Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alan Bustos Kleiman e aprovada pela Banca Examinadora.

Campinas, 17 de Julho de 2007.

Prof. Tomasz Kowaltowski
Instituto de Computação, Unicamp
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Kleiman, Alan Bustos

K672a Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação / Alan Bustos Kleiman -- Campinas, [S.P. :s.n.], 2007.

Orientador : Tomasz Kowaltowski

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Análise sintática. 2. Informática na educação. 3. Ciência da computação. 4. Kolmogorov, Complexidade de. I. Kowaltowski, Tomasz. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Qualitative analysis and comparison of plagiarism-detection systems on programming coursework.

Palavras-chave em inglês (Keywords): 1. Parsing. 2. Computer science education. 3. Computer science. 4. Kolmogorov complexity.

Área de concentração: Sistemas de Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Tomasz Kowaltowski (IC-UNICAMP)
Prof. Dr. Célio Cardoso Guimarães (IC-UNICAMP)
Prof. Dr. Guilherme Pimentel Telles (ICMC-USP/SC)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)

Data da defesa: 22/06/2007

Programa de Pós-Graduação: Mestrado em Ciência da Computação

Instituto de Computação
Universidade Estadual de Campinas

Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação

Alan Bustos Kleiman[1]

Julho de 2007

Banca Examinadora:

- Prof. Tomasz Kowaltowski
  Instituto de Computação, Unicamp (Orientador)
- Prof. Célio Cardoso Guimarães
  Instuto de Computação, Unicamp
- Prof. Guilherme Pimentel Telles
  Instituto de Ciências Matemáticas e de Computação, USP/SC
- Prof. Ricardo de Oliveira Anido
  Instituto de Computação, Unicamp (Suplente)

[1]Suporte financeiro de: Bolsa da CAPES 2005–2007

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 22 de junho de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Guilherme Pimentel Telles
ICMC - USP / SC.

Prof. Dr. Celio Cardoso Guimarães
IC - UNICAMP.

Prof. Dr. Tomasz Kowaltowski
IC - UNICAMP.

Resumo

Plágio em submissões de alunos é um problema que vem aumentando ao longo do tempo e instituições de ensino têm trabalho considerável para eliminá-lo. Examinamos o problema do ponto de vista de submissões de alunos em disciplinas introdutórias de programação, fazendo um resumo de alguns sistemas e algoritmos existentes. Implementamos vários algoritmos descritos com a finalidade de efetuar uma comparação direta e qualitativa, com foco no pré-processamento de programas. Em particular, desenvolvemos um mecanismo para a normalização de programas através de uma análise sintática cuidadosa e reordenação da árvore de sintaxe abstrata de maneira a minimizar a quantidade de ruído criada por plagiadores ao tentar copiar e modificar programas de outros. Conseguimos resultados positivos utilizando esse método de pré-processamento, especialmente quando combinado com o algoritmo conhecido como *Running Karp Rabin Greedy String Tiling*. Esses resultados positivos reforçam nossa conclusão de que o pré-processamento pode ser até mais importante que o algoritmo em si, e apontam novas direções para pesquisas futuras.

Abstract

Encountering plagiarism in student coursework has become increasingly common, and significant effort has been undertaken to counter this problem. We focus on the plagiarism in student submissions in programming courses, in particular in introductory computer science courses, describing some of the existing systems and algorithms already dedicated to this problem. We implement many of the algorithms so that we could undertake a direct and qualitative comparison, with a special focus on pre-processing student programs. In particular, we develop a mechanism for normalizing programs through careful parsing and ordering of their abstract syntax trees so as to minimize the noise created by plagiarists attempting to copy and modify someone else's program. We achieve positive results utilizing this new pre-processing method, particularly with the Running Karp Rabin Greedy String Tiling algorithm. The positive results reinforce our conclusion that pre-processing may be more important than the algorithm itself and point to new directions for further research.

Agradecimentos

Agradeço a todos que, de alguma forma, compartilharam do desenvolvimento dessa pesquisa:

Principalmente ao meu orientador, Tomasz Kowaltowski, que me concedeu suporte e ajuda para o desenvolvimento desse estudo, e pelas valiosas sugestões e por seu envolvimento nessa pesquisa.

À minha família, especialmente meus pais, que me deram apoio durante o desenvolvimento desse trabalho.

E à CAPES — Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, que me concedeu bolsa para a realização do mestrado.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Existem muitas definições do que constitui plágio. Para nossos propósitos consideraremos apenas plágio envolvendo cópia integral ou parcial de trechos de código em tarefas de programação. Em particular, concentraremos atenção em casos em que o plágio é do trabalho de colegas. Considerar situações em que alunos podem obter código da internet também é plágio, mas é mais difícil de detectar, simplesmente pela dificuldade de se encontrar todas as possíveis fontes de código.

Não estamos considerando, em momento algum, plágio em tarefas que não são de programação (como por exemplo plágio em monografias), nem estamos considerando plágio de código de um âmbito mais geral (como plágio de código em projetos de código livre).

Existem diversas técnicas para efetuar plágio em tarefas de programação. A mais simples é quando um aluno copia e cola trechos do programa do colega. De fato, esse tipo de plágio é extremamente simples, e mesmo uma inspeção visual consegue facilmente revelá-lo, apesar de que inspeção visual pode ser muito trabalhosa. Uma variante dessa técnica envolve o processo de cópia e cola, mas com alteração do nome de variáveis, ou inserção de espaços em branco e outras reformatações. Isso dificulta a detecção, mas mesmo os sistemas mais simples conseguem ignorar essas mudanças. É claro que é possível tornar o processo de plágio mais e mais elaborado e, conseqüentemente, mais difícil de detectar. Por exemplo, em muitos casos é possível alterar a posição de blocos de código sem alterar o funcionamento do programa. Às vezes, é possível inserir código "nulo", que não faz nada.

De fato, dadas as possibilidades de alteração, é provavelmente impossível gerar um sistema que consiga nunca ser enganado por plágio. Com alterações suficientes, dois programas que eram iguais podem passar a não ser — no caso da inserção de código nulo, por exemplo. É necessário, entretanto, lembrar do contexto no qual se está tentando detectar plágio: em geral, se um aluno está copiando do outro é porque não tem conhecimento ou tempo suficiente para completar a tarefa por si próprio. Isso significa que esse mesmo

aluno provavelmente não fará alterações extremamente complexas, devido a essa mesma falta de conhecimento. Claro que existem exceções.

O problema do plágio vem tendo um impacto crescente nos últimos anos. Uma pesquisa publicada pelo Center of Academic Integrity[14] revelou que, em 2005, 40% dos alunos admitiram ter efetuado plágio, comparado aos 10% que admitiram o mesmo em 1999. Outra pesquisa conduzida por um professor da Universidade de Rutgers[16] revelou que 38% dos alunos participaram em plágio *on-line*. A Universidade de Stanford[1] e a Universidade da Califórnia em Berkeley, entre outras, também apresentaram um aumento no casos de plágio em anos recentes, apesar de essas universidades terem punições bastante severas[4].

Um grande problema para a detecção de plágio é a dificuldade de efetuá-la. Como descrito anteriormente, é trivial alterar um programa de modo que visualmente ele não se pareça mais com o programa original. Isso, adicionado a grande quantidade de submissões que um examinador deve verificar, torna a detecção manual de plágio impraticável. É essa a importância dos sistemas de detecção de plágio automáticos, em particular como um mecanismo de triagem, idealmente reduzindo o número de submissões que devem ser inspeccionados manualmente.

Apesar da dificuldade de se detectar o plágio manualmente, não é aconselhável tornar a detecção de plágio inteiramente automática. Sempre podem ocorrer casos de falsos positivos, em que um programa é acusado de apresentar plágio quando isso não é verdade. Falsos negativos, se ocorrerem, são claramente mais difíceis de se lidar. Por isso, é preferível que um sistema de detecção de plágio seja sensível demais, ao contrário de sensível de menos – uma vez que estamos supondo será feita uma inspeção manual nos resultados obtidos pelo sistema de detecção. O ideal seria um sistema que é robusto quanto a falsos positivos e falsos negativos, ao mesmo tempo.

1.1 Contagem de atributos e comparação de estruturas

Duas das políticas comuns para comparação de programas são: contagem de atributos e comparação de estruturas. Sistemas que utilizam contagem calculam o número de ocorrências de atributos em um dado programa. Por exemplo: contar o número de vezes que palavras-chave como **if** ou **for** ocorrem em um programa em C, ou mesmo contar quantas vezes, em um programa, ocorre um laço ou expressão condicional. Essa política oferece resultados bons para casos onde houve pouca tentativa de alteração. Além do problema de robustez, existe o problema de que é difícil determinar quais atributos devem ser contados para evitar falsos positivos, uma vez que a proporção de atributos em

uma dada linguagem é uniforme. Portanto, programas diferentes mas de comprimento semelhante podem apresentar contagens muito parecidas.

A outra estratégia, mais interessante no nosso caso, é de análise de estrutura. Em estratégias de análise de estrutura o objetivo é buscar trechos de um programa que, após processamento, se apresentam como muito parecidos com trechos de outro programa. Existem vários algoritmos baseados nessa política, entre eles *peneiragem*[1], *emparelhamento de ladrilhos ganancioso*[2] e mesmo o algoritmo de *maior subseqüência comum*[3], que estudaremos mais adiante. A vantagem da análise de estrutura é que ela permite encontrar trechos de plágio parcial [20] — casos onde foi copiada apenas parte do programa de um aluno, algo que ocorre bastante na prática.

1.2 Organização da dissertação

Essa dissertação está organizada da seguinte maneira: o capítulo 2 contém uma descrição dos vários algoritmos utilizados pelos sistemas existentes para a detecção de plágio. No capítulo 3 fazemos um resumo dos sistemas de detecção existentes. No capítulo 4 abordamos nossas idéias para melhorias sobre os algoritmos e sistemas descritos no capítulo 3 e, no capítulo 4, discutimos como foram implementadas algumas dessas idéias. Já no capítulo 5 fazemos uma comparação entre os resultados obtidos pelos sistemas implementados no capítulo 4 e os resultados de dois dos sistemas já existentes mais importantes. A partir disso apresentamos nossas conclusões no capítulo 6.

[1] *Winnowing*, em inglês.
[2] *Greedy String Tiling* ou GST, em inglês
[3] *Longest Common Subsequence*, em inglês

Capítulo 2

Algoritmos

Neste capítulo, serão apresentados alguns algoritmos básicos para detecção de plágio: a peneiragem, o ladrilhamento ganancioso de cadeias[1], a medida de informação compartilhada[2], maior subseqüência comum[3] e o alinhamento local[4]. Entretanto, em primeiro lugar introduziremos alguns conceitos importantes. O primeiro conceito é o de átomos, obtidos da análise léxica dos programas a serem analisados. Programas a serem analisados consistem, portanto, de cadeias de átomos em uma ordem que representa a estrutura do programa. Outro conceito importante, por ser utilizado em alguns dos algoritmos descritos nesse capítulo, é o de assinaturas "rolantes" de cadeias.

