modificações que alteram não só o conteúdo textual e o número de variáveis e operadores, mas também a ordem de execução de blocos independentes do código. A inserção de variáveis (seção 4.2.5) é uma forma de mudança de posição das partes do código-fonte original, mas não de ordem.

Apesar de a estrutura semântica continuar idêntica nesse caso, essas alterações mudam consideravelmente o texto do código e, intuitivamente, deveriam implicar em resultados ruins com algoritmos de análise sintática.

O que acontece na prática é justamente o contrário, no caso do Algoritmo de Sherlock. Como dito na seção 4.1, a comparação das assinaturas obtidas é feita elemento por elemento, não importante a ordem em que se encontram no vetor, o que faz com que esse método seja pouco sensível a alterações na ordem dos elementos do código.

```
#include <stdio.h>

int sq(int u){
    return u*u+1;
}
int triang(int u){
    if(u == 0) return 0;
    else return u+ triang(u-1);
}
int main(void){
    int i,j, m = 9, n, cont1 = 0, cont2 = 0, sum = 0;
    int matriz[10][10];
    for (i = 0; i<10; i = i+1){
        for (j = i; j<10; j = j+1){
            matriz[i][j] = triang(i+j);
            cont1++;
            sum = sum+matriz[i][j];
        }
    }
    while (m > -1){
        n = m-1;
        while(n > -1){
            matriz[m][n] = sq(m+n);
            n = n-1;
            cont2 = cont2 + 1;
            sum = sum - matriz[m][n];
        }
    }
    sum = (cont1 + cont2)*(sum-1);
    return 0;
}
```

Figura 26: Experimento VI, Código 1

Os códigos utilizados (Figuras 26 e 27) inicializam uma matriz com diferentes funções implementadas no começo do texto e tem como resultado uma fórmula aplicada sobre os valores obtidos. Apesar de idênticos em termos funcionais, foi alterada a ordem de dois laços, de algumas declarações de variáveis e funções e de operações aritméticas.

```
#include <stdio.h>

int triang(int u);
int sq(int u){
    return u*u+1;
}
int main(void){
    int matriz[10][10];
    int cont1 = 0, cont2 = 0, sum = 0, i, j, m = 9, n;
    while (m > -1){
        n = m-1;
        while(n > -1){
            matriz[m][n] = sq(m+n);
            n = n-1;
            cont2 = cont2 + 1;
            sum = - matriz[m][n]+sum;
        }
        for (i = 0; i<10; i = i+1){
            for (j = i; j<10; j = j+1){
                matriz[i][j] = triang(i+j);
                cont1++;
                sum = matriz[i][j]+sum;
            }
        }
        sum = (sum-1)*(cont1 + cont2);
        return 0;
    }
    int triang(int u){
        if(u == 0) return 0;
        else return u+ triang(u-1);
    }
}
```

Figura 27: Experimento VI, Código 2

Como dito antes, os resultados foram positivos para o primeiro método (Tabela 8) e observamos que foram melhores ainda para $N = 3$, algo inédito. Isso ocorre pois a utilização de um número maior de *tokens* nos sub-vetores faz com que blocos independentes de código que sofreram apenas alteração na ordem de execução sejam menos separados no momento do cálculo dos valores de *hash* e possamos obter um grau maior de similaridade. Como veremos, essa vantagem desaparece quando utilizamos a combinação desta com outras técnicas de plágio.

As normalizações não tiveram aqui um resultado tão relevante quanto nas vezes passadas, o que pode ser explicado pela eliminação de trechos textuais importantes, como o nome de variáveis, fazendo com que os *tokens* sejam menores e, portanto, mais sensíveis a “quebras” quando fazemos alterações na ordem dos blocos.

Como o número de operandos e operadores praticamente não mudou, as métricas foram bastante interessantes para complementar o resultado, mas o alto grau de similaridade obtido com os métodos anteriores não torna a inserção das medidas essencial.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
75%	77%	88,5%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
77%	54%	77%

Tabela 8: Resultados do Experimento VI

4.2.7 Experimento VII – Estrutura semântica

O estudo das técnicas de análise semântica mencionadas no capítulo 2 fogem do escopo do projeto, mas vale a pena estudar casos em que há utilização delas para identificar eventuais falhas nos métodos estudados nesse trabalho.

