Rorschach: Uma Ferramenta para Detecção de Plágio

Yuri Karan Benevides Tomas[1], **Edna Ayako Hoshino (orientadora)**[1]

[1] Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS

iruynarak@gmail.com, eah@facom.ufms.br

Abstract. *Considering the growing magnitude of ideas recent years, the plagiarism detection has become constant demand, in texts, songs, as well as source codes. This work proposes the creation of the tool Rorschachfor plagiarism detection in simple texts. She has GNU GPL license and was designed to allow extension for plagiarism detection in source codes. The tool was tested and results are presented in this paper.*

Resumo. *Considerando a crescente magnitude das ideias nos últimos anos, a detecção de plágio tem se tornado uma demanda constante, tanto em textos, músicas, bem como em códigos-fonte. Este trabalho propõe a criação da ferramenta Rorschachpara detecção de plágio em textos simples. Ela possui licença GNU GPL e foi planejada de forma a permitir extensão para a detecção de plágio em códigos-fonte. A ferramenta foi testada e os resultados são apresentados neste artigo.*

1. Introdução

Plagiar é o ato de assumir a autoria ou utilizar como fonte 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.

Diversos artigos já foram publicados comparando o desempenho e os algoritmos utilizados pelas principais ferramentas de detecção de plágio existentes. Em [Hage et al. 2010] 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 et al. 2002] e o Marble [Hage 2007], seguidos pelo MOSS [Aiken et al. 2005].

Em [Đurić and Gašević 2012] 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 et al. 2014], 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 et al. 2002], o CPD [Copeland 2003], o Plaggie [Ahtiainen et al. 2006], o Marble [Hage 2007] e o YAP3 [Wise 1996].

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.

A ferramenta Plaggie ([Ahtiainen et al. 2006]) 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 utiliza a versão 3 da licença GNU General Public License [GNU 2014] e foi documentado através da ferramenta [Doxygen 2014], de licença GNU GPL e pelas informações presentes neste artigo. Esta escolha foi feita para facilitar o estudo ou modificação do código.

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

Nesta seção, são descritos conceitos teóricos importantes para devida compreensão deste trabalho.

2.1. Conceitos de Linguagens Formais

A seguir serão apresentados alguns conceitos de linguagens formais, de acordo com [Hopcroft and Ullman 1979], 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*. 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, ou tamanho, de w, representado por $|w|$ 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 que 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 informação sem tamanho fixo para um 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 o 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. A figura 1 ilustra um exemplo de janela.

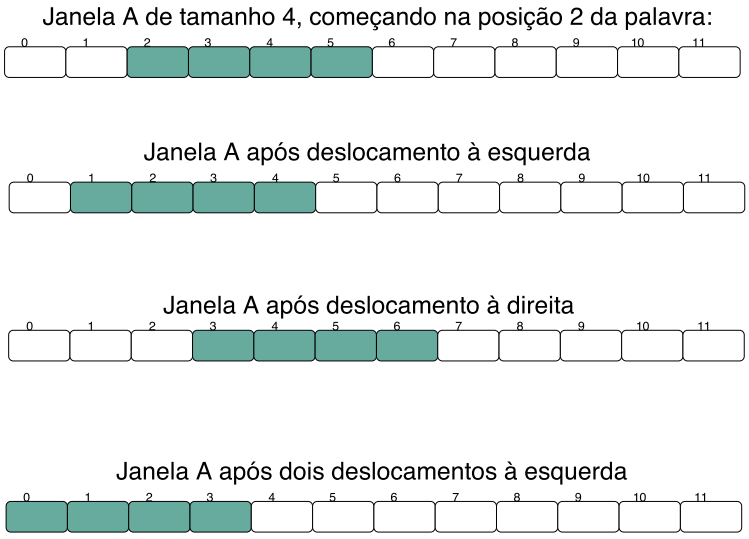

Figura 1. Exemplo de janela.

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*([MIT OpenCourseWare]) 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, todas as *letras* contidas na *janela* serão as mesmas 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* da *palavra* será o próprio número gerado pela analogia.

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

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 igual a 3, pois $1*2^1+1*2^0 = 3$. De forma análoga: ab = 4, ba = 5, bb = 6.