Nesse capítulo nos referiremos a *átomos*[5], que representam os menores elementos a serem tratados pelos sistemas de detecção. Todos os sistemas e algoritmos aqui descritos trabalham sobre cadeias de átomos, e os sistemas de pré-processamento processam programas e geram tais cadeias.

2.1 Preliminares

Primeiro definiremos alguns termos que iremos utilizar bastante daqui para a frente.

k-gramas: os k-gramas de uma cadeia S são as subcadeias de comprimento k contíguas e sobrepostas da cadeia S. Por exemplo, para a cadeia "o rato roeu o rato", ignorando espaços, os 5-gramas são "orato", "rator", "atoro", "toroe", "oroeu", "oeuor", "euora", "uorat" e "orato". Os 10-gramas seriam "oratoroeuo", "ratoroeuor", "atoroeuora", "toroeuorat" e "oroeuorato"

[1] GST
[2] SID
[3] LCS
[4] *Local alignment*
[5] *Tokens*

átomos: definimos átomos como sendo elementos indivisíveis de uma cadeia ou seqüência. Podem ser caracteres, mas não necessáriamente. Os algoritmos a seguir trabalham sobre átomos, e é tarefa do pré-processamento transformar um programa em uma cadeia de átomos. No caso desse trabalho, todos os átomos são representados por bytes.

semelhança: se é dada a distância $d(A, B)$ entre dois programas A e B, podemos obter a semelhança pela equação $sim(A, B) = 1 - d(A, B)$

distância: análogamente, dada a semelhança $sim(A, B)$ entre dois programas A e B, podemos obter a distância pela equação $d(A, B) = 1 - sim(A, B)$

2.2 Assinatura rolante de cadeias

A maioria dos algoritmos discutidos a seguir precisa calcular assinaturas[6] para k-gramas, no Moss (que veremos na seção 3.3), ou para efetuar o algoritmo RKRGST, no caso do JPlag e YAP3 (seções 3.2 e 3.1 respectivamente). Por isso discutiremos uma estratégia para gerar essas assinaturas em tempo linear no tamanho da cadeia.

Dado um k-grama (uma subcadeia de tamanho k) $c_1 \dots c_k$, considere esse k-grama como um número de k dígitos, em uma base b qualquer. A assinatura $H(c_1 \dots c_k)$ desse k-grama seria dada pelo seguinte número:

$$c_1 * b^{k-1} + c_2 * b^{k-2} + \ldots + c_{k-1} * b + c_k$$

Para computar a assinatura do k-grama $c2 \cdots c_{k+1}$ é só subtrair o elemento mais significativo, multiplicar o número por b, e somar c_{k+1}. Então temos a seguinte identidade.

$$H(c_2 \dots c_{k+1}) = (H(c_1 \dots c_k) - c_1 * b^{k-1}) * b + c_{k+1}$$

Porém, Schleimer et al. em [18] mencionam que a soma de c_{k+1} altera apenas os digitos menos significativos da assinatura, de cadeia a cadeia. Isso é devido ao fato de que os valores c_i são números relativamente pequenos. Então ele sugere que a primeira assinatura seja multiplicada por outro b e pela troca da ordem das operações no passo incremental, de maneira que a nova função de assinatura, $H'(c_1 \dots \ c_k)$ seja dada da forma:

$$H'(c_2 \dots c_{k+1}) = ((H'(c_1 \dots c_k) - c_1 * b^{k-1}) + c_{k+1}) * b$$

Essa conta não utiliza aritmética modular; o valor máximo dessa assinatura é ditado pelo tamanho do k-grama e pela base escolhida. Vale notar que essa última fórmula não é compatível com as fórmulas anteriores a ela; os valores gerados para os mesmos valores de c, b e k podem ser diferentes.

[6] *Hash* em inglês.

2.3 Subseqüência comum mais comprida

Dadas duas cadeias, uma subseqüência comum consiste da seqüência de símbolos obtida ao remover os símbolos que não são comuns às duas cadeias. Assim, uma subseqüência comum não necessariamente representa símbolos contíguos das duas cadeias originais; porém a ordem dos símbolos deve ser mantida. Por exemplo, para as cadeias `quinzecorvos` e `oursocantou` uma subseqüência comum seria `uco`. Com essa definição, a descrição do problema da maior subseqüência comum[7] (que pode não ser única) fica óbvia[9].

À primeira vista, o problema do LCS poderia ser aplicado à detecção de plágio. De fato, se dois programas têm uma subseqüência comum grande em relação ao seu comprimento é provável que tenha havido plágio. O problema surge quando o plagiador inverte a ordem de trechos de código. Nesses casos, o comprimento da maior subseqüência comum será reduzido. Isso previne que uma solução para esse problema seja aplicada de uma maneira ingênua. Entretanto, o problema da maior subseqüência comum ainda pode ser útil, portanto discutiremos o algoritmo mais comum para identificar o LCS.

O algoritmo folclórico

Uma peculiaridade do problema do LCS é que não se conhece um algoritmo prático cuja eficiência seja melhor do que o produto dos comprimentos das duas cadeias. De fato, o problema pode ser resolvido por um algoritmo simples com essa complexidade. É denominado algoritmo folclórico por ter sido desenvolvido independentemente em vários locais diferentes, fazendo parte do 'folclore' da área.

A seguir, descreveremos algumas características do algoritmo folclórico. Existem diversas variantes do LCS, algumas mais eficientes do que outras, e muitas otimizações[8].

Sejam $u = u_1u_2 \ldots u_n$ e $v = v_1v_2 \ldots v_m$ duas cadeias de comprimento n e m sobre um alfabeto A. O algoritmo folclórico consiste em computar o comprimento $D(i, j)$ da maior subseqüência comum das subcadeias $u_1u_2 \ldots u_i$ e $v_1v_2 \ldots v_j$. Isso pode ser feito observando que: $D(i, j) = 0$, para $i = 0$ ou $j = 0$, e aplicando a seguinte fórmula de recorrência:

$$D(i,j) = \begin{cases} 1 + D(i-1, j-1) \text{ se } u_i = v_j, \\ \max\{D(i-1, j), D(i, j-1)\} \text{ se } u_i \neq v_j \end{cases}$$

Uma maneira óbvia de se implementar o algoritmo é utilizando uma matriz de valores $D(i, j)$ intermediários para obter $D(m, n)$. Essa técnica de programação dinâmica tem o lado ruim de utilizar espaço $m \times n$. Porém, é possível melhorar esse resultado se se deseja apenas o comprimento da maior subseqüência, escolhendo com cuidado a ordem em que são calculados os valores de $D(i, j)$. Conforme a fórmula acima, para se determinar o

[7] *Longest common subsequence* em inglês, ou LCS

valor de um $D(i, j)$ qualquer é necessário ter apenas a linha anterior da matriz e a linha sendo calculada no momento. Infelizmente, se se deseja obter a maior subseqüência, é necessário ter toda a matriz.

Aplicação para casos de plágio

Apesar de que o LCS não pode ser aplicado diretamente ao problema de plágio de maneira produtiva, é possível utilizá-lo para tentar obter dados de plágio aplicando transformações e normalizações nos dados de entrada. Em particular, se ordenarmos os átomos de um programa a ser testado, podemos tentar maximizar o tamanho da máxima subseqüência comum. Essa ordenação deve ser feita com cuidado de maneira a não quebrar a estrutura do programa, enquanto que eliminando diferenças que existem apenas em virtude de estarem ordenados diferentemente (por exemplo: dois programas em C idênticos, mas que trocam a ordem dos *ifs* e *elses*). Veremos mais detalhes sobre essa ordenação na seção 5.2.

2.4 Peneiragem

A idéia da *peneiragem*[8], definida por Schleimer et. al[18] é de escolher uma assinatura para um documento, de forma que a assinatura identifique o documento e que essa mesma assinatura possa ser usada para detectar semelhanças. A peneiragem utiliza o conceito de k-gramas. Um documento é divido em k-gramas e para cada k-grama é calculada uma assinatura — idealmente utilizando uma função de assinatura rolante, como descrito em 2.2. É sobre esses valores que é aplicada a peneiragem, que reduz drasticamente o tamanho da assinatura (uma representação em k-gramas terá quase tantos k-gramas quantos há átomos no documento).

Um dos requisitos da peneiragem, no que se refere a emparelhamentos de subcadeias de documentos a serem comparados, é que qualquer emparelhamento de comprimento igual ou maior ao *limiar de garantia* t será detectado. Analogamente, qualquer casamento de comprimento menor que o *limiar de ruído* k deve ser ignorado. Esses valores, t e k são escolhidos pelo usuário, e serão utilizados para determinar tamanho de *janelas*. Uma janela é um agrupamento contíguo de assinaturas calculadas.

Por exemplo, considerando os 5-gramas da frase "o rato roeu a roupa": 'orato', 'rator', 'atoro', 'toroe', 'oroeu', 'roeua', 'oeuar', 'uarou', 'aroup' e 'roupa', suponhamos ter como assinaturas dos 5-gramas os números 20, 63, 40, 74, 33, 05, 88, 53 e 15. Suas janelas de tamanho 4 seriam:

[8] *Winnowing*

[20 63 40 74]
[63 40 74 33]
[40 74 33 05]
[74 33 05 88]
[33 05 88 53]
[05 88 53 15]

Para a peneiragem, o tamanho da janela é dado como $w = t - k + 1$. Claramente, o valor de k deve ser menor que ou igual ao valor de t.

Para cada janela, um valor é escolhido da seguinte forma: em cada janela, escolha o valor mínimo. Se houver mais de um valor mínimo, escolha o que estiver mais à direita. Guarde todos os valores escolhidos: esses formarão a assinatura do documento. Para as janelas do exemplo anterior, as escolhas seriam: 20, 33, 05. Note que as últimas três janelas escolhem o mesmo valor como o mínimo. Para efeitos de comparação, junto com cada valor escolhido deve ser armazenada a posição dentro do programa daquele valor. No nosso caso, teríamos os 3 pares (20,0), (33, 4) e (05, 5) (supondo que começamos na posição 0).

Existe uma desvantagem no algoritmo de emparelhamento. Se os dados a serem comparados são muito homogêneos (por exemplo, cadeias de dados com muitos caracteres repetidos) haverá muitos empates para uma dada janela, e serão escolhidos mais valores do que seria necessário. Por isso Schleimer et al.[18] propuseram uma alteração ao algoritmo de peneiragem, a peneiragem robusta[9]. A diferença nesse algoritmo é que no caso de empate, deve-se tentar quebrá-lo escolhendo o mesmo valor que a janela da esquerda escolheu. Dessa forma, para cadeias de entropia baixa, como a mencionada acima, apenas um valor será escolhido para a subcadeia de caracteres repetidos.

No algoritmo 1 vemos como funciona a peneiragem. Como a geração de assinaturas e a peneiragem podem ser feitas linearmente no comprimento da cadeia, a peneiragem tem complexidade linear. A etapa de comparação das assinaturas, porém, ainda continua quadrática.