Foi utilizado um par de códigos bem simples nesse caso (Figuras 28 e 29). Ambos têm como função a inicialização de uma matriz seguindo algumas regras e o cálculo da soma dos elementos da mesma.

A estrutura foi completamente alterada através da substituição de laços (*for* por *while*), da adição de variáveis e da troca de comandos de controle de fluxo (*if* e *switch*, no caso).

```
#include <stdio.h>

int main(void){
    int i,j, matriz[5][5], sum = 0;
    for (i = 0; i<5; i = i+1){
        for(j = 0; j<5; j++){
            if(i == j){
                matriz[i][j] = 2;sum = sum + 2;
            }
            else if (i<j){
                matriz[i][j] = 0;
            }
            else{
                matriz[i][j] = 1;sum++;
            }
        }
    }
    return 0;}
```

Figura 28: Experimento VII, Código 1

```
#include <stdio.h>

int main(void){
    int i = 0, j = 4, c, matriz[5][5], sum = 0;
    while (i<5){
        while(j >-1){
            if(i==j) c = 2;
            else if (i>j) c = 1;
            else c = 0;
            matriz[i][j] = c;
            switch(c){
                case 0:
                    break;
                case 1:
                    sum +=1;
                    break;
                case 2:
                    sum+=2;
                    break;
            }
            j--;
        }
        i++;
    }
}
```

Figura 29: Experimento VII, Código 2

Dado alto nível de modificação do conteúdo textual, os resultados ruins obtidos para os dois primeiros métodos já eram esperados.

As métricas aqui foram essenciais para a obtenção de resultados mais próximos da realidade, mesmo que eles não sejam ainda tão bons quanto os que temos com algoritmos de análise semântica (ver seção 4.3).

Devemos observar entretanto que, como o número de operadores e operandos aumenta com o tamanho do código, as proporções entre eles ficam mais distorcidas e as medidas de complexidade acrescentam menos informação aos métodos anteriores para programas mais complexos, o que indica que a eficiência é maior em classes de programação básica.

É notória a necessidade de utilização de ferramentas como o *MOSS* e o *JPlag* nesse caso, pois a análise do código-fonte deixa claramente de ser suficiente para detectar o plágio.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
45%	32%	56%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
0%	12,5%	41,75%

Tabela 9: Resultados do Experimento VII

4.2.8 Experimento VIII – Caso de alta complexidade

Muitos dos estudantes que copiam códigos em classes de programação tem bons conhecimentos. É nesses casos, em que são feitas alterações de toda natureza e cada técnica de plágio discutida anteriormente é utilizada, que os métodos de análise sintática se mostram mais falhos.

São utilizados aqui dois códigos-fonte que implementam um programa capaz de calcular a soma dos números primos menores ou iguais a uma dada entrada N e a soma dos quadrados do número de divisores dos naturais de 1 até N (Figuras 30 e 31).

Foram alterados nomes e posições de variáveis e funções, estruturas de controle de fluxo, laços, escopos, operadores e a ordem dos blocos independentes do código. Além disso, foram adicionadas ou eliminadas variáveis inúteis, espaços e linhas.

Assim como no experimento anterior, os resultados dos dois primeiros métodos estão longe de ser satisfatórios, o que, mais uma vez, já era esperado, dada a complexidade das alterações sintáticas realizadas.

As métricas aqui não tem o mesmo poder dos casos anteriores, já que a relação entre operadores e operandos foi completamente modificada, mas ainda sim apresenta resultados relevantes. Mesmo os algoritmos de análise semântica e outras ferramentas modernas ainda se mostram ineficientes aqui, o que é uma grande motivação para o desenvolvimento de novas tecnologias capazes de lidar bem com esse tipo de plágio.

É importante deixar claro que estudantes capazes de fazer tais seriam provavelmente capazes de elaborar sua própria solução, tendo em vista que, apesar de não terem conseguido ou querido fazer isso, são capazes de entender e alterar com tal nível de profundidade um programa já existente. Esse pensamento não se aplica, obviamente, no contexto da detecção de plágio em *softwares*, pois a quebra de patentes é um assunto bem mais sensível.