Para a construção do valor de *hash* da *janela* inicial o número é calculado integralmente, não havendo ganho de desempenho por causa da *Rolling Hash*. O valor de *hash* desta *palavra* após a realização de um deslocamento pode ser calculado em tempo linear utilizando o valor de *hash* antes deste deslocamento.

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 ao número referente ao valor de *hash* da janela. Além disto, o algarismo referente à *letra* mais à direita da nova *janela* passa a fazer parte do valor de hash da janela atual.

Após um deslocamento à direita, a *letra* mais à esquerda na janela deixa de pertencer a ela e a *letra* à direita da *janela*, passa a fazer parte dela. Assim, o número referente ao valor de *hash* após o deslocamento à direita será o igual ao atual, exceto pela remoção de um algarismo à esquerda e o acréscimo de outra à direita. Matematicamente as seguintes operações são realizadas para a *janela* inicial:

1. Zerar valor de hash:
   $hashValue = 0$
2. Para cada *letra* a ser adicionada:
   Incrementar a posição de cada termo do número:
   $hashValue = hashValue \times base$
   Adicionar o novo termo na posição menos significativa do número
   $hashValue = hashValue + newLetter$

Para a *janela* após um deslocamento à direita

1. Remover o termo mais significativo do número (oldLetter):
   $hashValue = hashValue - base^{oldIndex-1} \times oldLetter$
2. Incrementar a posição de cada termo do número:
   $hashValue = hashValue \times base$
3. Adicionar o novo termo (newLetter) na posição menos significativa do número
   $hashValue = hashValue + newLetter$

Cada uma dessas três operações feitas para encontrar o valor de *hash* após um deslocamento é feita em O(1), fazendo com que o custo computacional para a descoberta dos valores de *hash* seja linear.

Para encontrar o valor inicial de *hash* para uma sequência de caracteres, por exemplo, o transformamos em um número. Se fizermos isto para caracteres acentuados a base

deste número deverá ser 256 já que o ASCII extendido, o alfabeto em questão, possui 256 valores possíveis.

A Tabela 1 ilustra os valores das *letras* que compõem a *palavra*"so" de acordo com a tabela ASCII .

Índice da posição	3	2	1
Código ASCII extendido	115	111	109
Caractere	s	o	m

Tabela 1. Valores correspondentes na tabela ASCII extendida.

Dessa forma, o 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
   $7302497 = 7302400 + 97$

O número resultante destas operações pode acabar estourando a quantidade de bits máxima determinada para o tipo usado para armazená-lo, principalmente quando o 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:
   aux = $(oldLetter \times (base^{oldIndex-1} \bmod primeNumber) \bmod primeNumber$
2. Remover o termo mais significativo do número:
   $hashValue = hashValue - aux$
3. Operação de resto para evitar overflow
   $hashValue = hashValue \bmod primeNumber$
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
   $hashValue = hashValue \bmod primeNumber$

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*.

2.5.1. 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*, iniciada em uma posição 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 $O((|text| - |pattern|) \times |pattern|) = O(|text| \times |pattern|)$.

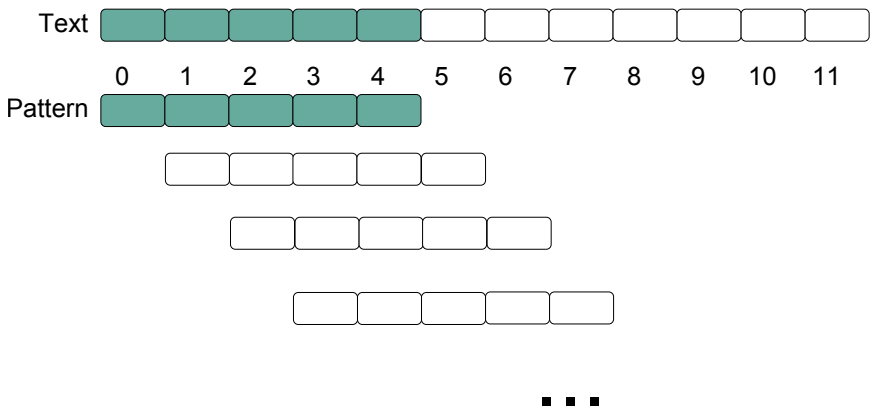

Figura 2. Buscando um padrão contendo cinco *letras* em um texto. *Subcadeias* do texto de cinco *letras* são comparadas ao padrão.