[9] *Robust winnowing*

Algoritmo 1 Peneiragem robusta

Entrada: Cadeia S
Saída: *assinaturas*
  $assinaturas \leftarrow \{\}$
  Dada uma cadeia S, de comprimento L compute as assinaturas
  de comprimento k, $S_1, S_2, \ldots S_{L-k+1}$
  Defina as W janelas de tamanho w contendo as $L - k + 1$ assinaturas de S: $S_1, S_2, \ldots S_w; S_2, S_3, \ldots S_{w+1}; \ldots S_{L-k-w}, S_{L-k-w+1}, \ldots S_W$
  para Cada janela ω **faça**
    Escolha a menor assinatura da janela, s
    se s é diferente da última assinatura escolhida **então**
      $assinaturas \leftarrow assinaturas \oplus \{s\}$
    fim se
  fim para

2.5 Ladrilhamento ganancioso

O emparelhamento ganancioso[10] de ladrilhos foi proposto por Wise [22]. A idéia desse algoritmo, dadas duas cadeias A e B, é encontrar uma cobertura maximal por *ladrilhos*[11] (subcadeias de átomos contíguos comuns a A e B) tal que:

- a soma dos comprimentos das subcadeias comuns de A e B tem o maior valor possível e
- não há sobreposição entre nenhum ladrilho em A ou em B.

A figura 2.1 passa a idéia de como é feito o ladrilhamento sobre duas cadeias, os trechos com a mesma textura representando os ladrilhos iguais.

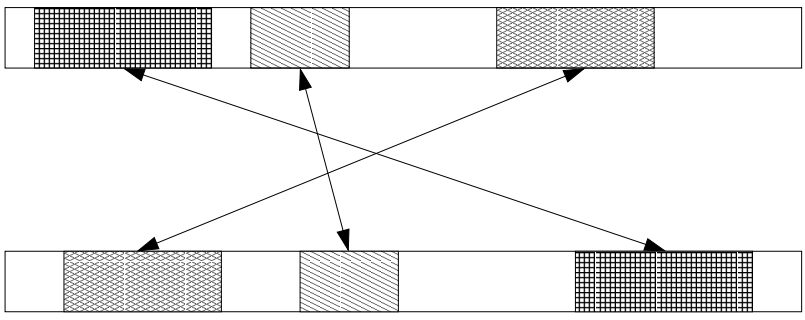


Figura 2.1: Ladrilhamento ganancioso

[10] *Greedy matching*
[11] *Tiles*

Além disso, o algoritmo ignora qualquer emparelhamento de cadeias cujo comprimento seja menor que um valor M (que pode ser definido pelo usuário, ou mesmo um valor fixo no programa; varia de implementação a implementação).

Existem variantes do algoritmo: veremos a seguir o algoritmo original, que melhor passará a sua idéia.

Algoritmo GST

No algoritmo 2, a variável *ladrilhos* é a saída do algoritmo: o conjunto de ladrilhos que obedece às restrições descritas anteriormente. Já *emparelhmax* e *emparelhamentos* são o comprimento do maior emparelhamento encontrado até então (ou então o mínimo comprimento de emparelhamentos) e o conjunto dos emparelhamentos, respectivamente. Finalmente, $emparelhamento(a, b, j)$ representa um emparelhamento que tem início em A_a e B_b e comprimento j. Deve-se notar que para $M > 1$ o conjunto de subcadeias comuns pode não ser maximal, pois cadeias comuns de comprimento menor que M não serão detectadas. Isso não se apresenta como problema na utilização do algoritmo, uma vez que em geral não se tem interesse em emparelhamentos de tamanho muito pequeno. Esse algoritmo tem complexidade $\Theta\left((|A| + |B|)^2\right)$.

Otimizações

Existe uma otimização sobre o algoritmo, utilizada em todas as suas implementações até então, que gera assinaturas para todas as subcadeias de A e B, variando o comprimento dessas subcadeias durante a execução do algoritmo. A geração de assinaturas pode ser feita em tempo linear, como visto nas preliminares, na seção 2.2. As assinaturas de uma das cadeias são inseridas numa tabela de espalhamento[12], e as assinaturas da segunda cadeia são usadas como chaves para uma busca. Isso facilita a busca inicial, uma vez que o número de assinaturas em cada cadeia é linear no seu comprimento. Além disso, para valores maiores de M, é provável que uma dada assinatura aparecerá poucas vezes na segunda cadeia.

Essa alteração recebe o nome de *Running Karp-Rabin Greedy String Tiling*, ou RKRGST, que é como será referida daqui em diante. O nome é devido ao fato de que Wise, autor do RKRGST, baseou-se no algoritmo de Karp-Rabin[12] para casamento de cadeias.

O algoritmo original define s, o comprimento da cadeia da qual foi calculada uma assinatura como sendo inicialmente igual a M. Após calcular as assinaturas, o algoritmo efetua a busca: quando um casamento de assinaturas é encontrado, o algoritmo tenta

[12] *Hash table*

Algoritmo 2 GST

Entrada: Cadeias A e B
Saída: *ladrilhos*

```
ladrilhos ← {}
repita
  emparelhmax ← M
  emparelhamentos ← {}
  para todo átomo não marcado A_a em A faça
    para todo átomo não marcado B_b em B faça
      j ← 0
      enquanto A_{a+j} = B_{b+j} ∧ A_{a+j} não foi marcado ∧B_{b+j} não foi marcado faça
        j ← j + 1
      fim enquanto
      se j = emparelhmax então
        emparelhamentos ← emparelhamentos ⊕ emparelhamento(a, b, j)
      senão se j > emparelhmax então
        emparelhamentos ← {emparelhamento(a, b, j)}
        emparelhmax = j
      fim se
    fim para
  fim para
  para todo emparelhamento(a, b, emparelhmax) ∈ emparelhamentos faça
    para j = 0 a emparelhmax − 1 faça
      marca(A_{a+j})
      marca(B_{b+j})
    fim para
    ladrilhos = ladrilhos ∪ emparelhamento(a, b, emparelhmax)
  fim para
até que emparelhmax ≤ M
```

estender o casamento, símbolo por símbolo. Caso o casamento venha a ser grande demais (maior que duas vezes o valor de s) o algoritmo reinicializa e recalcula as assinaturas, repetindo o processo. O processo respeita os ladrilhos já marcados.

Esse processo de recalcular as assinaturas é custoso; por isso, Prechelt et al.[17] utilizaram-se de outro mecanismo: ao invés de usar assinaturas de cadeias de comprimento variável, o JPlag fixa o comprimento das cadeias como sendo M e utiliza as assinaturas como um mecanismo para acelerar a busca de possíveis casamentos. Algum cuidado extra é necessário para evitar que ladrilhos marcados sejam utilizados mais de uma vez. O algoritmo 3 demonstra a implementação. Esse algoritmo continua com complexidade quadrática, mas na prática aproxima-se de $\Theta\left((|A| + |B|)^{1.2}\right)$.

Algoritmo 3 RKRGST utilizado em JPlag

Entrada: Cadeias A e B
Saída: *ladrilhos*
  $H_A \leftarrow$ pares (assinatura,pos) da cadeia A
  $H_B \leftarrow$ pares (assinatura,pos) da cadeia B
  $ladrilhos \leftarrow \{\}$
  $emparelhmax \leftarrow M$
  para todo $(h_B, pos(h_B))$ em H_B e $pos(h_B)$ não está marcada **faça**
    se h_B está em H_A e sua posição em A não está marcada **então**
      {Verifique o casamento}
      $p_A \leftarrow pos(h_A)$
      $p_B \leftarrow pos(h_B)$
      $i \leftarrow M - 1$
      enquanto $A_{p_A+i} = B_{p_B+i} \wedge i \leq 0$ **faça**
        $i \leftarrow i - 1$
      fim enquanto
      se $i \leq 0$ **então**
        Continue para próxima iteração
      fim se
      $i \leftarrow 1$
      $j \leftarrow emparelhmax$
      enquanto $A_{p_A+i} = B_{p_B+i}$ e não atingimos o final de A ou B **faça**
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
      fim enquanto
      se $j > emparelhmax$ **então**
        $emparelhmax \leftarrow j$
        $emparelhamentos \leftarrow emparelhamento(p_A, p_B, j)$
        Marca temporariamente o ladrilho em A e B
      senão se $j = emparelhmax$ **então**
        se ladrilho em A e B não está marcado temporariamente **então**
          $emparelhamentos \leftarrow emparelhamentos \oplus emparelhamento(p_A, p_B, j)$
          Marca o ladrilho em A e B temporariamente
        fim se
      fim se
    fim se
    para todo (a, b, j) em $emparelhamentos$ **faça**
      se $j = emparelhmax$ **então**
        Marca o ladrilho com início na posição a de comprimento j na cadeia A
        Marca o ladrilho com início na posição b de comprimento j na cadeia B
        $ladrilhos = ladrilhos \cup emparelhamento(a, b, j)$
      fim se
    fim para
  fim para

Alguns comentários sobre esse algoritmo: após encontrar, na tabela de espalhamento, a cadeia correspondendente, é necessário validá-la. No caso é feita uma pequena otimização: como os textos sendo analisados são programas, o autor do JPlag[17] descobriu que diferenças comumente se apresentam no final das subcadeias. Por isso a análise é feita de trás para frente, para depois tentar estender o casamento o máximo possível.

Além disso, devido à modificação sobre o RKRGST que permite que as assinaturas não sejam recalculadas a cada iteração, é necessário um mecanismo mais complexo para controlar as assinaturas. Especificamente, é necessário marcar as subcadeias dentro de cada iteração, assinalado no algoritmo como uma marcação temporária.

2.6 Alinhamento local

O algoritmo de alinhamento local[13][10] tem como objetivo determinar a semelhança entre duas cadeias. Dadas duas cadeias, para cada elemento de uma cadeia existem três possibilidades. Podemos ter um casamento, uma diferença ou um espaço, com pesos m, d e g, respectivamente[14]. Dependendo da implementação, podemos escolher pesos arbitrários para m, d e g, embora seja comum escolher um valor para m maior que d ou g, e também convém escolher um valor de d maior que g. Por exemplo, para o exemplo dado abaixo escolhemos $m = 1$, $d = -1$ e $g = -2$.

A idéia central do algoritmo é achar um casamento máximo para duas cadeias. É preferível encontrar cadeias que diferem em um caractere do que ser obrigado a inserir um espaço: por isso d tende a ser maior que g (um casamento que difere em um caractere vai ter um valor menor que um casamento que precisa ser alinhado). Por exemplo, para as cadeias “capivara” e “arara”:

c	a	p	i	v	a	r	a
	a	r			a	r	a
-2	1	-1	-2	-2	1	1	1

No exemplo acima, se $m = 1$, $d = -1$ e $g = -2$, a cadeia com o maior valor seria “ara” final de “capivara” com o “ara” final de “arara”. Para o algoritmo, o valor entre um bloco de pares é a soma dos valores de todos os pares, e o valor de alinhamento é o valor do maior bloco de pares. Nesse caso, o valor do maior alinhamento é 3.

