Universidade Federal de Mato Grosso do Sul
Faculdade de Computação
Ciência da Computação

Uma ferramenta para detecção de plágio

Aluno: Yuri Karan Benevides Tomas

Orientadora: Edna Ayako Hoshino

2014

1 Introdução

Plagiar é o ato de assumir a autoria de uma obra intelectual pertencente a outra pessoa. Ideias possuem um valor cada vez maior na sociedade conforme o setor quaternário, setor econômico relacionado a informação, ganha importância. Por este motivo, há um interesse crescente na detecção de plágio em textos, músicas, vídeos e até mesmo em códigos-fonte.

Foi proposta neste trabalho a ferramenta Rorschach para detecção de plágio em textos que utiliza licença GNU GLP baseada no algoritmo RKR-GST. A documentação da ferramenta foi elaborada por meio de ferramenta Doxygen que, através de marcações no código, gera arquivos de forma similar ao Javadoc. Além disto, o código da ferramenta foi projetado para facilitar sua manutenção evolutiva na implementação de detecção de plágio em códigos-fonte.

Este texto está dividido em mais três seções. Na Seção 2 são apresentados os principais algoritmos e os conceitos necessários para a compreensão do RKR-GST. A Seção 3 discute sobre o projeto desenvolvido neste trabalho. Por fim, a Seção 4 apresenta uma conclusão, as contribuições do trabalho e os desafios enfrentados.

2 Metodologia

2.1 Conceitos de Linguagens Formais

A seguir serão apresentados conceitos de linguagens formais que são importantes para a compreensão deste trabalho.

2.1.1 Alfabeto

Um alfabeto é um conjunto finito e não vazio de elementos, que por sua vez são chamados de *letras*. Geralmente utilizamos a letra grega Σ (sigma maiúsculo) para representar um alfabeto arbitrário. Alguns exemplos de alfabetos:

- $\Sigma = \{0, 1\}$ o alfabeto binário;
- $\Sigma = \{0, 1, ..., 9\}$, o alfabeto numérico;
- $\Sigma = \{a, b, ..., z\}$, o alfabeto das letras minúsculas; e
- O conjunto de caracteres que compõem o código ASCII.

2.1.2 Palavra

Uma *palavra* w, sobre um alfabeto Σ, é uma sequência de *letras* de Σ. A i-ésima letra de w é denotada por w_i. O comprimento de w, representado por $|x|$ é a quantidade de *letras* que a compõe.

2.1.3 Subcadeia

Uma *subcadeia* de w é uma *palavra* x cujas *letras* pertencem a w e estão em x na mesma sequência em w. Por exemplo, a *palavra* "tarde"é subcadeia da *palavra* "Boa tarde". Isto é, x é *subcadeia* de y se $\exists$ z e w tal que $zxw = y$.

2.2 Função *Hash*

Função hash é o nome dado a qualquer função que pode ser utilizada para mapear qualquer dado sem tamanho fixo para um dado de tamanho fixo. O valor retornado pela função de *hash* é chamado valor de *hash*. Ela pode ser utilizada na criação de uma estrutura de dados compacta – e consequentemente eficiente e econômica em relação ao consumo de memória – evitando *overflow*. Funções *hash* também podem ser utilizadas em criptografia de dados.

Por exemplo, podemos criar uma função *hash* para mapeamento de números inteiros para o intervalo de inteiros $[0, 99]$ através da função $f(x) = x \ mod \ 100$. Assim o valor de *hash* para o número 1039 seria 39.

2.3 Janela

Uma *janela* é uma sequência de tamanho fixo de *letras* em uma *palavra*. Esta sequência pode se deslocar para a direita ou para a esquerda dentro da palavra. Quando ocorre um deslocamento à direita, o termo mais à esquerda da *janela* anterior ao deslocamento passa a não pertencer à *janela*. Além disto, o termo mais à direita da nova *janela* passa a ser o elemento seguinte ao elemento mais à direita da *janela* anterior.

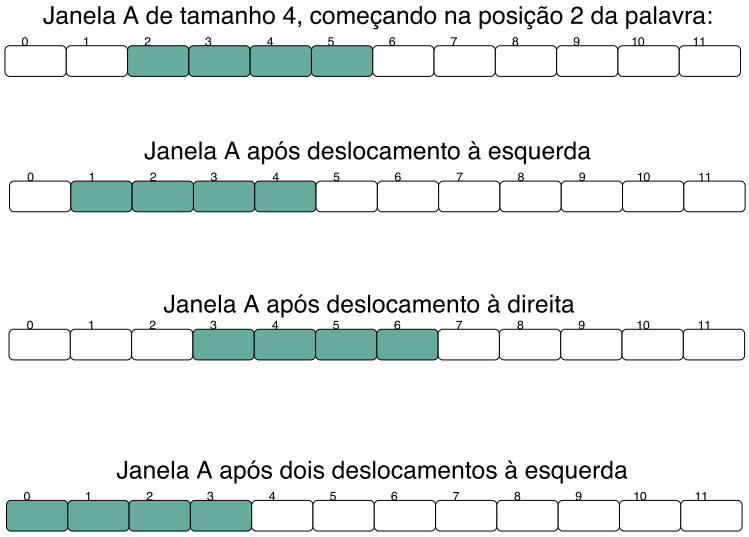

Figura 1: Exemplo de janela.