Sherlock (N = 2)	Inserção da Normalização (N = 2)	Inserção de métricas (N = 2)
10%	23%	36,5%
Sherlock (N = 3)	Inserção da Normalização (N = 3)	Inserção de métricas (N = 3)
0%	0%	25%

Tabela 10: Resultados do Experimento VIII

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int contdiv(int n){

int i, cont = 0;

for (i = 1; i < n+1; i++){

if(n%i == 0) cont++;

}
return cont;
}

int main(void){

int n, m;
scanf("%d", &m);
n = m;

int sum1 = 0, sum2 = 0, i;

while(n > 0){

if(contdiv(n) > 2 || contdiv(n) ==1) sum2 = sum2 +
contdiv(n)*contdiv(n);

else{
    sum1 = sum1 + n;
    sum2 = sum2 + contdiv(n)*contdiv(n);

}

n--;

}

printf("\nSoma dos primos menores que %d: %d\n", m, sum1);
printf("Soma dos quadrados dos divisores dos numeros ate %d:
%d\n", m, sum2);

return 0;

}

```

Figura 30: Experimento VIII, Código 1

```

#include <math.h>
#include <stdio.h>

long int teto;
int numdivide(int a, int b);
int numdivisores(long int n);

int main(void){

int loop;
long int somaprimos, somaquaddiv;
somaprimos = 0;
somaquaddiv = 0;
int pot = 2, 1;
scanf("%ld", &teto);
for(loop = 1; loop < teto+1; loop = loop + 1){
if(loop == 1) somaquaddiv++;
else if(numdivisores(loop) == 2){
somaquaddiv = somaquaddiv +
numdivisores(loop)*numdivisores(loop);
somaprimos = somaprimos+loop;
}
else somaquaddiv = somaquaddiv +
pow(numdivisores(loop),pot);
}