O valor ótimo para um alinhamento é o valor máximo para todos os alinhamentos. O algoritmo utilizado para calcular o valor ótimo pode ser calculado por programação dinâmica, como a seguir: defina $D(i, j)$ como o valor ótimo de alinhamento para duas

[13] *Local alignment*

[14] Os nomes provêem das palavras *match*, *difference* e *gap*

subcadeias $s_1..s_i$ e $t_1..t_j$. Estamos buscando por $\max_{1\leq i\leq|s|,1\leq j\leq|t|} D(i,j)$. Dessa forma, o alinhamento local pode ser visto como uma generalização do LCS: se aplicarmos pesos uniformes a esse algoritmo teremos resultados bastante semelhantes ao do LCS.

Definimos *valor* como a seguir:

$$valor(s_i,t_j) = \begin{cases} m & \text{se } s_i = t_j; \\ d & \text{caso contrário.} \end{cases}$$

A solução pode ser computada utilizando a seguinte relação de recorrência:

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + valor(s_i,t_j) \\ D(i-1,j) + g, \\ D(i,j-1) + g, \\ 0 \end{cases}$$

As condições de contorno são dadas por $D(0,i) = i*g$ e $D(j,0) = j*g$. Os elementos da matriz D podem ser computados inicializando a primeira linha e coluna para as condições de contorno e, depois, avaliando os elementos da esquerda para a direita e de cima para baixo. Ou, assim como no LCS, os valores de $D(i,j)$ podem ser calculados utilizando apenas três valores anteriores, se a ordem do cálculo for feita com cuidado.

2.7 Informação Compartilhada

A idéia de informação compartilhada foi definida por Li et al. em [13] como uma medida de distância entre seqüências. Utiliza a noção de entropia algorítmica ou complexidade de Kolmogorov.

A complexidade Kolmogorov $K(s)$ representa o tamanho do menor programa que imprime s como sua saída. A notação $K(s|t)$ representa quanta informação a cadeia t desconhece de s; mais precisamente, representa o tamanho do menor programa que, tendo t como entrada, imprime s. Essa complexidade, no entanto, não é computável.

A medida de distância é dada pela seguinte expressão:

$$d(x,y) = 1 - \frac{K(x) - K(x|y)}{K(xy)}$$

Onde $K(xy)$ é a complexidade da concatenação de x e y. Ademais, no artigo mencionado acima é provada uma noção de "universalidade". Para uma medida de distância $D(x,y)$, normalizada para o intervalo $[0,1]$, para quaisquer cadeias x e y, o seguinte é verdade:

$$d(x, y) \leq 2D(x, y) + O(1)$$

Chen et al. definem essa equação em [6] como o seguinte: se dois programas são semelhantes nessa medida $D(x, y)$, então eles serão semelhantes na medida $d(x, y)$. Dessa maneira, chegam à conclusão de que $d(x, y)$ é comprovadamente impossível de se trapacear.

Aproximação da medida

No mesmo artigo, Chen et al.[6] mencionam um problema com a métrica definida acima: a complexidade de Kolmogorov não é computável. Portanto, os mesmos sugerem utilizar como heurística o comprimento da cadeia obtida pela compressão. Como a medida de Kolmogorov requer a compressão perfeita da cadeia, quanto melhor a compressão obtida, melhor será a métrica. Denotando esse comprimento por $Comp(x)$, temos que:

$$d(x, y) \approx 1 - \frac{Comp(x) - Comp(x|y)}{Comp(xy)}$$

O valor $Comp(x|y)$ representa uma compressão condicional. Podemos aproximá-la por $Comp(yx) - Comp(y)$.

Podemos tentar validar essa aproximação para quando as cadeias comparadas são idênticas. Utilizando a aproximação mencionada acima temos:

$$d(x, y) \approx 1 - \frac{Comp(x) - Comp(yx) + Comp(y)}{Comp(xy)}$$

Se $x = y$, então a fórmula acima se torna:

$$d(x, x) \approx 1 - \frac{2Comp(x) - Comp(xx)}{Comp(xx)}$$

Supondo uma compressão ideal, podemos concluir que $Comp(xx) = Comp(x) + \epsilon$ para um valor muito baixo de ϵ. Então novamente, podemos reduzir a expressão acima para:

$$d(x, x) \approx 1 - \frac{Comp(x) - \epsilon}{Comp(x) + \epsilon}$$

Nesse caso, como é esperado, o valor de $d(x, x)$ é próximo de zero, já que temos uma medida de distância. Esse resultado será útil mais a seguir.

Outra verificação: se x é totalmente diferente de y, então:

$$Comp(x|y) \cong Comp(x) + \epsilon$$

$$Comp(xy) \cong Comp(x) + Comp(y)$$

Portanto, utilizando as equações acima:

$$d(x, y) \approx 1 - \frac{\epsilon}{Comp(x) + Comp(y)} \approx 1 - \epsilon'$$

que é muito próximo de 1, confirmando a aproximação.

Capítulo 3

Sistemas de detecção de plágio

Neste capítulo descreveremos alguns sistemas de detecção de plágio para programas.

3.1 YAP3

O YAP3[23] foi desenvolvido por Wise para ser um sistema de detecção de plágio em trabalhos de programação, utilizando um algoritmo proposto pelo mesmo: o *Running Karp-Rabin Greedy String Tiling*. A vantagem desse algoritmo sobre outros utilizados até então é que ele é robusto em relação a situações de plágio em que apenas trechos de programas são copiados, e seu uso de ladrilhamento guloso o torna robusto em relação a mudanças de ordem. O YAP3 foi um dos primeiros a utilizar análise de estruturas ao invés de contagem de atributos.

O YAP3 é o único dos sistemas estudados que está totalmente disponível; ao contrário do MOSS e do JPlag, não é um sistema *on-line*, pode ser executado localmente. O sistema, junto com seu código fonte, pode ser obtido do seguinte endereço: `http://www.bio.cam.ac.uk/~mw263/YAP.html`.

O YAP3 processa bastante os dados de entrada, comparado com os outros sistemas estudados: o objetivo é gerar uma seqüência de átomos que representa o programa. Para fazer isso o YAP3 faz o seguinte:

1. Remove comentários e constantes de caracteres.

2. Converte letras maiúsculas para minúsculas.

3. Converte sinônimos para uma forma comum: sinônimos podem ser funções da biblioteca comum que efetuam o mesmo trabalho (*strcpy* e *strncpy* são dados como exemplo.

4. Expande a primeira chamada de uma função para o corpo da função; chamadas subseqüentes a essa função são substituídas por um átomo que representa essa função. Esse passo pode aumentar o tamanho do programa significativamente.

5. Remove todo átomo que não faz parte do léxico da linguagem — átomos que não são palavras-chave, chamadas de função da biblioteca padrão. Isso eliminaria, por exemplo, variáveis do corpo do programa.

Após esses passos é gerada a seqüência de átomos, e essas sequências são comparadas, par a par, utilizando o algoritmo RKRGST descrito anteriormente. Ele fornece então uma lista de programas semelhantes e uma lista de programas pequenos demais para serem comparados.

3.2 JPlag

JPlag foi implementado em Java por Prechelt et al.[17], disponível para utilização no endereço `http://www.jplag.de/`.

O JPlag utiliza uma versão modificada do algoritmo RKRGST, descrito na seção 2.5.

Dividimos a descrição do funcionamento do JPlag em três partes: pré-processamento, onde o JPlag processa os programas a serem comparados e escolhe dois par a par; o processamento, onde o JPlag compara as seqüências de átomos resultante do pré-processamento e determina qual a porcentagem de semelhança entre os programas sendo comparados; e o pós-processamento, onde o JPlag organiza os resultados e os apresenta para o usuário.

Funcionamento

Como a maioria dos sistemas de detecção de plágio, o JPlag é um sistema fechado, cujos detalhes de implementação não são liberados ao público. No entanto, dos sistemas estudados, o JPlag é o mais aberto, com mais informação sobre sua implementação. Do artigo de Prechelt et al.[17] e da observação do funcionamento do sistema foi possível verificar seu funcionamento, e categorizá-lo da seguinte forma.

O sistema funciona como uma aplicação Java WebStart que opera via uma interface gráfica. O usuário tem como opções definir a sensibilidade do programa e submeter um trecho de código que será utilizado como código-base, entre outros. O usuário pode enviar vários arquivos dentro de um mesmo diretório, um arquivo por aluno, ou submeter um diretório com vários subdiretórios, um subdiretório por aluno. Os resultados são processados imediatamente e os resultados são entregues como um conjunto de páginas HTML locais.

Pré-processamento

Após submetidos os programas para serem comparados, os mesmos são processados por um analisador léxico ou um analisador sintático, dependendo da linguagem (C e C++ utilizam um analisador léxico, mas Java utiliza um analisador sintático). Esse passo tem o propósito de gerar uma seqüencia de átomos a partir dos programas. A vantagem do analisador sintático é que o mesmo pode obter mais dados sobre o programa sendo analisado. Por exemplo, em C e C++, um "{" é apenas um "abre chave", enquanto que em Java esse mesmo "{" pode ser o início de uma declaração de classe, o início de um método, o início de um laço etc.

Esse processamento tem o propósito de ignorar mudanças ingênuas, como mudança de nome de variáveis. Uma declaração de variável será independente do tipo da variável, ou do seu nome. Esse processamento não se aprofunda mais na sintaxe do programa; estruturas de laço, como *for* e *while*, são considerados como iniciadas por átomos diferentes.

Processamento

Após esse passo, o JPlag começa a comparar cada seqüência de átomos, par a par. Note-se que nesse algoritmo, se houver um espaço entre ladrilhos menor que o tamanho mínimo de emparelhamento, esse trecho ficará descoberto. Ou seja, o algoritmo para s maior que 1 não é ótimo. No caso, interessa mais obter resultados pertinentes do que conseguir uma cobertura ótima das duas cadeias.

Pós-processamento

Após calculadas as semelhanças entre seqüências, é computada a percentagem de cada programa (dada pelo tamanho do casamento dividido pelo tamanho do programa sendo comparado) que foi coberta pelo algoritmo descrito acima. Os pares de programas são ordenados de acordo com a sua semelhança e é gerada a página HTML correspondente àquela submissão.

O índice de semelhança do JPlag é dado pela função $sim(A, B)$, definida pelas seguintes equações, onde $|l|$ é o comprimento do ladrilho:

$$sim(A, B) = \frac{2 \cdot cobertura(ladrilhos)}{|A| + |B|}$$

$$cobertura(ladrilhos) = \sum_{l \in ladrilhos} |l|$$

Vimos a função *emparelhamento* e o conjunto de resultados *ladrilhos* na seção 2.5.

3.3 MOSS

O MOSS (Measure of Software Similarity) é um sistema desenvolvido por Schleimer et al.[18], disponível no endereço `http://theory.stanford.edu/~aiken/moss/`.