2.4 *Rolling Hash*

O custo computacional para o cálculo do valor de *hash* a cada deslocamento da *janela* pode ser muito alto dependendo das operações matemáticas envolvidas na função de *hash*. O *Rolling Hash*(OpenCourseWare [11]) é um algoritmo utilizado para encontrar o valor de *hash* em *janelas*, proposto a fim de diminuir o custo desse processamento. Para isto, ele aproveita o valor de *hash* da posição atual da *janela* para o cálculo do valor após um deslocamento à direita.

Após um deslocamento à direita, todos os termos contidos na *janela* serão os mesmos exceto pela inclusão de uma nova *letra* à direita e a exclusão de outra à esquerda. Para obter vantagem desta característica é feita uma certa analogia da *palavra* a um número. O valor de *hash* será o próprio número gerado.

Cada *palavra* é associada um número em uma base numérica especial definida pelo tamanho do alfabeto e cada *letra* do *alfabeto* é associado a um algarismo do número. Assim, sendo w, $hash(\mathrm{w}) = \sum_{i=0}^{|w|-1} w_i * (base)^{|w|-i}$.

Por exemplo, se $\Sigma = \{a, b\}$, como $|\Sigma| = 2$, então a base é 2. Podemos associar $a = 1$ e $b = 2$. Assim a *palavra* "aa"possui valor de hash 3, pois $1 * 2^1 + 1 * 2^0$. De forma análoga: ab = 4, ba = 5, bb = 6 é igual a 3.

Para a construção do valor de *hash* da *janela* inicial o número é gerado integralmente, não havendo ganho de desempenho por causa da *Rolling Hash*. Após isto, o valor de *hash* das próximas iterações da *janela* podem ser calculados em tempo linear utilizando o valor de *hash* das iterações anteriores.

Considere que cada algarismo do número encontrado pela função de *hash* (considerando a base numérica adotada) refere-se a uma letra da *janela*. Para encontrar o valor de *hash* após um deslocamento à direita, o algarismo mais à esquerda deixa de pertencer à número referente ao valor de *hash* da janela. Além disto, o algarismo referente à *letra* mais à direita da nova *janela* anterior passa a fazer parte do valor de hash da janela atual.

Matematicamente as seguintes operações são realizadas:
Para a *janela* inicial A

1. Para cada *letra* a ser adicionada:

   Incrementando a posição de cada termo do número:

   $hashValue = hashValue * base$

   Adicionar o novo termo na posição menos significativa do número

   $hashValue = hashValue + newLetter$

Para a *janela* A após um deslocamento à direita

1. Remover o termo mais significativo do número (oldLetter):

   $hashValue = hashValue - base^{oldIndex-1} \times oldLetter$

2. Incrementando a posição de cada termo do número:

   $hashValue = hashValue * base$

3. Adicionar o novo termo (newLetter) na posição menos significativa do número

   $hashValue = hashValue + newLetter$

Cada uma dessas três operações é feita em O(1), fazendo com que o custo computacional para a criação das *hash* posteriores à primeira seja linear. Para uma grande sequência de dados temos que o custo para a criação da primeira *hash* se torna irrelevante.

Para encontrar o valor inicial de *hash* para uma sequência de caracteres, por exemplo, o transformamos em um número. Se fizermos isto para caracteres com acentuação a base deste número deverá ser 256 já que o ASCII extendido, o alfabeto em questão, possui 256 valores possíveis. Assim, o valor correspondente à "som"seria:

Valores correspondentes na tabela ASCII extendida:

índice da posição	3	2	1
código ASCII extendido	115	111	109
caracter	s	o	m

Valor de *hash* para a *palavra* "som"=
$115 \times 256^2 + 111 \times 256^1 + 109 \times 256^0 = 7565165$

Caso a *letra* à direita da *janela* atual fosse "a"e deslocássemos a *janela* nesta direção, as sequintes operações ocorreriam na próxima iteração:

1. Remover o termo mais significativo do número:

   $28525 = 7565165 - 115 \times 256^2$

2. Incrementando a posição de cada termo do número:

   $7302400 = 28525 \times 256$

3. Adicionar o novo termo na posição menos significativa do número

   $7302497 = 7302400 + 97$

Como podemos perceber, o número resultante destas operações pode acabar estourando a quantidade de bits máxima determinada para o tipo usado para armazená-lo, principalmente quando i tamanho da janela é muito grande. Por este motivo, após cada operação, é feita uma operação de aritmética modular utilizando um número primo maior do que o tamanho do alfabeto utilizado. Assim, as operações matemáticas se tornariam:

1. Encontrar o valor do termo mais significativo:
   $$\text{aux} = (oldLetter \times (base^{oldIndex-1} mod primeNumber) mod primeNumber$$
2. Remover o termo mais significativo do número:
   $$hashValue = hashValue - aux$$
3. Operação de resto para evitar overflow
   $$hashValue = hashValue mod primeNumber$$
4. Incrementando a posição de cada termo do número:
   $$hashValue = hashValue * base$$
5. Adicionar o novo termo na posição menos significativa do número
   $$hashValue = hashValue + newLetter$$
6. Evitando overflow que a adição do termo poderia causar
   $$hashValue = hashValue mod primeNumber$$