printf("%d\n", somaprimos);
printf("%d\n", somaquaddiv);
return 0;
}

int numdivide(int a, int b){
if(a%b==0) return 1;
else return 0;
}

int numdivisores (long int n){

int conta = 0;
long int j;
for (j = n; j > 0; j--){
if(numdivide(n,j) == 1) conta++;
}
return conta;
}

```

Figura 31: Experimento VIII, Código 2

4.3 Resumo dos resultados

Na Tabela 11 é apresentado o resumo dos resultados obtidos para cada experimento. Além dos métodos estudados no trabalho, foram adicionados os graus de similaridade de cada par utilizado quando submetidos aos métodos SIM (puramente sintático) e JPlag (semântico). Esses últimos dados foram conseguidos através da plataforma online desenvolvida no projeto de mestrado de Danilo Leal Maciel, aluno do Programa de Pós-Graduação em Engenharia de Teleinformática.

O objetivo é ter uma síntese de fácil visualização para a elaboração da conclusão do trabalho.

Utilizamos aqui algumas legendas:

- M1: Algoritmo de Sherlock, com N = 3
- M2: Inserção de normalizações, com N = 3
- M3: Inserção de métricas, com N = 3
- M4: Algoritmo de Sherlock, com N = 2
- M5: Inserção de normalizações, com N = 2
- M6: Inserção de métricas, com N = 2

	M1	M2	M3	M4	M5	M6	SIM	JPlag
I	50%	100%	95%	66%	100%	100%	100%	100%
II	12%	62,5%	81,25%	40%	50%	75%	100%	100%
III	66%	65%	57,5%	100%	100%	75%	40%	34,7%
IV	6%	60%	70%	21%	61,5%	70,75%	48%	50%
V	28%	60%	60%	57%	83%	71,5%	0%	17,2%
VI	77%	54%	77%	75%	77%	88,5%	59%	100%
VII	0%	12,5%	41,75%	45%	32%	51%	0%	19,2%
VIII	0%	0%	25%	10%	23%	36,5%	0%	18.7%

Tabela 11: Resumo dos Resultados

5 CONCLUSÃO

Foi possível observar claramente na seção de resultados que as normalizações melhoraram consideravelmente o Algoritmo de Sherlock, principalmente quando as alterações feitas nos códigos-fonte são unicamente textuais (Experimentos I, II, III, IV e V).

Isso se deve ao fato de, ao eliminarmos componentes que poderiam facilmente ser alterados sem mudar a estrutura do código, como nomes de variáveis, comentários, espaços, linhas e escopos, dispensarmos informação inútil, melhorando a análise com métodos sintáticos.

Apesar da clara evolução entre o primeiro método e o segundo, as métricas de complexidade se mostraram extremamente importantes em casos em que não havia alterações consideráveis na estrutura semântica e o conteúdo textual foi bastante alterado, principalmente nos Experimentos de I a VI. Isso condiz com a teoria explicada no capítulo 2 e comprova a tese central do projeto, que é a do alto grau de utilidade dessas técnicas (nos casos estudados, os resultados foram razoáveis mesmo nos experimentos VII e VIII).

Resumindo, quando não existem grandes alterações no número de operadores e operandos do programa, o terceiro método é o mais eficiente, tendendo a ter resultados bem superiores aos dois primeiros.

Uma outra conclusão interessante observada foi que o sexto experimento apresenta um ponto forte do Algoritmo de Sherlock, que é a baixa sensibilidade à alteração na ordem dos blocos independentes do código. O método *SIM*, por exemplo, que também é puramente sintático, apresenta resultados inferiores nesse quesito. O *JPlag*, puramente semântico, não é vulnerável a esse tipo de modificação.

Mesmo nos Experimentos VII e VIII, onde houve fortes alterações na estrutura dos códigos, os dois últimos métodos estudados aqui se mostraram melhores que o *SIM* e o *JPlag*. No caso do último, a situação se reverte quando o tamanho dos códigos aumenta, pois a estrutura semântica tende a se tornar cada vez mais relevante e o aumento das alterações sintáticas prejudicam o Algoritmo de Sherlock.

Observou-se também que mudanças em nomes de variáveis podem causar problemas em algoritmos de análise sintática, mas não no *JPlag*. Apesar disso, a análise léxica ainda é uma ferramenta mais eficiente quando são utilizados vários tipos de modificação textual (como no Experimento IV).

Com os bons resultados obtidos ao longo dos últimos anos, a detecção de plágio em classes de programação, principalmente nas mais básicas, tem se tornado um desafio menor. A implantação de métricas de complexidade agrega valor às ferramentas utilizadas hoje em dia e pode colaborar para diminuir irregularidades no meio acadêmico.

5.1 Trabalhos Futuros

São listadas abaixo quatro ideias que mostraram ter potencial para melhorar ferramentas existentes ou mesmo motivar a criação de novas técnicas de detecção de plágio, principalmente no contexto da Educação, foco do projeto:

- A análise da curva de resultados quando variamos os parâmetros de entrada do Algoritmo Sherlock fornece muitas informações, principalmente ao estudarmos a variável N. Pelo que foi observado, uma pequena variação nas respostas do método para valores consecutivos de N é um indicador de plágio mais importante do que uma grande variação.
- Para um grande número de códigos de entrada, a criação de estatísticas com as métricas de complexidade pode agregar bastante valor ao trabalho. Isso ocorre porque altos valores de variância indicam menor presença, de uma forma geral, de códigos plagiados. Além disso, uma vez calculada a média de cada uma das medidas para o conjunto de entrada, códigos-fonte com parâmetros distantes desse valor tem menos chance de apresentar irregularidades.
- A utilização do método de Montecarlo para obter os valores ideais para o parâmetro de precisão da função de similaridade utilizada nas métricas pode render bons resultados
- O estudo de melhorias na estimativa dos operandos é outra possibilidade de melhoria para as métricas

Além dessas abordagens, é possível também trabalhar mais com algoritmos de análise semântica e com as novas técnicas que tem surgido na área de detecção de plágio em algoritmos, onde são estudados os tempos de execução de blocos independentes do código, formando um vetor característico que carrega bastante informação.