Esse sistema utiliza o algoritmo de peneiragem para efetuar a comparação de documentos. Detalhes de sua implementação não foram liberados ao público, sob o risco de que, ao conhecer o funcionamento do sistema, alguém poderia descobrir uma maneira de "enganar"o sistema. Os arquivos a serem enviados são submetidos via um *script* fornecido pelo autor, e os resultados são exibidos numa página em HTML; a interface de exibição de resultados muito semelhante à do JPlag. O MOSS aceita uma grande gama de linguagens, como C, Java e LISP; o MOSS também efetua comparações sobre textos comuns. Possivelmente o valor de k dos k-gramas varia de acordo com a linguagem; há menção de que, para determinados tipos de texto, um valor baixo de k pode acusar construções comuns como plágio, enquanto que um valor maior de k, para o mesmo documento, acusaria plágio corretamente.

Funcionamento do MOSS

Existem poucos detalhes sobre o funcionamento do MOSS; no artigo de Schleimer et al. muita informação é omitida sobre o funcionamento e implementação. Porém, o mesmo artigo descreve os seguintes passos do processo de detecção de plágio.

Como mencionado anteriormente, programas são submetidos ao sistema utilizando um *script* de envio fornecido pelo autor. Não é possível especificar a sensibilidade desejada, embora seja possível enviar um código esqueleto (por exemplo, código oferecido pelo instrutor e que deve ser utilizado por todos os alunos) e ignorar código que ocorre vezes demais. Devido ao fato do *script* usar linha de comando, a especificação de quais programas enviar ao sistema é mais flexível. Os resultados são processados imediatamente, e o *script* aponta a uma página da internet onde o usuário pode acessar os resultados de sua submissão.

Pré-processamento

Não há detalhes específicos sobre como funciona o pré-processamento do MOSS. Sabemos que ele remove dados que acredita não serem importantes e, devido a algumas observações do funcionamento do sistema, podemos supor que ele não faz grande manipulação sintática do documento. Como nomes diferentes para variáveis gerariam assinaturas diferentes, nomes de variáveis são ignorados. Além do pré-processamento feito por um analisador léxico (ou sintático) os documentos a serem comparados são separados em k-gramas, e depois em janelas.

Processamento

O MOSS utiliza a *peneiragem robusta* para escolher valores de espalhamento para cada janela. Além disso, é armazenada informação posicional. Depois disso é montado um índice invertido, mapeando todas os pedaços das assinaturas a posições em todos os documentos. O MOSS então calcula a assinatura para todos os documentos novamente, e as assinaturas são usadas para consultar o índice. Isso fornece a lista de todas as assinaturas que casam com um dado documento.

Pós-processamento

As listas então são ordenadas por documento, fornecendo uma lista de pares de documentos com assinaturas em comum. Essa lista é ordenada, também, pelo número de assinaturas em comum, e os maiores emparelhamentos são dados ao usuário. Os resultados são exibidos em uma página em HTML. Há um problema na exibição, porém: devido ao fato de que apenas alguns k-gramas são escolhidos, alguma informação é perdida. Particularmente, o início e fim de cada emparelhamento às vezes não aparecem na exibição final.

O MOSS não fornece dados explícitos de como faz o cálculo da semelhança entre dois programas.

3.4 SIM

O SIM utiliza o algoritmo de alinhamento local[10]. O sistema foi implementado primariamente em C++, com a interface gráfica implementada em Tcl/Tk.

Pré-processamento

O sistema, antes de processar algum documento, o quebra em cadeias de átomos. É interessante notar que o SIM não é muito seletivo quanto a quais tipos de átomos ele ignora; diferenças de nomes são levadas em consideração, armazenadas em uma tabela de símbolos que é depois lida pelo programa durante a comparação. Comentários são considerados como um tipo de átomo, e espaços em branco são descartados.

Processamento

Após a quebra dos programas em átomos, o arquivo de átomos é quebrado em módulos (funções ou subrotinas). Cada módulo é comparado com a cadeia de átomos do primeiro programa; a técnica de alinhamento local calcula a semelhança de cada módulo

do segundo programa em relação ao primeiro programa. Isso permite que o SIM consiga determinar semelhança entre programas mesmo quando houve reordenação de módulos de um programa para o outro. É interessante notar que mudanças de nomes de variáveis e de funções podem influenciar na semelhança entre programas.

A medida de semelhança é normalizada pela seguinte fórmula, sendo $semelhanca(x, y)$ a soma dos valores obtidos por alinhamento local para cada um dos blocos dos programas p_1 e p_2:

$$s = \frac{2 \times semelhanca(p_1, p_2)}{semelhanca(p_1, p_1) + semelhanca(p_2, p_2)}$$

O valor s será, então, um valor entre 0 e 1.

Pós-processamento

Após calcular a semelhança entre pares de programas, os valores de s são exibidos graficamente. Vale notar que o passo de comparação, utilizando alinhamento local, executa em tempo quadrático, apesar de haver planos para melhorar seu desempenho. Já que cada par de programas deve ser comparado, existe um número quadrático dessas comparações — portanto o tempo de execução do SIM não é insignificante.

3.5 SID

O SID foi desenvolvido por Chen et al.[6] na Universidade de Waterloo. Utiliza a noção de informação compartilhada, como visto anteriormente. Está disponível *online*, no endereço `http://genome.math.uwaterloo.ca/SID/`.

Pré-processamento

Como todos os outros algoritmos estudados aqui, o primeiro passo converte o programa em cadeias de átomos. Não há grandes detalhes sobre o algoritmo utilizado, mas como a linguagem aceita pelo SID é Java, eles utilizam um compilador que gera bytecode, e sobre esse código gerado efetuam sua análise.

Processamento

Como vimos na seção 2.7, a medida de informação compartilhada $d(x, y)$ depende da complexidade de Kolmogorov. Contudo, a complexidade Kolmogorov não é computável. Portanto é necessário trabalhar com uma aproximação para $K(x)$, utilizando uma função

de compressão. Essa aproximação, $Comp(x)$ é calculada como sendo o tamanho da compressão da cadeia x, e serve como uma cota superior para $K(x)$.

Em [7] Chen et al. definem o funcionamento do algoritmo de compressão utilizado pelo SID para cadeias de DNA, denominado GenCompress. O mesmo utiliza um algoritmo Lempel-Ziv modificado, trabalhando sobre casamento aproximado de cadeias. Para efetuar isso, o mesmo calcula a distância de edição para transformar uma cadeia na outra.

Além disso, não há limite quanto ao tamanho do dicionário utilizado pelo algoritmo. Apesar disso, para limitar o tamanho do espaço de busca, o algoritmo estabelece um limite ao tamanho dos prefixos utilizados na compressão: para uma cadeia u, o número de operações de edição sobre uma subcadeia de tamanho k do prefixo s de u não deve ultrapassar um limite pré-estabelecido b.

Além disso, o algoritmo mantém uma *função de ganho*, que calcula se a operação efetuada oferece um ganho à compressão. Para determinar o prefixo ótimo de u para uma cadeia vu, com v já comprimido, o algoritmo deve maximizar essa função de ganho.

A aplicação direta desse algoritmo para textos ou mesmo programas pode ser questionada, uma vez que Chen et al. também mencionam que existe uma grande diferença entre algoritmos de compressão para texto e para cadeias de DNA.

Pós-processamento

A saída é exibida em HTML, e o critério para determinar semelhança é dado através da medida de distância $d(x, y)$.

Capítulo 4

Idéias para melhoria

Após a pesquisa bibliográfica, o trabalho de implementação teve ênfase em dois aspectos: o desenvolvimento de sistemas de detecção e um foco no pré-processamento. Os algoritmos de detecção cujos detalhes estavam disponíveis (ou cujo funcionamento era mais fácil de se deduzir) foram implementados, para melhor entender seu funcionamento e efetuar melhorias. Todos os casos de teste foram programas na linguagem C.

No entanto, testes efetuados nos sistemas de detecção de plágio exibiram vulnerabilidades: dadas certas transformações (diferentes para diferentes sistemas) era fácil garantir um falso negativo. Testes subseqüentes nos permitiram prever parte da razão para tais vulnerabilidades: o pré-processamento ignorava grande parte da estrutura do programa, ou então deixava de descartar trechos sem importância.

- Chaves são sempre consideradas significativas, mesmo quando envolvem apenas um comando. Essas chaves agem como ruído na representação do programa, e sistemas de detecção de plágio que tentam detectar blocos comuns podem falhar. Isso ocorre com o JPlag, por exemplo, enquanto que o MOSS é mais robusto quanto a isso.

- Estruturas diferentes, mas com mesmo valor semântico (as diferentes construções de laço, em C, por exemplo) também podem gerar falsos negativos, uma vez que o pré-processamento se limita à análise léxica. Isso parece acontecer com todos os sistemas analisados.

- Reordenação de comandos, quando os comandos tem comprimento grande não conseguem enganar os sistemas, de modo geral. A maioria consegue detectar esse tipo de embaralhamento. Quando o tamanho do comando reordenado começa a se aproximar do tamanho do menor trecho detectado, começamos a ter um índice de semelhança pequeno, alguns inclusive que poderiam ser vistos como falsos negativos.

4.1 Desenvolvimento de sistemas de detecção

Dois dos algoritmos descritos anteriormente foram implementados e utilizados para detectar plágio: o RKRGST e a peneiragem. Ambos foram implementados em Python, com um sistema de pré-processamento desenvolvido em Bison, Flex e C. Versões iniciais tinham como interesse apenas mostrar o nível de semelhança entre dois programas. Versões subseqüentes tiveram seu objetivo alterado para colocar mais ênfase no pré-processamento e pós-processamento, com o propósito de obter um ambiente que pudesse ser utilizado em combinação com qualquer algoritmo de comparação.

Houve tentativa de implementar o sistema utilizando a medida de informação compartilhada, o mesmo algoritmo que o SID utiliza, porém não foi possível obter detalhes sobre o algoritmo de compressão utilizado; sabe-se apenas que o sistema utiliza uma variante do LZ77 com alguns recursos adicionais: em particular, a versão utilizada parece conseguir detectar cadeias aproximadas. Devido ao prazo relativamente curto e o retorno envolvido foi decidido não pesquisar algoritmos de compressão e focar em outros aspectos da detecção. Vale notar que o próprio SID não consegue determinar certos casos de plágio, gerando falsos negativos. Em geral, o sistema não consegue tratar os casos descritos acima, com seus resultados oferecendo divergências maiores que o MOSS ou o JPlag.

4.2 Pré-processamento

Determinamos que grande parte do problema de detecção está associado ao pré-processamento. A maioria dos sistemas utiliza um pré-processamento que não passa de uma análise léxica, descartando grande parte da informação do programa. O pré-processador de C do JPlag, por exemplo, descarta expressões, enquanto que todos os sistemas tratam chaves ({}) como sendo significativas, mesmo quando não há motivo. Isso "polui" o programa e tende a confundir os sistemas de detecção.