2.5.2. Algoritmo Karp-Rabin

Embora simples e custoso, o algoritmo trivial foi a base para a criação do algoritmo Karp-Rabin ([Karp and Rabin 1987]). A diferença está na comparação feita entre *pattern* e as *subcadeias* 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 uma *subcadeia* e a formada após um deslocamento como explicado na Seção 2.3 e na Seção 2.4.

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

2.6. Definições Necessárias para o Algoritmo *RKR-GST*

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

2.6.1. Casamento

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

2.6.2. *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 *subcadeias* são *marcadas* para evitar a sobreposição de *tiles*.

2.6.3. Letra Marcada

Uma *letra* considerada *marcada* é indisponível para novos *casamentos*, ou seja, *casamentos* não são permitidos se alguma das *subcadeias* possui *letra* marcada.

2.6.4. Casamento Maximal

Um *casamento maximal* é um *casamento* de uma *subcadeia* de *pattern* e outra de *text* que não permite aumento do seu tamanho, ou seja, não permite aumento do tamanho das suas *subcadeias*. O aumento do tamanho das *subcadeias casadas* pode causa diferença entre elas, inclusão de *letra marcada* em alguma delas ou encontro do fim de alguma das palavras.

2.6.5. Tamanho Mínimo de Casamento

Tamanho mínimo de casamento é o tamanho mínimo permitido para um *casamento maximal*. Qualquer *casamento maximal* encontrado com valor inferior ao *tamanho mínimo de casamento* definido é 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.

2.6.6. 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* grande é um indício maior de similaridade do que um pequeno, por representar um trecho maior similar entre as *palavras* comparadas. Por este motivo, é possível utilizar um algoritmo guloso, como o *Greedy-String-Tilling* ([Karp and Rabin 1987]), também chamado de GST, para encontrar *tiles*, dando preferência aos maiores.

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! = tamanhoMinimo 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 casamento(p, t j) à lista de casamentos
10:             else if j > maxMatch then
11:                 Começar nova lista de casamentos com casamento(p, t, j) e maxMatch
    = j
12:     for cada casamento(p, t, maxMatch) na lista de casamentos do
13:         if as duas subcadeias do casamento 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* antes dos menores. Caso um *ca-*

samento maior que os já encontrados na iteração atual seja encontrado, os outros 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 das linhas 12 à 18 e *marca* as *letras* dos *tiles* formados.

Como provado em [Wise 1993], 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;
- Utilizar 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.

2.7. *Algoritmo Running-Karp-Rabin Greedy String Tilling*

As otimizações do GST serviram de base para a criação do RKR-GST. O 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. Criar valores de *hash* para todas as *subcadeias* 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 todas as *subcadeias* de *text* utilizando seus valores de *hash* para o posicionamento na tabela;
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*.

Vale ressaltar que, 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, o algoritmo que marca os *tiles* é apresentado no Algoritmo 3:

Algorithm 3 markArrays

```
1:  for each Fila de casamentos do
2:      while Fila atual não é vazia do
3:          Remova um casamento(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 Tamanho do trecho não marcado do casamento >= s then
10:             Guardar trecho não marcado na lista adequada ao seu tamanho.
```

O Algoritmo 4 apresenta o pseudocódigo do *scanPattern*.