Outra área de estudo importante é a de análise léxica em linguagem natural. A detecção de plágio em textos é um assunto importantíssimo no meio acadêmico e, apesar de já ser amplamente discutido, ainda tem sido citado em resultados recentes.

6 REFERÊNCIAS BIBLIOGRÁFICAS

BLUNCK, J. **Halstead Source-Code**. Disponível em: <https://github.com/jonasblunck/ccm>. Acesso em: 11 de Novembro de 2013.

CLOUGH, P. **Plagiarism in natural and programming languages: an overview of current tools and technologies**. Sheffield: Universidade de Sheffield, Departamento de Ciências da Computação. 2000.

DJURIC, Z.; GASEVIC, D. **A Source Code Similarity System for Plagiarism Detection**. Banjaluka: Universidade de Banjaluka, Faculdade de Engenharia Elétrica. 2012.

DONALDSON, J.; LANCASTER, A.; SPOSAT, P. **A plagiarism detection system**. St. Louis: 12th SIGCSE Technical Symposium on Computer Science Education. 1981.

FAIDHI, J. A. W.; ROBINSON, S. K. **An empirical approach for detecting program similarity and plagiarism within a university programming environment**. Exbridge: Universidade de Brunei, Departamento de Ciências da Computação. 1987.

FOTEL, C.; LANGER, L. **Bachelor's Project: A Plagiarism Detection Tool**. Athens: Universidade de Ohio, Departamento de Computação. 2004.

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.

GITCHELL, D.; TRAN, N. **SIM: A utility for detecting similarity in computer programs**. Wichita: Universidade Estadual de Wichita, Departamento de Ciências da Computação. 1997.

HAGE, J.; RADEMAKER, P.; VUGT, N. V. **A comparison of plagiarism detection tools**. Utrecht: Utrecht Uni-versity. 2010.

HALL, L. O.; BOWYER, K. W.; **Experience Using "MOSS" to Detect Cheating On Programming Assignments**. Tampa: Universidade do Sul da Florida, Departamento de Engenharia e Ciências da Computação. 2010.

HALSTEAD, M. H.; **Natural laws controlling algorithm structure**. New York: ACM SIGPLAN Notices. 1972.

HALSTEAD, M. H. **Elements of software science (Operating and programming systems series)**. New York: Elsevier Science Inc. 1977.

HECKEL, P. **A technique for isolating differences between files**. New York: Communications of the ACM. 1978.

JONES, E. L.; **Plagiarism monitoring and detection – Towards an open discussion**. Bransson: 7th Annual CCSC Central Plains Conference. 2007.

JOY, M.; LUCK, M. **Plagiarism in Programming Assignments**. Washington DC: IEEE Transactions on Education. 1999.

JPLAG. **JPlag – Detecting Software Plagiarism**. Disponível em: <https://www.ipd.uni-karlsruhe.de/jplag/>. Acesso em: 12 de Novembro de 2013.

KLEIMAN, A. B.; **Dissertação de Mestrado: Análise e comparação qualitativa de sistemas de detecção deplágio em tarefas de programação**. Campinas: Universidade Estadual de Campinas. 2007.

KLEIMAN, A. B.; KOWALTOWSKI, T. **Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs**. Campinas: Universidade Estadual de Campinas, Instituto de Computação. 2009.

LANCASTER, T.; CULWIN, F. **Towards an error free plagiarism detection process**. New York: 6th annual conference on Innovation and technology in computer science education. 2001.

LIU, C.; CHEN, C.; HAN, J.; YU, P. **GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis**. Philadelphia: 2th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2006.

MACIEL, D. L.; SOARES, J. M.; FRANÇA, A. B. **Análise de similaridade de códigos-fonte como estratégia para o acompanhamento de atividades de laboratório de programação**. Porto Alegre: CINTED-UFRGS. 2012.

MAHMOUD, H. A. **Detection Disguised Plagiarism**. Waterloo: Universidade de Waterloo. 2009.

MCCABE, T. **A complexity measure**. New York: IEEE Transactions on Software Engineering, VOL. SE-2, NO.4. 1976.

MOSS. **MOSS (Measure Of Software Similarity) plagiarism detection system**. Berkeley: University of California. Disponível em "<http://theory.stanford.edu/~aiken/moss/>". Acesso em: 16 de Novembro de 2013.

MOTA, A. M; GOYA, D. H. **Técnicas para análise de similaridade de códigos de software em litígios de propriedade intelectual**. São Paulo: Universidade de São Paulo, Departamento de Engenharia Elétrica. 2010.

NICKERSON. **Limits of the visual**. Disponível em: <http://www.nickerson.to/visprog/ch7/ch7.htm>. Acesso em: 20 de Novembro de 2012.

PARKER, A.; HAMBLIN, J. **Computer algorithms for plagiarism detection**. Washington DC: IEEE Transactions on Education. 1989.

PIKE, R. **The Sherlock Plagiarism Detector**. Disponível em: <http://sydney.edu.au/engineering/it/~scilect/sherlock/>. Acesso em: 05 de Novembro de 2013.

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

RAMAMURTHY, B.; SETTEMBRE, S. **Using MOSS: Plagiarism Detection Software**. Buffalo: Universidade de Buffalo. 2008.

SCHLEIMER, S.; WILJERSON, D. S.; AIKEN, A. **Winnowing: local algorithms for document fingerprinting**. New York: ACM SIGMOD International Conference on Management of Data. 2003.

SHIBATCHU. **Implementation of Halstead**. Disponível em: <http://www.slideshare.net/vamshibatchu/implementation-of-halstead>. Acesso em: 28 de Novembro de 2013.

WHALE, G. **Identification of Program Similarity in Large Populations**. The Computer Journal, Vol. 33, Number 2. 1990.

WISE, M.J. **Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing**. ACM SIGSCE Bulletin. 1992.

WISE, M.J. **YAP3: Improved Detection of Similarities in Computer Programs and Other Texts**. ACM SIGCSE Bulletin. 1996.

WISE, M. J. **String similarity via greedy string tiling and running Karp-Rabin matching**. Sidney: Universidade de Sydney, Departamento de Ciência da Computação. 1993.

APÊNDICE A – ALGORITMO DE SHERLOCK

Segue abaixo a versão original do algoritmo de Sherlock em C, com comentários em inglês (PIKE EWHITE, 2004):