Como solução, decidimos fazer um processamento mais extenso do programa: além da análise léxica vista até então, é utilizada análise sintática. Isso permite manter informação pertinente que outros sistemas ignoram (como o conteúdo de expressões) e, ao mesmo tempo, ignorar construções sem valor sintático (como chaves supérfluas). Existe um problema com essa estratégia, porém: ao manter muita informação sobre um programa, pequenas mudanças passam a ter mais significado e, sem o devido cuidado, é possível gerar variações que poderiam ser consideradas diferentes o suficiente. Por exemplo, mudar a ordem de um comando if-else geraria um programa significativamente diferente.

Por isso decidimos ordenar o programa lexicograficamente, de maneira recursiva. Por exemplo, em um bloco if-else, a escolha cujo valor lexicográfico (a ser explicado na seção 5.2) é menor é posta antes do outro bloco. Dessa forma, se um plagiador muda a ordem dos

```
if( a < b )
  puts(s);
else {
  if (t)
    s=a;
  else
    s=foo(b);
}
```

```
if( a >= b ) {
  if(!t)
    s=foo(b);
  else
    s=a;
}
else
  puts(s);
```

Figura 4.1: Trecho de código e seu 'plágio'

blocos em um if-else, por exemplo, na sua tentativa de plágio, a ordenação "normalizaria" o programa e geraria uma representação consistente. Essa funcionalidade é opcional e programas podem ser analisados sem ser feita a ordenação. Como essa ordenação ocorre para cada construção do programa, e se estende de maneira recursiva, dois programas plagiados apenas por reordenação apresentariam uma estrutura muito semelhante, se não idêntica. Vale notar que para os outros sistemas reordenação é uma das maneiras de os ludibriar, apesar de que na maioria dos casos de plágio envolvendo reordenação, a mesma é feita envolvendo blocos de código de tamanho considerável. O JPlag, por utilizar RKRGST, consegue encontrar um ladrilho grande nesses casos, por exemplo.

Ilustramos esse conceito através de um exemplo apresentado nas figuras 4.1, 4.2 e 4.3. Na figura 4.1 vemos um trecho de código e ao lado seu código 'plagiado'. Nesse caso, o nome das variáveis permanece o mesmo, mas ambos os `if`s tiveram seu bloco `else` trocado de ordem, com a devida modificação na condição do `if`.

Nas figura 4.2 vemos, primeiro, duas possíveis árvores sintáticas derivadas desses dois trechos de código. Podemos ver que, ao considerar expressões em detalhe e não efetuar a reordenação, as árvores ficam bastante distintas. De fato, se utilizássemos o ladrilhamento nessas árvores com um tamanho mínimo de casamento igual a 3 encontraríamos apenas três trechos idênticos:

- `if - comp - id - id`
- `atr - id - id`
- `atr - id - fun - id`

Já com uma ordenação temos trechos idênticos bem maiores:

- `if - comp - id - id`
- `id - atr - id - fun - id - atr - id - id - fun - id`

Vemos assim, a vantagem da ordenação

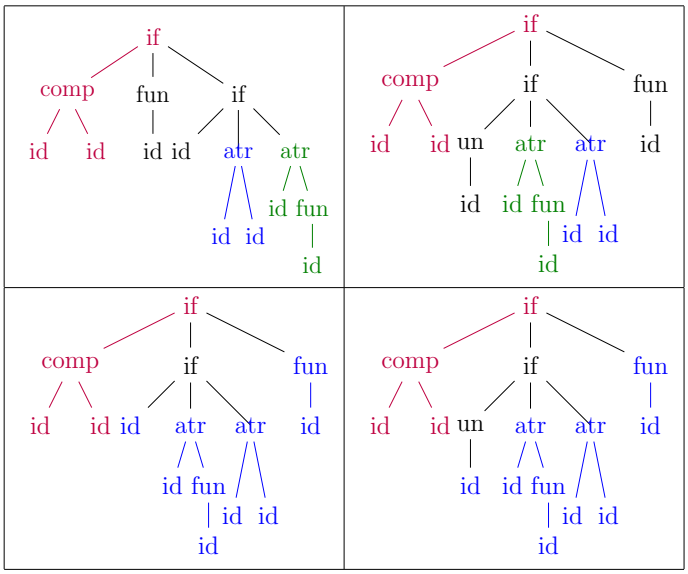


Figura 4.2: Árvores sintáticas sem e com reordenação

Já na figura 4.3 temos a mesma situação, porém agora ignoramos expressões propriamente ditas. Ou seja, consideramos uma expressão, não importa quantos operadores e operandos tenha como sendo um átomo só. Por exemplo, `a+b*c/d` e `a*b` seriam representados pelo mesmo átomo. Isso pode reduzir em muito o tamanho da representação e, dessa forma, se corre o risco de perder informações importantes, o que pode ocasionar num número maior de falsos positivos. Nesse caso, na árvore sem reordenação temos os seguintes trechos idênticos:

- `atr - id - exp`
- `atr - id - fun - exp`

Já com reordenação, podemos ver que as árvores são idênticas.

Algumas convenções foram adotadas para facilitar na normalização do programa, particularmente para que as diversas estruturas de repetição pudessem ser representadas de uma maneira coerente. Uma estrutura *do-while* e *while* tem a mesma representação, apenas mudando a posição da expressão condicional. Já um *for* tem uma normalização mais elaborada, conforme descrito na seção 5.2.

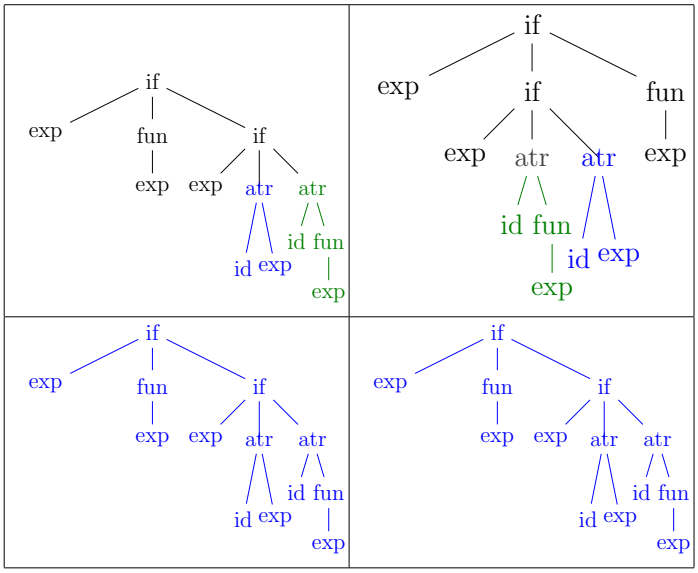


Figura 4.3: Árvores sintáticas, ignorando expressões, sem e com reordenação

Um aspecto que vale a pena comentar é que a ordenação, tal qual descrita acima e, particularmente a ordenação da figura 4.2, não garante que duas expressões com ordem trocada se tornem idênticas após a ordenação. Por exemplo, consideraremos a expressão `a+b+c`, onde `a`,`b` e `c` representam três expressões quaisquer que já foram ordenadas e têm valores lexicográficos 10, 20 e 30, respectivamente. Supondo que o símbolo '`+`' tem valor lexicográfico 25, teremos uma estrutura semelhante a:

```
  +
 + c
a b
```

Que uma vez ordenado, geraria `++abc`. Agora, se consideraremos a mesma expressão, mas com ordem trocada: `c+b+a`. Na árvore, isso se torna:

```
  +
 + a
c b
```

Isso geraria a cadeia `+a+bc`. Por isso, após uma troca de ordem nas expressões, mesmo após a ordenação, existe uma chance de que os programas comparados não serão idênticos. Semelhantemente, existe um problema com o uso de parênteses que forçam uma ordem de avaliação diferente e podem afetar a árvore sintática. Por exemplo, o analisador montaria duas árvores diferentes para '`a+b+c`' e '`a+(b+c)`'.

O tratamento de comandos tem um problema semelhante, quando se trata de comandos compostos (denotados por chaves: '{}'). O analisador sintático não dá valor especial a chaves, mas pela especificação da linguagem, cada conjunto de chaves desse tipo forma um *statement list*[1], e essas estruturas são adicionadas à árvore sintática. O problema ocorre quando o plagiador começa a inserir chaves supérfluas, por exemplo, substituindo o comando '`a=b+c;`' por '`{a=b+c;}`'. Apesar de que não há diferença entre o significado desses dois trechos, existe uma diferença sintática: no primeiro caso temos, de acordo com a especificação da linguagem, um *'assignment statement'*, e na segunda um *'compound statement'* com um *'assignment statement'* dentro.

Esse problema, portanto, é análogo ao das expressões, mas mais fácil de simplificar, já que existe menos flexibilidade quando se refere a comandos. A solução é 'achatar' a árvore sintática. Comandos compostos encontrados dentro de outros comandos compostos, e não associados aos comandos de controle de fluxo (`while, if, for`, etc.) são transformados em listas de comandos, e concatenados ao comando pai. Então, no exemplo anterior, o comando composto contendo apenas um comando de atribuição se torna um comando de atribuição.

O problema dos parênteses em expressões poderia ser resolvido de maneira semelhante, mas existem algumas complicações. Em particular, expressões aritméticas podem ter sinal. Então ao tratar uma expressão como '`a-(b+c)`' não seria suficiente simplesmente 'remover' os parênteses, como foi feito para comandos. No caso, teríamos que inverter o sinal da subexpressão '`b+c`'. Talvez fosse desejável, no caso de implementações futuras, realizar um tratamento mais completo de expressões.

[1] lista de comandos

Capítulo 5

Implementação e experimentos

5.1 Algoritmos de comparação

Um dos primeiros passos na implementação foi elaborar os algoritmos de comparação conhecidos em uma linguagem de *scripting*, de modo a facilitar a criação do ambiente de detecção de plágio. A idéia era construir um alicerce sobre o qual poderiam ser adicionadas novas linguagens, novos algoritmos ou novos mecanismos de exibição de resultados. Embora o foco tenha mudado para o pré-processamento, grande parte dessa implementação permanece.

Foram implementados vários algoritmos; o RKRGST como existe no JPlag, uma variante da peneiragem, o alinhamento local e uma variante do método de informação compartilhada.

A implementação dos algoritmos de detecção foi feita em Python, por facilidade de programação e pelo fato de que o *script* que integraria todos esses sistemas estava sendo escrito em Python também. Já o analisador sintático é uma combinação de Flex, Bison e linguagem C.

O problema da exibição

Um problema encontrado na primeira tentativa de implementação, na construção do ambiente, era o de exibição de resultados. Em particular, deseja-se exibição de resultados da maneira que o JPlag faz, com os programas mostrados lado a lado com os trechos plagiados marcados em cores diferentes.