2.5 Busca de padrão em palavra

A busca de padrão em *palavra*, também nomeado como *string-matching* ou *string-searching* é o nome dado a uma classe de algoritmos. Neles procura-se por um padrão em um texto. Tanto o padrão quanto o texto são *palavras* do mesmo alfabeto Σ. Neste artigo denominaremos o texto como *text*, o padrão como *pattern* e *subText* uma *subcadeia* de text. Assim, a busca de padrão procura um *subText* de tamanho $|pattern|$ que seja igual a *pattern*.

2.6 Algoritmo trivial para string-matching

A solução trivial é dada pelo algoritmo abaixo :

Algorithm 1 Algorimo Trivial de String-Matching

```
1: int i, j
2: for (i = 0; i < textSize − patternSize; i++) do
3:     for (j = 0; j ¡ patternSize ; j++) do
4:         if (! (text[i + j] == pattern[j]) then
5:             break;
6:     if (j == patternSize) then
7:         return 1;
8: return 0 ;
```

O algoritmo compara cada *subcadeia* de *text* que cuja primeira *letra* éw_i, ao *pattern* buscado, *letra* a *letra*. Caso a comparação indique igualdade entre as

duas, encontramos o padrão buscado. Caso *subtext* difira de *pattern*, incrementamos o i e repetimos o processo.

Realizamos este laço até, no pior caso, $|text| - |pattern|$ vezes, sendo o valor inicial de i igual ao índice do primero caracter do texto. Após este ponto não haverá caracteres suficientes para criar um *subtext* de tamanho $|pattern|$. Cada iteração do laço, no pior caso, realiza $|pattern|$ comparações já que a diferença pode estar apenas na última *letra*.

Dessa forma temos que a complexidade do algoritmo trivial é de $O((|text| - |pattern|) \times |pattern|) = O(|text| \times |pattern|)$.

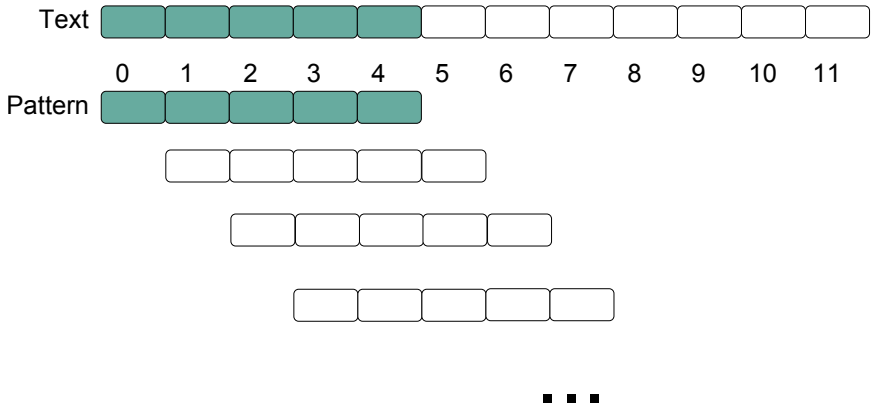

Figura 2: Buscando um padrão contendo cinco *letras* em um texto. *Subcadeias* do texto de cinco *letras* são comparadas ao padrão.

2.6.1 Algoritmo Karp-Rabin

Embora simples e custoso, o algoritmo trivial foi a base para a criação do algoritmo Karp-Rabin (Karp and Rabin [8]). A diferença está na comparação feita entre *pattern* e as *subtext* de tamanho $|pattern|$.

No algoritmo Karp-Rabin, ao invés de compararmos *pattern* e uma *subcadeia* de *text* de tamanho $|pattern|$, *letra* a *letra*, damos a cada um deles um valor calculado através de uma *hash*. Através disto é possível realizar comparações em tempo linear já que estamos apenas comparando números. Se, de alguma forma, possuirmos os valores de *hash*, o tempo passará a ser $O(|text| - |pattern|)$.

Uma vez que *text* e *pattern* não são fixos, seus valores de *hash* não são previamente conhecidos. Ainda assim, podemos suprimir o custo de cálculo de uma *hash*, tornando-o linear. O fato que possibilita isto é a similaridade entre as *subcadeias* de iterações vizinhas. A *subcadeia* da iteração *i+1* é igual ao da iteração i com a retirada do termo mais à esquerda e o acréscimo do termo mais à direita.

Para encontrar os valores de *hash* de cada iteração utilizamos o algoritmo *Rolling Hash*, o que nos possibilita encontrar

Dessa forma utilizamos o conceito de *Rolling Hash* para nos aproveitar desta característica.

TO DO:

- Mostrar a complexidade do algoritmo Karp-Rabin

TO DO:

- Atualizar operações de mod utilizadas para a última versão

2.7 Definições necessárias para o algoritmo RKR-GST

Para um melhor entendimento do Running-Karp-Rabin Greedy-String-Tilling , ou simplesmente RKR-GST, alguma definições devem ser vistas.