```
* This program takes filenames given on the command line,
* and reads those files into memory, then compares them
* all pairwise to find those which are most similar.
*
* It uses a digital signature generation scheme to
randomly
* discard information, thus allowing a better match.
* Essentially it hashes up N adjacent 'words' of input,
* and semi-randomly throws away many of the hashed values
* so that it become hard to hide the plagiarised text.
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *      Progame = "sherlock";
int         Ntoken = 3;
int         Zerobits = 4;
unsigned long zeromask;
int         ntoken = 0;
char **     token;
FILE *      Outfile;
int         Thresh = 20;

/* characters to ignore at start and end of each word */
char *      Ignore = " \t\n";

/* characters to treat as word-separators or words on their
own */
char *      Punct_full = ",.<>/?:'\\"~[]{}\\|!@#$$%^&*()-
+_=\";
char *      Punct = "\";

typedef struct Sig Sig;
struct Sig
{
    int          nval;
    unsigned long *val;
};

void  init_token_array(void);
Sig * signature(FILE *);
```

```

int    compare(Sig *, Sig *);

void usage(void)
{
    fprintf(stderr, "%s: find similar files\n", Prognam);

    fprintf(stderr, "usage: %s", Prognam);
    fprintf(stderr, " [options] file1 file2 ...\n");

    fprintf(stderr, "options:");
    fprintf(stderr, " [-t threshold%%]");
    fprintf(stderr, " [-z zerobits]");
    fprintf(stderr, " [-n chainlength]");
    fprintf(stderr, " [-o outfile]");
    fprintf(stderr, "\n");

    fprintf(stderr, "defaults:");
    fprintf(stderr, " threshold=20%%");
    fprintf(stderr, " zerobits=3");
    fprintf(stderr, " chainlength=4");
    fprintf(stderr, " outfile=the screen");
    fprintf(stderr, "\n");
    exit(2);
}

int main(int argc, char *argv[])
{
    FILE *f;
    int i, j, nfiles, start, percent;
    char *s, *outname;
    Sig **sig;

    Outfile = stdout;
    outname = NULL;

    /* handle options */
    for (start=1; start < argc; start++) {
        if (argv[start][0] != '-')
            break;
        switch (argv[start][1]) {
        case 't':
            s = argv[++start];
            if (s == NULL)
                usage();
            Thresh = atoi(s);
            if (Thresh < 0 || Thresh > 100)
                usage();
            break;
        case 'z':

```