Para isso, é necessário, além da taxa de semelhança, conseguir identificar quais são os trechos que são iguais. Isso é fácil para o GST, uma vez que o resultado do algoritmo é o conjunto de ladrilhos; ou seja, o início de cada um dos trechos idênticos e seu comprimento. Para a peneiragem isso já é um problema mais complexo; em parte porque o conjunto

de assinaturas gerado em teoria não respeita a ordem que os mesmos ocorrem no código original. Isso é intrínseco ao algoritmo, uma vez que se a ordem tivesse importância seria mais difícil encontrar plágios em casos onde a ordem de trechos são trocados. A falta de informação de ordem se torna um problema particularmente se uma mesma assinatura ocorre mais de uma vez. Pode ocorrer de, na exibição, uma assinatura não apontar para o trecho em que melhor se encaixa.

Além disso, existe o problema da discontinuidade; na peneiragem é escolhido apenas um subconjunto de assinaturas. Duas assinaturas adjacentes nesse subconjunto peneirado podem não representar blocos de texto que estavam adjacentes no programa original.

O sistema precisa conseguir identificar esses casos corretamente para efetuar uma exibição precisa. Apesar disso, o MOSS realiza exibição, e portanto o problema não é intratável. É possível que os autores tenham implementado algum algoritmo para determinar a cobertura das cadeias de assinaturas, algo semelhante ao GST, mas isso é especulação. As cadeias de assinaturas são consideravelmente menores do que os programas originais, e apenas programas que apresentam semelhança suficiente são processados, então a complexidade dessa comparação pode não ser problemática.

A informação compartilhada apresenta problemas maiores ainda. Como a métrica é definida apenas sobre o comprimento das cadeias comprimidas, não existe informação alguma sobre quais trechos de programa correspondem a quais outros, pelo menos no sistema como descrito por Chen et. al[6]. É bem possivel que o SID utilize, no seu algoritmo de compressão otimizado, um mecanismo para conseguir fazer essa correlação. De fato, a descrição de alto-nível do algoritmo de compressão sugere que isso é verdade.

Um outro problema com a exibição é conseguir correlacionar o conjunto de átomos do programa pré-processado, que em geral não se assemelha ao programa original com trechos de código do programa original. Isso é simples se foi feita apenas uma análise léxica para gerar a seqüência de átomos, pois é necessário apenas armazenar dados de posição ao mesmo tempo que se armazena um átomo. Com uma análise sintática mais complexa o problema se torna mais complicado. Em particular, na implementação feita, utilizando uma árvore generalizada para armazenar a árvore sintática seria necessário ampliar a estrutura de dados para armazenar informação de posição.

Dado o foco final do trabalho, foi implementada apenas a exibição da medida de semelhança.

Implementação do RKRGST

Os artigos de Prechelt et al.[17] e Wise[22] descrevem com bastante detalhe o algoritmo do GST e do RKRGST, embora Prechelt et al. não entram em detalhes quanto à implementação do RKRGST como ele existe no JPlag. A diferença entre o algoritmo utilizado

no YAP3 e o algoritmo utilizado no JPlag é que o JPlag usa um valor fixo para o comprimento das assinaturas. O preço é uma utilização maior de espaço, uma vez que é necessário ter um controle mais preciso de quais ladrilhos foram cobertos e quais não foram.

Ao sistema desenvolvido em Python foi dado o nome *pygst*.

Implementação da peneiragem

A implementação da peneiragem como método de detecção de plágio seguiu, em maior parte, o procedimento descrito no artigo do Schleimer et al[18]. A diferença ocorreu nos aspectos do sistema que não foram detalhados pelos autores. Em particular, como efetuar o casamento de conjuntos de assinaturas não está perfeitamente claro. Uma tentativa, e a primeira utilizada, foi efetuar uma intersecção dos conjuntos de assinaturas, e obter uma razão entre o tamanho do conjunto da intersecção e o conjunto original. Um problema dessa estratégia é que foram utilizadas estruturas de dados que ignoram dados repetidos; para um programa com muitos dados repetidos e para valores de assinatura pequenos isso pode gerar resultados imprecisos.

Essa estratégia difere do que se sabe do sistema de Schleimer et al.[18], que utiliza um índice invertido para obter um casamento, como visto na seção 2.4. Um outro problema da solução utilizada é na exibição dos resultados; enquanto que para o GST exibir os resultados semelhantes lado a lado é trivial a partir do resultado do algoritmo, na peneiragem apenas têm-se as várias assinaturas comuns. E, utilizando uma simples intersecção de conjuntos, mesmo se dados repetidos não fossem ignorados, ainda teríamos um problema: não há como saber qual assinatura corresponde a qual outra assinatura, e exibição de resultados pode não corresponder à realidade, particularmente dado um pré-processamento agressivo.

Ao sistema desenvolvido em Python foi dado o nome *pywinnow*.

Implementação da informação compartilhada

A implementação da informação compartilhada foi a mais problemática; no artigo de Chen et. al.[6] foi descrito brevemente um algoritmo de compressão bem-adaptado a problemas de plágio. Ao que parece, é uma variante do Lempel-Ziv 77[25], mas com uma janela de tamanho variável e mecanismos para processar cadeias semelhantes (diferentemente de cadeias idênticas, como faz o Lempel-Ziv canônico).

A primeira implementação de um sistema de compressão foi a do Lempel-Ziv 78, que não utiliza a janela deslizante do Lempel-Ziv 77, e sim um dicionário para efetuar a compressão. O algoritmo implementado era relativamente simples, sem preocupações com otimização. Foi nesse momento que se percebeu a dificuldade da exibição, e algumas das

fraquezas desse particular algoritmo. Em particular, o mecanismo de dicionário do LZ78 requer cadeias muito semelhantes para se obter um bom resultado. Dessa forma o LZ77 é superior para efetuar esse tipo de comparação. No entanto, uma vez que se determinou o problema da exibição decidiu-se utilizar bibliotecas de compressão já implementadas, como o *zlib* e *bz2*.

Um importante problema da implementação é que, provavelmente devido ao *overhead* do programa de compressão, a medida de semelhança entre dois programas idênticos não é igual a um, como deveria ser na teoria. Provavelmente porque aproximanos $Comp(x|y)$ por $Comp(yx) - Comp(y)$. Lembrando a seção 2.7:

$$d(x, y) \approx 1 - \frac{2Comp(x) - Comp(yx)}{Comp(xy)}$$

Porém, naquele caso, pudemos supor uma função de compressão ideal, cujo resultado para uma cadeia repetida seria extremamente próximo ao resultado da cadeia original. Esse não parece ser o caso com esses algoritmos de compressão e, assim, obtemos uma distância maior que zero e conseqüentemente um resultado diferente de um. Interessantemente, isso também está visível utilizando o program de compressão GenCompress, fornecido pelo grupo do SID, um programa que supostamente seria o melhor para efetuar compressão de programas com o objetivo de efetuar comparações.

Ao sistema desenvolvido em Python foi dado o nome *pyshared*.

Implementação da maior subseqüência comum

O algoritmo LCS, descrito na seção 2.3, foi implementado em Python, para se obter um *baseline* contra o qual comparar os outros algoritmos de detecção. Sua implementação segue o que foi descrito anteriormente, utilizando apenas dois vetores para se calcular o valor final, uma vez que não estamos interessados em mais do que o valor da maior subseqüência.

Para determinar o valor $d(x, y)$, utilizamos uma medida de distância análoga ao que o RKRGST utiliza:

$$d(x, y) = \frac{2 \cdot LCS(x, y)}{|x| + |y|}$$

5.2 Pré-processamento

O pré-processamento passou por várias fases de implementação. A primeira, e a mais ingênua, não fazia nada mais que uma análise léxica, traduzindo cada átomo do programa

em caractere ASCII, ignorando nomes de identificadores. Isso foi efetuado utilizando uma especificação da linguagem C[2, 3] para o gerador de analisadores léxicos Flex.

Uma primeira mudança na abordagem foi a geração de arquivos auxiliares que associavam cada caractere gerado pela análise léxica com uma posição no programa original. Isso foi feito com o intuito de facilitar a saída do processo de detecção, apontando trechos semelhantes entre programas, como é feito nos sistemas apresentados anteriormente.

Foi determinado, após essa fase, que seria necessário efetuar uma análise mais profunda do programa a ser analisado. Especificamente, seria necessário efetuar uma análise sintática sobre o programa a ser analisado. Isso gerou um novo problema: para efetuar a análise sintática seria necessário manter um catálogo dos tipos que não fazem parte do padrão da linguagem (os tipos definidos pelo usuário). Foi utilizada uma especificação para a linguagem C para o gerador de analisadores sintáticos Bison, com alterações que permitem armazenar os tipos encontrados em um arquivo, com o propósito de se fazer um pré-processamento das bibliotecas padrão. A geração da saída é alterada, utilizando recursos disponíveis na análise sintática

Infelizmente, viu-se que a informação obtida na análise sintática ingênua não era suficiente para prevenir certos casos de falsos positivos e falsos negativos. Decidiu-se manter muita informação sobre a estrutura do programa, e efetuar ordenação lexicográfica. Como essa ordenação poderia ser feita naturalmente na árvore sintática do programa, foram feitas modificações para geração da mesma. Embora referimos à estrutura de dados como uma árvore, ela é mais precisamente uma árvore generalizada. Em geral, cada registro contém apontadores para outros registros, representando os elementos de cada comando ou estrutura e um apontador para uma lista de registros, representando uma lista de comandos, onde apropriado.

Geração da árvore sintática

A árvore criada consiste de vários registros ligados. Cada registro contém, além do campo que representa qual o tipo de estrutura representada pelo registro (que pode ser uma função, um comando, uma expressão ou mesmo uma constante), uma lista de registros, que pode estar vazia, e quatro campos que podem apontar para os componentes de uma dada estrutura.

Como o Bison não constrói a árvore sintática, nem percorre as produções de forma recursiva descendente, foi necessário implementar uma pilha para manter o controle sobre quais produções foram executadas recentemente, e montar nós da árvore a partir dessas produções.

A estrutura da árvore em si é um tanto simplificada, composta de um mesmo registro que representa todas as estruturas de C que decidimos representar.

Achatamento

Como mencionado na seção 4.2, a presença de chaves desnecessárias pode poluir a árvore sintática, sem alterar o significado de um programa. Para consertar isso, fazemos um 'achatamento' da árvore sintática, como no algoritmo 4:

Algoritmo 4 Achatamento da árvore sintática

Entrada: *registro* da árvore sintática
Saída: *registro* achatado

```
Achata(registro)
listatemp ← {}
se registro.tipo é for, while, do-while, if ou switch então
    Achata registro.corpo
    se Tipo de registro é if e existe campo else então
        Achata registro.else
    fim se
senão se registro.tipo é de lista de comandos então
    para Cada elemento elem de registro.lista faça
        Achata elem
        se elem é uma lista de comandos então
            listatemp ← listatemp + elem.lista
        senão
            listatemp ← listatemp + elem
        fim se
    fim para
    registro.lista ← listatemp
fim se
```

Como a árvore sintática não contém ciclos, as chamadas terminam nas folhas da árvore. Cada nó é visitada uma vez só, então a complexidade desse algoritmo é linear no número de nós na árvore. O propósito desse algoritmo é normalizar a árvore e remover listas de comandos que não alteram o significado do programa. O algoritmo funciona da seguinte maneira: os campos de cada registro são varridos, recursivamente, dependendo do tipo de registro. Registros que são listas de comandos são varridos, e cada elemento da lista é achatado. Registros que representam estruturas de controle, como `for`, `while`, `if` ou `switch` têm suas listas de comandos varridas. Todos os outros tipos de registros são ignorados, por não conterem listas de comandos.