2.7.1 Casamento

O *casamento* ocorre quando uma *subcadeia* de tamanho l é igual a outra de mesmo tamanho. Podemos utilizar a notação casamento(p, t, l) para representar um *casamento* entre *subcadeia*de tamanho l de *pattern* e *text* que iniciam nas *letras* p e t, respectivamente.

2.7.2 Tile

O *tile* é um *casamento* único e permanente entre uma *subcadeia* de *text* e outra de *pattern*. Após a formação de um *tile*, as *letras* de ambas as *subcadeia* são *marcadas*.

2.7.3 Letra marcada

Uma *letra* considerada *marcada* se ela é parte de um *tile*, tornando-a indisponível para novos *casamentos*.

2.7.4 Casamento maximal

Um *casamento maximal* é o *casamento* de uma *subcadeia* de *pattern* e outra de *text* que possui o maior tamanho possível, ou seja, até a próxima diferença entre elas ou o final de alguma das duas *palavras* ou até encontrarmos uma *letra marcada*. Por ser um *casamento maximal* ser um *casamento*, podemos representá-lo por *casamento(p, t, l)*.

2.7.5 Tamanho mínimo de casamento

Tamanho mínimo de casamento é tamanho mínimo permitido para um *casamento maximal*. Qualquer *casamento maximal* encontrado com valor inferior ao *tamanho mínimo de casamento* definido será ignorado. A decisão do valor adotado possui uma grande influência na detecção de plágio e depende do contexto.Quando comparamos códigos-fonte, por exemplo, não é interessante encontrar palavras-chave da linguagem em questão durante a busca por plágio. Na linguagem C, por exemplo, pode-se escolher um *tamanho mínimo de casamento* maior do que |"if"|, o que possibilita ignorar vários trechos que contém a palavra-chave "if", mas cujas condições são diferentes.

2.7.6 O algoritmo Greedy-String-Tilling

A busca das *tiles* que maximizam o número de *letras* cobertas sem sobreposição é um problema de tempo polinomial. Ainda assim, um *tile* maior é um grande

indício de similaridade e, por este motivo, é possível utilizar um algoritmo guloso, como o Greedy-String-Tilling (Karp and Rabin [8]), também chamado de GST, para encontrar um *tile* de comprimento máximo.

Fazendo *P[i]* a *letra* de índice i na palavra *palavra* P, temos o seguinte pseudocódigo do algoritmo GST :

Algorithm 2 Algorimo GST

```
1:  lengthTiled = 0
2:  while maxMatch! = minimumMatchLength do
3:      for Cada letra não marcada de índice p de pattern do
4:          for Cada letra não marcada de índice t de text do
5:              j = 0
6:              while pattern[p + j] = text[t + j] E text[t +j] não é marcada
    do
7:                  j := j + 1
8:              if j = maxMatch then
9:                  Adicionar o match(p, t j) a lista de matches
10:             else if j > maxMatch then
11:                 Começar nova lista de matches com match(p, t, j) e maxMatch
    = j
12:     for cada match(p, t, maxMatch) na lista do
13:         if as duas subcadeia do match não contenham letras marcadas then
14:             Criar tile
15:             for j de 0 a maxMatch-1 do
16:                 Marcar text[t + j]
17:                 Marcar pattern[p + j]
18:             lengthTiled := lengthTiled + maxMatch
        return lengthTiled
```

Neste algoritmo tenta-se criar os maiores *tiles* possíveis antes dos menores. Caso um *casamento* maior que os já encontrados na iteração atual seja encontrado, os menores previamente encontrados são descartados.

Podemos dividir o algoritmo do GST em duas partes. A primeira, que chamaremos de *scanPattern*, é representada pelo laço da 4 à 11 e busca os maiores *casamentos* possíveis. A segunda, que chamaremos de *markArrays*, é representada pelo laço da linha 12 à 18 e *marca* as *letras* dos *tiles* formados.

Como provado em Wise [14], o pior caso para o GST possui complexidade $O(n^3)$. Algumas técnicas podem ser utilizadas para melhorar o desempenho dele. Entre elas estão:

- A estrutura de dados utilizada para armazenar as *letras*, tanto de *pattern* quanto de *text*, deverá conter um ponteiro para a próxima *letra* não *marcada*. Dessa forma, a cada novo *tile* criado, menor será o espaço de busca.
- Se a distância do ponteiro atual até o próximo *tile* for menor do que o valor atribuído ao *tamanho mínimo de casamento*, ir para a primeira posição após o *tile*, já que um *casamento* menor que *tamanho mínimo de casamento* deve ser ignorado.

- É utilizado o algoritmo Karp-Rabin para a criação de valores de *hash* para todas as *subcadeias* de *pattern* e *text* do tamanho da variável *maxMatch*, que indica o tamanho do *casamento* buscado na atual iteração.

2.8 Algoritmo Running-Karp-Rabin Greedy String Tiling

As otimizações do GST serviram de base para a criação do RKR-GST. Um algoritmo chamado Running-Karp-Rabin foi criado e utilizado, em que invés de criarmos um valor de hash para *pattern*, como no Karp-Rabin, criamos uma para cada *subcadeia*, formada por *letras* não *marcadas*, de tamanho *s*.