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 Karp-Rabin 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|
```

O Algoritmo 5 representa o algoritmo RKR-GST. Ao contrário do algoritmo GST ele tenta alterar o tamanho do *casamento* buscado em cada iteração através de uma esti-

mativa estática. Isto é feito através de comparações entre o tamanho buscado, o tamanho encontrado e o tamanho mínimo para o *casamento*.

Algorithm 5 Running-Karp-Rabin Greedy String Tilling

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

3. Abordagem

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 o token gerado após a análise léxica da linguagem de programação do código-fonte. Esta seção descreve as decisões do projeto e os resultados dos testes realizados para avaliar a ferramenta. A figura 3 ilustra as classes difinidas no projeto.

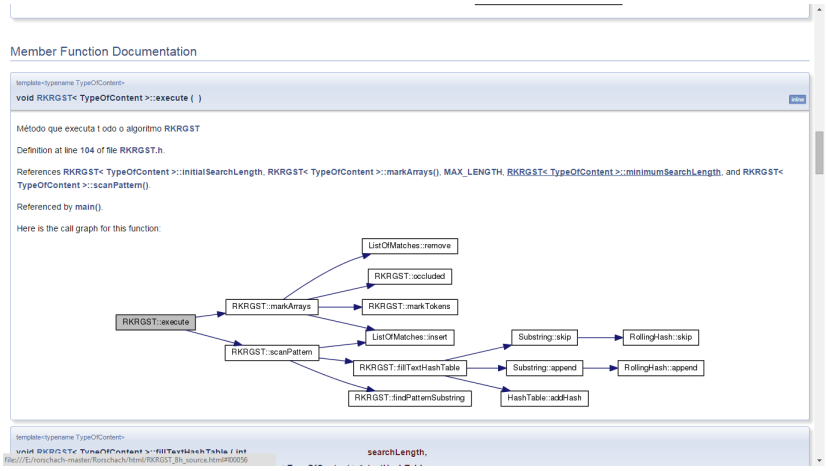

Figura 3. Exemplo de documentação gerada pelo Doxygen.

3.1. Decisões de Projeto

A seguir, são apresentadas as principais decisões de projeto tomadas para a implementação do Rorschach.

3.1.1. Classes Criadas

Uma descrição de cada classe é apresentada a seguir:

- 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, o método filesToBox deve ser executado. Este método 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* da *palavra* que será comparada a outra. 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 1993], 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.

O símbolo (*) indica as classes que utilizam templates referentes 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 2012]:

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

A similaridade quantifica o proporção da porção semelhante entre as duas *palavras* comparadas em relação ao tamanho total delas. Uma similaridade de 100% significa que a porção semelhante em cada uma das *palavras* é equivalemente a *palavra* inteira.

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, gerada pelo conjunto de operações feitas para remover ou adicionar uma letra na RollingHash. Mesmo que não tenha sido alertado em [Wise 1993], 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 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, podendo ultrapassar o tamanho máximo possível para uma variável. Para contornar o 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 2012] 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 saída do programa;
  - Renomeação de identificadores;
  - Quebra ou junção de declarações de variáveis;
  - Adição, remoção ou modificação de modificadores; e
  - 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;
  - Adição de sentenças ou variáveis redundantes;
  - Modificação das estruturas de controle;
  - Mudança de tipos de dados e modificar estruturas de dados;
  - Refatoração de métodos;
  - Redundância em geral; e
  - 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; e reordenação de trecho.

Para a realização dos testes foi utilizado o conteúdo publicado no site [Slash Dot]. As notícias publicadas neste site são sumários de notícias de outros sites que são enviados por leitores. Esses sumários são avaliados 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;
- `reordering.txt`: o sumário com trechos fora de ordem;
- `redundancy.txt`: o sumário com acréscimo de trechos quaisquer; e
- `redundancyAndReordering.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. Para cada comparação, o tempo gasto em milissegundos e a similaridade foram registrados. Abaixo os resultados encontrados:

Grupo	Comparação	Tempo	Similaridade
1	resume X reordering	25,79	94,2%
1	resume X redundancy	30,29	60,6%
1	resume X redundancyAndReordering	31,22	59,9%
1	resume X original	33,34	30,5%
2	resume X reordering	12,33	95,0%
2	resume X redundancy	26,85	40,4%
2	resume X redundancyAndReordering	54,15	40,2%
2	resume X original	31,11	28,5%
3	resume X reordering	12,46	94,3%
3	resume X redundancy	27,44	39,9%
3	resume X redundancyAndReordering	52,69	39,9%
3	resume X original	20,29	24,4%
4	resume X reordering	15,24	92,1%
4	resume X redundancy	16,46	57,0%
4	resume X redundancyAndReordering	20,06	57,0%
4	resume X original	35,11	27,2%
5	resume X reordering	15,18	94,9%