```

        s = argv[++start];
        if (s == NULL)
            usage();
        Zerobits = atoi(s);
        if (Zerobits < 0 || Zerobits > 31)
            usage();
        break;
    case 'n':
        s = argv[++start];
        if (s == NULL)
            usage();
        Ntoken = atoi(s);
        if (Ntoken <= 0)
            usage();
        break;
    case 'o':
        s = argv[++start];
        if (s == NULL)
            usage();
        outname = s;
        break;
    default:
        usage();
}

}

nfiles = argc - start;
if (nfiles < 2)
    usage();

/* initialise */
if (outname != NULL)
    Outfile = fopen(outname, "w");
init_token_array();
zeromask = (1<<Zerobits)-1;
sig = malloc(nfiles * sizeof(Sig *));

/* generate signatures for each file */
for (i=0; i < nfiles; i++) {
    /* fprintf(stderr, "%s: Reading %s\n", Prognome,
argv[i+start]); */
    f = fopen(argv[i+start], "r");
    if (f == NULL) {
        fprintf(stderr, "%s: can't open %s:",
                Prognome, argv[i+start]);
        perror(NULL);
        continue;
    }
    sig[i] = signature(f);
}

```

```

        fclose(f);
    }

    /* compare each signature pair-wise */
    for (i=0; i < nfiles; i++)
        for (j=i+1; j < nfiles; j++) {
            percent = compare(sig[i], sig[j]);
            if (percent >= Thresh)
                fprintf(Outfile, "%s and %s:
%d%%\n",
                                argv[i+start], argv[j+start],
percent);
        }

    return 0;
}

/* read_word: read a 'word' from the input, ignoring
leading characters
which are inside the 'ignore' string, and stopping
if one of
the 'ignore' or 'punct' characters is found.
Uses memory allocation to avoid buffer overflow
problems.
*/

char * read_word(FILE *f, int *length, char *ignore,
char *punct)
{
    long max;
    char *word;
    long pos;
    char *c;
    int ch, is_ignore, is_punct;

    /* check for EOF first */
    if (feof(f)) {
        length = 0;
        return NULL;
    }

    /* allocate a buffer to hold the string */
    pos = 0;
    max = 128;
    word = malloc(sizeof(char) * max);
    c = & word[pos];

    /* initialise some defaults */
    if (ignore == NULL)

```



```

        ignore = "";
    if (punct == NULL)
        punct = "";

    /* read characters into the buffer, resizing it if
necessary */
    while ((ch = getc(f)) != EOF) {
        is_ignore = (strchr(ignore, ch) != NULL);
        if (pos == 0) {
            if (is_ignore)
                /* ignorable char found at start,
skip it */
                continue;
        }
        if (is_ignore)
            /* ignorable char found after start, stop
*/
            break;
        is_punct = (strchr(punct, ch) != NULL);
        if (is_punct && (pos > 0)) {
            ungetc(ch, f);
            break;
        }
        *c = ch;
        c++;
        pos++;
        if (is_punct)
            break;
        if (pos == max) {
            /* realloc buffer twice the size */
            max += max;
            word = realloc(word, max);
            c = & word[pos];
        }
    }

    /* set length and check for EOF condition */
    *length = pos;
    if (pos == 0) {
        free(word);
        return NULL;
    }

    /* terminate the string and shrink to smallest
required space */
    *c = '\\0';
    word = realloc(word, pos+1);
    return word;
}

```

```

/* ulcmp:  compare *p1 and *p2 */
int ulcmp(const void *p1, const void *p2)
{
    unsigned long v1, v2;

    v1 = *(unsigned long *) p1;
    v2 = *(unsigned long *) p2;
    if (v1 < v2)
        return -1;
    else if (v1 == v2)
        return 0;
    else
        return 1;
}

/* hash:  hash an array of char* into an unsigned long hash
code */
unsigned long hash(char *tok[])
{
    unsigned long h;
    unsigned char *s;
    int i;

    h = 0;
    for (i=0; i < Ntoken; i++)
        for (s=(unsigned char*)tok[i]; *s; s++)
            h = h*31 + *s;
    return h;
}

void init_token_array(void)
{
    int i;

    /* create global array of char* and initialise all to
NULL */
    token = malloc(Ntoken * sizeof(char*));
    for (i=0; i < Ntoken; i++)
        token[i] = NULL;
}