Outra mudança significativa está na escolha do tamanho *s* utilizado a cada iteração. Ao invés de ser decrementado até alcançar *tamanho mínimo de casamento*, seu valor na próxima iteração é decidido através de uma estimativa. Esta estimativa será comentada posteriormente.

De forma simplificada podemos descrever o RKR-GST pelo algoritmo abaixo:

1. Criamos valores de *hash* para todas as *subcadeia* de *text* e de *pattern*, contendo apenas letras não *marcadas*, de tamanho *s*. Para reduzir o custo de comparação uma tabela de hash é criada para armazenar *subcadeia* de *text* utilizando seus valores de *hash*.

2. Comparar os valores de hash de todas as *subcadeia* de tamanho *s* de ambas as *palavras*. Caso sejam iguais há uma grande chance das *subcadeia* serem iguais, ou seja, são candidatas a formarem um *casamento*. Posteriormente comparamos as duas *subcadeia*, *letra* a *letra* para verificar se o *casamento* ocorre. Assim como no GST, comprovada a existência de um *casamento*, tenta-se transformá-lo em um *casamento maximal*.

3. Repetir os passos 1 e 2 até que o tamanho buscado *s* seja igual a *tamanho mínimo de casamento*.

OBS: No algoritmo original os *casamentos maximais* de tamanhos diferentes são guardados em listas diferentes. Na implementação foi utilizada apenas uma lista ordenada pelo tamanho.

Sendo s o tamanho das *subadeias* buscadas na iteração atual:

Algorithm 3 markArrays

```
1:  for each F doila de matches
2:      while F doila atual não é vazia
3:          Remova um match(p, t, l) da fila
4:          if Match não possui letras marcadas then
5:              for (i = 0 ; i ¡ l; i++) do
6:                  Marcar pattern_{p+i}
7:                  Marcar text_{t+i}
8:              lengthOfTokensTiled = lengthOfTokensTiled + l
9:          else if l − l_{ocluso} >= s then
10:             Guardar trecho não marcado na lista adequada ao seu tamanho.
```

Algorithm 4 scanPattern(s)

```
1: for each letra, não marcada, t de text do
2:     if Distância para o próximo tile de text for menor do que s then
3:         Ir para primeira letra após o próximo tile
4:     else
5:         Criar valor de hash para a subcadeia de text que começa na posição
   t e possui tamanho s.
6: for each letra, não marcada, p de pattern do
7:     if Distância para o próximo tile de pattern for menor do que s then
8:         Ir para primeira letra após o próximo tile
9:     else
10:        Criar valor de hash, patternHash, utilizando o KR para a subcadeia
   que começa na posição p de pattern e possui s letras.
11:        for each Entrada da tabela de hash, textHash, igual a patternHash
   do
12:            if Todas as letras das subcadeia representadas por patternHash e
   textHash forem iguais, ou seja, se o match for confirmado  then
13:                k = s
14:                while pattern[p + k] = text[t + k] E pattern[p+k] não for
   marcado E text[t + k] não for marcado do
15:                    k++
16:                if k > 2 * s then
17:                    return k(recomeçar o scanPattern com s = k)
18:                else
19:                    registar novo maximalMatch guardado-o em uma fila.
20: return |maximalMatch|
```

Algorithm 5 Running-Karp-Rabin Greedy String Tilling

```
1: searchLength s = initialSearchLength
2: while true do
3:     maxLength = scanPattern(s)
4:     if s > 2 * minimumMatchLength then
5:         s = s / 2
6:     elsemarkArrays(s)
7:         if s > 2 * minimumMatchLength then
8:             s / 2
9:         else if s > minimumMatchLength then
10:            s = minimumMatchLength
11:        else
12:            break
```

3 Abordagem

Diversos artigos já foram publicados comparando o desempenho e os algoritmos utilizados pelas principais ferramentas de detecção de plágio existentes. Em Hage, Rademaker, and van Vugt [7] o autor compara o desempenho de cada uma delas. Apesar de haver uma variação de resultados entre os testes realizados, aqueles com melhores resultados, de maneira geral, segundo o artigo foram o Jplag Prechelt, Malpohl, and Philippsen [12] e o Marble Hage [6], seguidos pelo MOSS Aiken et al. [2].

Em Đurić and Gašević [13] são descritos superficialmente os algoritmos utilizados por diversas ferramentas e seus recursos são comparados. Também é proposto um novo algoritmo baseado nas ferramentas JPlag e MOSS. Em Martins, Fonte, Henriques, and da Cruz [10], o desempenho de diversas ferramentas são comparadas, além de uma breve descrição técnica de cada uma delas.

Através destas pesquisas é possível perceber que muitas ferramentas utilizam o algoritmo RKR-GST como base para sua implementação. Entre elas podemos citar o Jplag Prechelt, Malpohl, and Philippsen [12], o CPD Copeland [3], o Plaggie Ahtiainen, Surakka, and Rahikainen [1], o Marble Hage [6] e o YAP3 Wise [15].