5	resume X redundancy	44,47	34,5%
5	resume X redundancyAndReordering	54,08	34,5%
5	resume X original	15,12	44,6%
6	resume X reordering	8,31	93,2%
6	resume X redundancy	35,85	26,2%
6	resume X redundancyAndReordering	56,41	26,2%
6	resume X original	43,14	15,2%

Tabela 2. Resultado dos testes

Na Tabela 1 pode-se perceber que a mudança de ordem (`reordering.txt` e `redundancyAndReordering.txt`) em uma *palavra* não é um empecilho para a detecção do plágio, já que, após a mudança de ordem em do sumário(`reordering.txt`), a pior similaridade encontrada foi de 93,2% e a mudança de ordem após a geração de redundância no resumo (`redundancyAndReordering.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 tamanho médio e grande requerem grande esforço, além de, pela quantidade de *letras* necessárias, se tornar perceptível facilmente.

4. Considerações Finais

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.

Como sugestão de trabalho futuro, está a extensão do Rorschach para a detecção de plágio em códigos-fonte e que seja utilizado durante a correção de trabalhos práticos

na Faculdade de Computação da Universidade Federal de Mato Grosso do Sul.

As alterações necessárias para que Rorschach seja utilizado para detecção de plágio em código-fonte consistem basicamente na a alteração da classe Reader para a leitura, remoção de comentários e espaços, tokenização do código-fonte e inclusão das *palavras* geradas através dos tokens criados em um objeto da classe Box.

Outra alteração importante é a sobrecarga de operadores da classe que representará o token para que possua os mesmos operadores utilizados neste trabalho em operações com *letra*. Entre as operações utilizadas estão comparação entre *letras* e o uso da operação de atribuição para atribuir a uma variável inteira o valor correspondente a uma *letra*.

neste programa pertencentes ao tipo *char*, o tipo paramétrico utilizado para a representação das *letras* neste trabalho.

O código-fonte, a sua documentação e os casos de teste utilizados foram disponibilizados[1] via GitHub [GitHub 2014].

Referências

[Ahtiainen et al. 2006] Ahtiainen, A., Surakka, S., and Rahikainen, M. (2006). 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.

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

[Copeland 2003] Copeland, T. (2003). Detecting duplicate code with pmd's cpd. *On Java*.

[Doxygen 2014] Doxygen (2014). Doxygen. Disponível em: <http://www.stack.nl/ dimitri/doxygen/>. Acesso em: 27 nov. 2014.

[GitHub 2014] GitHub (2014). Github. Disponível em: <https://github.com/>. Acesso em: 04 dez. 2014.

[GNU 2014] GNU (2014). GNU – General Public License. Disponível em: <http://www.gnu.org/licenses/gpl.html>. Acesso em: 29 nov. 2014.

[Hage 2007] Hage, J. (2007). Programmeerplagiaatdetectie met marble. *TINFON: Tijdschrift voor Informatica Onderwijs*, 16(1):4.

[Hage et al. 2010] Hage, J., Rademaker, P., and van Vugt, N. (2010). A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28.

[Hopcroft and Ullman 1979] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

[Karp and Rabin 1987] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260.

[Martins et al. 2014] Martins, V. T., Fonte, D., Henriques, P. R., and da Cruz, D. (2014). Plagiarism Detection: A Tool Survey and Comparison. In Pereira, M. J. V., Leal, J. P.,

[1] Endereço eletrônico: https://github.com/iruynarak/rorschach.

and Simões, A., editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 143–158, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[MIT OpenCourseWare] MIT OpenCourseWare. 9. table doubling, karp-rabin.

[Prechelt et al. 2002] Prechelt, L., Malpohl, G., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016.

[Slash Dot] Slash Dot. Slash Dot. Disponível em: <http://slashdot.org/>. Acesso em: 04 dez. 2014.

[Đurić and Gašević 2012] Đurić, Z. and Gašević, D. (2012). A source code similarity system for plagiarism detection. *The Computer Journal*, page bxs018.

[Wise 1993] Wise, M. J. (1993). String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119.

[Wise 1996] Wise, M. J. (1996). Yap3: Improved detection of similarities in computer program and other texts. In *ACM SIGCSE Bulletin*, volume 28, pages 130–134. ACM.