Sig * signature(FILE *f)
{
    int nv, na;
    unsigned long *v, h;
    char *str;
    int i, ntoken;
    Sig *sig;

```

```

        /* start loading hash values, after we have Ntoken of
them */
        v = NULL;
        na = 0;
        nv = 0;
        ntoken = 0;
        while ((str = read_word(f, &i, Ignore, Punct)) !=
NULL)
        {
            /* step words down by one */
            free(token[0]);
            for (i=0; i < Ntoken-1; i++)
                token[i] = token[i+1];
            /* add new word into array */
            token[Ntoken-1] = str;

            /* if we don't yet have enough words in the
array continue */
            ntoken++;
            if (ntoken < Ntoken)
                continue;

            /* hash the array of words */
            h = hash(token);
            if ((h & zeromask) != 0)
                continue;

            /* discard zeros from end of hash value */
            h = h >> Zerobits;

            /* add value into the signature array, resizing
if needed */
            if (nv == na) {
                na += 100;
                v = realloc(v, na*sizeof(unsigned long));
            }
            v[nv++] = h;
        }

        /* sort the array of hash values for speed */
        qsort(v, nv, sizeof(v[0]), ulcmp);

        /* allocate and return the Sig structure for this file
*/
        sig = malloc(sizeof(Sig));
        sig->nval = nv;
        sig->val = v;
        return sig;

```

```

}

int compare(Sig *s0, Sig *s1)
{
    int i0, i1, nboth, nsimilar;
    unsigned long v;

    i0 = 0;
    i1 = 0;
    nboth = 0;
    while (i0 < s0->nval && i1 < s1->nval) {
        if (s0->val[i0] == s1->val[i1]) {
            v = s0->val[i0];
            while (i0 < s0->nval && v == s0->val[i0])
            {
                i0++;
                nboth++;
            }
            while (i1 < s1->nval && v == s1->val[i1])
            {
                i1++;
                nboth++;
            }
            continue;
        }
        if (s0->val[i0] < s1->val[i1])
            i0++;
        else
            i1++;
    }

    if (s0->nval + s1->nval == 0)
        return 0;    /* ignore if both are empty files */

    if (s0->nval + s1->nval == nboth)
        return 100; /* perfect match if all hash codes
match */

    nsimilar = nboth / 2;
    return 100 * nsimilar / (s0->nval + s1->nval -
nsimilar);
}

/*
 * Let f1 == filesize(file1) == A+B
 * and f2 == filesize(file2) == A+C
 * where A is the similar section and B or C are dissimilar
 *
 * Similarity = 100 * A / (f1 + f2 - A)

```

```

*           = 100 * A / (A+B + A+C - A)
*           = 100 * A / (A+B+C)
*
*   Thus if A==B==C==n the similarity will be 33% (one
third)
*   This is desirable since we are finding the ratio of
similarities
*   as a fraction of (similarities+disimilarities).
*
*   The other way of doing things would be to find the ratio
of
*   the sum of similarities as a fraction of total file
size:
*   Similarity = 100 * (A+A) / (A+B + A+C)
*   This produces higher percentages and more false matches.
*/

```

APÊNDICE B – CÓDIGOS NORMALIZADOS

É apresentado aqui um exemplo de par de códigos normalizados utilizados nos experimentos.

Os quatro procedimentos de normalização mencionados na seção 2.2.2.2 são postos em prática e os arquivos que os contem foram baixados na plataforma *online* disponibilizada pela orientador após a inserção das versões originais.

Códigos do Experimento VIII:

Código 1:

```
( ) { , =; (=; <+; ++ ) { ( % ==) ++; } ; } () { , ; (, &); =; =, =, ; (>) { ((>||() ==) =  
+ ()*()); { = + ; = + ()*(); } --; } ( , , ); ( , , ); ; }
```

Código 2:

```
; ( , , ); ( , , ); () { ; , ; =; =; =, ; (, &); (=; <+; = + ) { ( ==) ++; (() ==) { = + ()*();  
=+; } = + ((), ); } ( , , ); ( , , ); ; } ( , , ) { ( % ==) ; ; } ( , , ) { =; ; (=; > ; --) { ((, ) ==)  
++; } ; }
```