Neste trabalho propôs-se a criação do programa Rorschach, um detector de plágio em textos, base para um futuro detector de plágio em códigos-fonte, que utiliza o algoritmo RKR-GST para encontrar a similaridade.

Decidiu-se utilizar a versão 3 da licença GNU General Public License [9] para este projeto. Esta escolha foi feita para permitir que qualquer pessoa possa estudar ou modificar o código.

A ferramenta Plaggie (Ahtiainen, Surakka, and Rahikainen [1]) possui licença GNU GPL e detecta plágio em códigos-fonte escritos em Java 1.5, mas sua única documentação são os comentários existentes no código. Rorschach foi documentado através da ferramenta Doxygen [5], de licença GNU GPL, além das informações presentes neste artigo.

3.1 Decisões de projeto

3.1.1 Classes criadas

Rorschach foi escrito de forma a permitir a fácil adaptação para a detecção de plágio em código-fonte. Esta facilidade se deve ao uso de classes paramétricas cujo parâmetro do template é o tipo ou classe que representa a *letra* utilizada. Dessa forma, para a detecção de plágio em textos este parâmetro do template será um caractere, enquanto para códigos-fonte será token gerado após a análise léxica da linguagem de programação em questão.

- Reader: Responsável pela leitura dos arquivos que contém *text* e *pattern* para o seu posterior tratamento. Para a detecção de plágio em textos, após construção de um objeto desta classe, deve executar o método filesToBox, que faz a leitura de *text* e *pattern* e agrupa as informações destes arquivos que serão úteis durante toda a execução do programa em um objeto da classe Box. Para a detecção de plágio em códigos-fonte será necessário implementar um método que faça a *tokenização* do conteúdo dos arquivos antes de grupar esses dados em um objeto da classe Box.

- Box: Classe que agrupa informações sobre os dados de entrada previamente tratados pelo Reader. Esta classe possui um vetor de *letras* de *text*, um vetor de *letras* de *pattern* e duas variáveis inteiras que representam os tamanhos de cada vetor.
- RandomNumber: Classe que gera número aleatórios. Utilizada para a criação de números primos aleatórios.
- RandomPrime: Classe que gera números primos aleatórios utilizando os números aleatórios criados por um objeto da classe RandomNumber.
- (*)Letter: Classe parametrizada que representa uma *letra*. Possui como atributos seu conteúdo, uma variável binária que representa se a letra está *marcada* ou não e ponteiros para as próxima e prévia letras *marcadas* dentro de uma estrutura externa que a *letra* em questão se encaixa.
- Match: Representação de um *casamento*, possuindo as três informações que a caracteriza: tamanho, posição inicial em *pattern* e posição inicial em *text*. Além disto, possui ponteiros para *casamentos* anteriores e posteriores ao objeto em questão, o que é útil quando este faz parte de uma estrutura externa que o engloba.
- (*)Substring: Classe que representa uma *subcadeia*, pedaço da *palavra* que será comparada a outro. Possui ponteiros para *subcadeias* anteriores e posteriores, útil quando o objeto desta classe for parte de uma estrutura de dados maior.
- Tile: Classe que representa uma *subcadeia* casada de forma definitiva. Possui apenas informações sobre suas posições inicial e final, além de ponteiros para *tiles* anteriores e posteriores a este quando fizer parte de uma estrutura que o engloba.
- ListOfMatches: Em Wise [14], foi criada uma lista diferente para *casamentos* de tamanhos diferentes, dando preferência a listas com *casamentos* maiores durante a criação de *tiles*. Neste trabalho optou-se por utilizar apenas uma lista ordenada em ordem decrescente.
- (*)ListOfSubstrings: Lista de *subcadeias*.
- ListOfTiles: Uma lista contendo todos os *tiles* criados.
- (*)RollingHash: Classe que gerencia as operações do algoritmo RollingHash. Para todo objeto desta classe os valores da base e o número primo utilizados devem ser os mesmos.
- (*)HashTable: Cria uma tabela de hash cujo número de entradas possíveis é igual ao número primo utilizado pela RollingHash.
- (*)RKRGST: Classe que implementa o algoritmo RKR-GST, possuindo três métodos: o markArrays, o scanPattern e o execute. Este último é o que executa o RKR-GST através de chamadas dos dois métodos anteriores.

(*) = Seu parâmetro do template refere-se ao tipo ou classe que representa a *letra*.

3.1.2 Outras decisões de projeto

Para quantificar a taxa de similaridade à partir dos *tiles* encontrados através do método execute da classe RKR-GST utilizamos a fórmula proposta em Đurić and Gašević [13]:

similaridade (a, b) = $(2 * numberOfTokensTiled)/(length(a) + length(b))$

O acréscimo de uma operação modular entre cada operação realizada, como foi explicado na seção 2.4, serve para evitar o estouro da variável, que armazena o valor da hash, gerado pelo conjunto de operações feitas para remover ou adicionar uma letra na RollingHash. Mesmo que não tenha sido alertado em *wise1993string*, durante a implementação deste trabalho foi descoberto que uma só operação pode facilmente provocar o estouro de uma variável por causa da grandeza dos operandos. Para evitar com que isto ocorra o valor da base foi alterado de 256, tamanho do alfabeto utilizado, para 2.

Após um deslocamento à direita da *janela*, a *letra* que deixa de pertencer a janela é a de algarismo mais significativo numericamente. Para a remoção do valor desta *letra* no valor de hash utilizamos, entre outras operações matemáticas, a operação $base^{|s|-1}$(mais detalhes na Seção 2.4). Mesmo que o valor da base tenha passado de 256 para 2, o valor desta expressão cresce exponencialmente e quando s é muito grande este valor pode ultrapassar o tamanho máximo possível para uma variável. Para contornar este problema limitamos o tamanho máximo das *subcadeias*através de uma constante. Esta alteração não possui muita influência na taxa de similaridade, já que se analisarmos a fórmula utilizada não importa quantos *tiles* sejam criados, mas o somatório dos seus tamanhos.

Se cada variável tivesse tamanho infinito, e dessa forma não fosse necessário a utilização de operações modulares, cada palavra possuiria um valor de hash único. O uso das operações modulares e de uma base muito menor do que a desejável criaram valores de hash iguais para algumas palavras diferentes, o que deixou código um pouco menos eficiente já que a linha 12 do algoritmo scanPattern será executada mais vezes sem que isto signifique a existência de mais *casamentos*.

Os caracteres dos arquivos referentes a *text* e *pattern* foram lidos como caracteres sem sinal já que alguns deles possuem valor maior do que 127 fazendo com que necessitem utilizar o bit mais significativo do byte para a sua representação, tornando-os negativos.

3.2 testes realizados

Em Đurić and Gašević [13] são descritas modificações utilizadas na criação de versões plagiadas de um código-fonte original. São elas:

- modificações léxicas:
  - Modificação do formato do código fonte.
  - Adição, remoção ou modificação de comentários.
  - Modificação da linguagem do código.
  - Modificação do formato de saida do programa.
  - Renomeação identificadores.
  - Quebra ou junção de declarações de variáveis.

  - Adicição, remoção ou modificação de modificadores.
  - Modificação de valores de constantes.

- modificações estruturais
  - Mudança de ordem de variáveis na sentença.
  - Mudança de ordem de sentenças dentro de um bloco de código.
  - Reordenação de blocos de código.
  - Adicição de sentenças ou variáveis redundantes.
  - Modificação das estruturas de controle.
  - Mudança tipos de dados e modificar estruturas de dados.
  - Refatoração de métodos.
  - Redundancia em geral.
  - Variáveis temporárias.

Muitas das alterações citadas poderiam ser detectadas somente através de uma prévia tokenização da entrada, sendo necessário para isto a implementação de análise léxica para o português, inglês ou outra linguagem escolhida. Isto tornaria a detecção mais precisa, mas o intuito deste trabalho é base para um futuro detector de plágio em códigos-fonte, não textos simples. Mesmo com essas restrições, duas modificações foram testadas:

- Acréscimo de redundância
- Reordenação de trecho

Para a realização dos testes foi utilizado o conteúdo publicado no site Dot [4]. As notícias publicadas neste site são sumários de notícias de outros sites que são enviados por leitores. Esses sumários são avaliadas pelos editores antes de serem publicados ou descartados.

Foram utilizados seis sumários para os testes. Os seguintes arquivos foram criados para cada sumário:

- resume.txt: O sumário publicado no site Slash Dot.
- original.txt : A notícia original.
- reorderingResume: O sumário com trechos fora de ordem.
- redundancyResume.txt: O sumário com acréscimo de trechos quaisquer.
- redundancyAndReorderingResume.txt: O sumário incluso fora de ordem em um texto qualquer

Os arquivos gerados foram agrupados de acordo com o sumário que o originou. Para cada grupo o respectivo arquivo resume.txt foi comparado aos outros arquivos gerados utilizando o valor 7 como *tamanho mínimo de casamento* e o valor 10 como o tamanho de *casamento* inicialmente buscado. Os seguintes resultados foram encontrados:

Grupo de teste	Palavras comparadas	Tempo em segundos	Similaridade
1	resume X reorderingResume	0,025798	94,269%
1	resume X redundancyResume	0,030290	60,668%
1	resume X redundancyAndReorderingResume	0,031225	59,931%
1	resume X original	0,033345	30,515%
2	resume X reorderingResume	0,012337	95,019%
2	resume X redundancyResume	0,026853	40,444%
2	resume X redundancyAndReorderingResume	0,054153	40,214%
2	resume X original	0,031114	28,535%
3	resume X reorderingResume	0,012465	94,399%
3	resume X redundancyResume	0,027442	39,918%
3	resume X redundancyAndReorderingResume	0,052695	39,926%
3	resume X original	0,020296	24,453%
4	resume X reorderingResume	0,015246	92,121%
4	resume X redundancyResume	0,016464	57,062%
4	resume X redundancyAndReorderingResume	0,020069	57,042%
4	resume X original	0,035113	27,280%
5	resume X reorderingResume	0,015189	94,977%
5	resume X redundancyResume	0,044476	34,597%
5	resume X redundancyAndReorderingResume	0,054084	34,541%
5	resume X original	0,015126	44,620%
6	resume X reorderingResume	0,008312	93,258%
6	resume X redundancyResume	0,035859	26,277%
6	resume X redundancyAndReorderingResume	0,056416	26,257%
6	resume X original	0,043144	15,233%

Tabela 1: Resultado dos testes

Na Tabela 1 pode-se perceber que a mudança de ordem (reorderingResume.txt e redundancyAndReorderingResume) em uma *palavra* não é um empecilho para a detecção do plágio, já que após a mudança de ordem em do sumário a pior similaridade encontrada foi de 93,258% e a mudança de ordem após a geração de redundância no resumo (redundancyResume.txt) teve uma influência ínfima na similaridade.

A geração de redundância teve uma grande influência na similaridade, mas vale salientar que a redundância adicionada ao sumário em todos os grupos de teste possuíam tamanho muito superior ao tamanho do sumário. Criar redundâncias deste porte em códigos-fonte de médio e grande requerem grande esforço, além de, pela quantidade de *letras* necessárias, se tornar perceptível facilmente.

4 Conclusões

Através deste trabalho foi possível criar o programa Rorschach. Ele possui licença GNU GPL e é melhor documentado do que o Plaggie, outro programa que utiliza a mesma licença, o que facilita o seu estudo e extensão.

Espera-se que Rorschach seja extendido para a detecção de plágio em códigos-fonte e utilizado durante a correção de trabalhos práticos na Faculdade de Computação da Universidade Federal de Mato Grosso do Sul.

O código-fonte, a sua documentação e os casos de teste utilizados foram disponibilizados no serviço Git Hub *gitHub*, que utiliza o sistema de controle de versão e de gerenciamento de código fonte Git. *git*. Rorschach pode ser acessado através do link https://github.com/iruynarak/rorschach.

Referências

[1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 141–142. ACM, 2006. Artigo sobre o Plaggie.

[2] Alex Aiken et al. Moss: A system for detecting software plagiarism. *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html*, 9, 2005.

[3] Tom Copeland. Detecting duplicate code with pmd's cpd. *On Java*, 2003. Artigo sobre o CPD.

[4] Slash Dot. Slash dot. URL http://slashdot.org/.

[5] Doxygen. Doxygen. URL http://www.stack.nl/~dimitri/doxygen/.

[6] Jurriaan Hage. Programmeerplagiaatdetectie met marble. *TINFON: Tijdschrift voor Informatica Onderwijs*, 16(1):4, 2007. Artigo sobre o Marble.

[7] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28, 2010. Não busca descrever as diferentes implementações das ferramentas existentes, mas fazer uma comparação dos resultados de cada uma delas. Os principais critérios usados foram usabilidade, recursos contidos e desempenho. Apesar de haver uma variação relativamente grande dos resultados dos subtestes realizados, aqueles com melhores resultados de maneira geral foram o Jplag e o Marble, seguidos de perto pelo MOSS.

[8] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2): 249–260, 1987. Artigo original do algoritmo Karp-Rabin.

[9] GNU General Public License. Gnu general public license. URL http://www.gnu.org/licenses/gpl.html.

[10] Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz. Plagiarism Detection: A Tool Survey and Comparison. In Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 143–158, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-68-2. doi: http://dx.doi.org/10.4230/OASIcs.SLATE.2014.143. URL http://drops.dagstuhl.de/opus/volltexte/2014/4566.

[11] MIT OpenCourseWare. 9. table doubling, karp-rabin, 2011. URL https://www.youtube.com/watch?v=BRO7mVIFt08. Aula do MIT em que Erik Demaine explica o funcionamento do Karp-Rabin e do Rolling Hash.

[12] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[13] Zoran Đurić and Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, page bxs018, 2012. Survey de 2012 contendo informações relevantes sobre ferramentas existentes, além de dar sua própria contribuição. O artigo começa falando um pouco sobre o plágio, a importância de sua detecção. Após citar alguns tipos de plágio, o autor cita trabalhos realizados na área, as técnicas que utlizaram e os respectivos resultados. Com base nisto o autor foca a pesquisa nas JPlag e MOSS por obterem ótimos resultados, serem referências a ferramentas criadas posteriormente e por serem implementados de formas distintas. Além disto algumas falhas do JPlag são apontadas. Através desta pesquisa inicial o autor propõem sua própria ferramenta levando em conta todos os pontos levantados. Para objetivar sua criação, alguns desafios são propostos. Em sua ferramenta proposta, a SCSDS, o autor utiliza tanto o RKR-GST, do JPlag, quanto o Winnowing, do MOSS. Para o cálculo de similaridade pesos são dados ao valor obtido por cada técnica durante a fase de medida de similaridade, sendo que como o RKR-GST é a principal delas seu peso é maior.

[14] Michael J Wise. String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119, 1993. Primeiro uso do RKR-GST.

[15] Michael J Wise. Yap3: Improved detection of similarities in computer program and other texts. In *ACM SIGCSE Bulletin*, volume 28, pages 130–134. ACM, 1996. Artigo sobre o YAP3.