Na figura 5.1 vemos um trecho de código "poluído" com chaves supérfluas e o mesmo trecho achatado, com as chaves removidas.

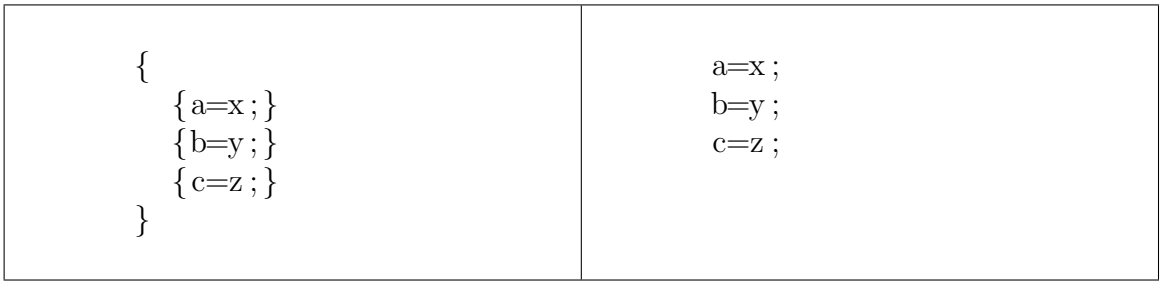


Figura 5.1: Trecho de código e achatamento do mesmo código

Ordenação

Todo nó da árvore tem um campo que contém a representação em forma de cadeia de caracteres daquele nó. A ordenação é feita de maneira recursiva, de baixo para cima, seguindo certas convenções:

- A cadeia que representa cada nó é formada pela concatenação dos seus filhos, ordenados lexicograficamente.
- Uma lista de expressões/comandos é representada pela concatenação do vetor de expressões pertencente ao nó.
- Estruturas de laço são representados de maneira uniforme:
  - Estruturas *while* e *do-while* são armazenadas com a condição de laço antes do corpo do laço.
  - Estruturas *for* têm as expressões de inicialização e de teste inseridas antes do corpo do laço, enquanto que a expressão de incremento é inserida após o corpo do laço.
- Uma expressão condicional (*if-else*) tem os corpos dos blocos inseridos após a expressão condicional, de acordo com sua ordem lexicográfica. Por exemplo: se o bloco do *else* vinha antes, lexicograficamente, do que o bloco *if*, o mesmo seria inserido antes deste.

Essa ordenação tem como propósito liberar a memória de elementos da árvore cujas cadeias já foram geradas e ordenar. Dessa maneira, após a geração da cadeia que representa toda a árvore, a árvore sintática é destruída, já que não é mais necessária.

O resultado final da ordenação é uma seqüência de átomos que representa o programa inteiro.

Capítulo 6

Comparação dos principais sistemas

Foram comparados o MOSS, Jplag, SID e os três sistemas implementados em Python, **pygst**, **pywinnow** e **pyshared**, de modo a determinar os melhores resultados. Como parte do caso de teste do **pygst**, do **pywinnow**, e do **pyshared** os metodos de pré-processamento foram alterados para melhor se adaptar ao problema.

6.1 Casos de teste

Foram adaptados dois conjuntos de testes, cada um resolvendo um problema diferente. Em cada um destes conjuntos, os primeiros oito testes foram modificados de maneira semelhante, a partir de um teste base, produzindo o que chamaremos de casos sintéticos:

1. caso base
2. cópia sem alterações
3. expressões modificadas, trocadas de ordem (por exemplo `d=a+b*c` por `d=b*c+a`) sem invalidar o programa
4. comandos substituídas por equivalentes (por exemplo `a=a+5` por `a+=5`) ou alteradas de ordem (trocando o corpo do `if` pelo corpo do `else`)
5. as mudanças de 3 e 4
6. repetir um trecho do programa em outro local, duas vezes; isso invalida o programa
7. expressões modificadas de qualquer maneira (por exemplo `c=a*b` por `c=63*a+b`); isso também invalida o programa
8. chaves supérfluas inseridas em torno de todo comando

Vale ressaltar que mudanças extremamente triviais, como mudança de nomes de varíaveis, alteração de comentário ou inserção de espaços em branco foram ignoradas, uma vez que o pré-processamento tem como ignorar todos esses fatores. No caso 3 as mudanças, em geral, foram do tipo que se aproveitam da comutatividade/associatividade da linguagem. Isso também envolvia aproveitar construções sintáticas da linguagem, como por exemplo expandir a expressão de autoincremento para uma operação de soma. No caso 4 as mudanças feitas envolviam, por exemplo, substituir um comando *for* por um comando *while*. Ou então trocar a ordem de um comando *if*, testando primeiro a negação da expressão anterior. Nos casos 6 e 7 é importante notar que os programas foram invalidados. No caso 7 as expressões simplesmente foram alteradas por outras, sem relação às originais.

O primeiro conjunto, 'pip' (Apêndice B.1), foi derivado de uma solução de um problema proposto numa das competições de programação. O teste base tem 95 linhas de código em C. Além dos oito variantes sintéticos descritos acima, foram acrescentados ao conjunto cinco programas que resultaram de um concurso de plágio convocado para esta finalidade (testes 9 a 13). O caso 9 foi o vencedor do concurso.

O segundo conjunto, 'huffman' (Apêndice B.2), é a solução do problema de compressão pelo método de Huffman proposto numa disciplina de programação. O caso base tem 393 linhas. Além dos variantes sintéticos, foram acrescentados cinco programas submetidos pelos alunos, escolhidos aleatoriamente. Nestes cinco casos não havia plágio.

A seguir vamos exibir os resultados obtidos, de maneira tabular. Temos duas situações de teste. Na primeira comparamos os diferentes mecanismos de pré-processamento, e seu efeito nos resultados. Para isso separamos os dados de acordo com sistema de detecção (GST, peneiragem etc.) e por programa. Na outra situação tentamos determinar, dado um certo pré-processamento, qual sistema apresenta melhores resultados. Para tanto, esses casos estão divididos por tipo de pré-processamento e por programa. Todos os programas exibidos estão sendo comparados contra o programa 1, o caso base. Para assinalar quais argumentos foram passados no pré-processamento, utilizaremos a seguinte chave:

1. **N**: sem ordenação
2. **I**: ignorando expressões, levando em conta apenas comandos
3. **NI**: ignorando expressões, sem ordenação

Para nossos casos de teste, acreditamos ser mais interessante efetuar a comparação entre os diferentes mecanismos de pré-processamento, por isso daremos mais foco a essa discussão. Além disso, para melhor situar nossos resultados, cada tabela terá também os resultados do MOSS e do JPlag para efeito de comparação. Vale lembrar, no entanto, que os valores não estão normalizados. Então uma comparação apenas numérica talvez não

seja a mais precisa; isso é especialmente importante quando avaliando os resultados da informação compartilhada. Como mencionado na seção 5.1, mesmo para casos idênticos, a informação compartilhada não obteve resultado igual a um.

Portanto, quando comparando um sistema contra o JPlag, por exemplo, provavelmente é mais útil ver como os dois se comportam diante dos diversos testes. Em particular, ver como um sistema se comporta com os plágios evidentes e com os casos que não são cópias.

Nas tabelas a seguir, o nome dos sistemas de detecção é auto-explicativo, cada um baseado nos sistemas **pygst**, **pywinnow** e **pyshared**, ou do **MOSS** e **JPlag**. No caso do **pywinnow** e do **pygst**, existe uma outra informação: o tamanho do casamento mínimo. Então, por exemplo, **G-05-I** significa que o programa foi comparado utilizando o **pygst** com tamanho de casamento mínimo igual a 5 e ignorando expressões. Outra informação adicional é no caso do **pyshared**. Como foram utilizadas duas funções de compressão, o *GenCompress* e o *zlib*, existe uma distinção feita nessas tabelas. As colunas assinaladas por **SG** utilizam *GenCompress* enquanto que as colunas assinaladas simplesmente por **S** utilizam *zlib*. Para os resultados do JPlag (o MOSS não permite que se ajuste o tamanho da janela ou k-grama) utilizamos um tamanho de casamento mínimo igual a cinco. Percebemos que isso nos fornecia os melhores valores.

Nas colunas do MOSS e do JPlag, os valores marcados com um '—' são pequenos demais para serem exibidos pelo MOSS ou pelo JPlag.

6.2 Tamanho de casamento mínimo

Nos algoritmos GST e na peneiragem, é possível ajustar o tamanho do comprimento mínimo a ser considerado. Nos resultados a seguir focaremos em tamanho de comprimento mínimo 5. Em geral, nos casos com tamanho de comprimento 10, vemos a diferença entre casos de plágio e de não plágio aumentar, relativamente. Vemos também que o valor numérico do resultado diminui. Na peneiragem vemos uma perturbação maior, já que a peneiragem é mais sensível a pequenas mudanças que os outros algoritmos, como veremos mais adiante. Já que acrescentam pouco à discussão, e por aumentarem o tamanho das tabelas a seguir, estamos deixando os resultados de comprimento 10 para o apêndice A.

Vale notar que, apesar de que o JPlag aceita ajustar o valor de semelhança, o MOSS não permite o mesmo. Também, no LCS e na informação compartilhada, o tamanho mínimo de comprimento não tem o mesmo significado. No LCS, o casamento é feito caractere a caractere, enquanto que, na informação compartilhada, tentamos extrair significado dos dados utilizando programas de compressão. É bem provável que o tamanho mínimo de comprimento tenha influência no próprio algoritmo de compressão, mas como estamos utilizando algoritmos já prontos (gzip e GenCompress) não podemos manipular esse parâmetro.

6.3 Resultados utilizando LCS

Para efeitos de comparação, utilizaremos o LCS como caso base para as tabelas a seguir. Isso porque o LCS é um algoritmo bastante conhecido, e também porque é bastante utilizado. É o algoritmo por trás do `diff`, por exemplo, e talvez o método mais ingênuo de se fazer detecção de plágio seja utilizando-o. Portanto, vamos tentar comparar a qualidade dos resultados do LCS com aqueles de outros sistemas.

Primeiro, porém, analisaremos o efeito dos diversos pré-processamentos no resultado do LCS. Na tabela 6.1, vemos os resultados no primeiro conjunto de testes, e na tabela 6.2, vemos os resultados no segundo.

Percebemos, de imediato, resultados inesperados, que vão parecer ainda mais inesperados quando comparados com os resultados dos outros sistemas de plágio. Talvez o resultado mais imprevisto venha dos casos 4 e 5 na tabela 6.2. Aí vemos um resultado mais próximo para o caso sem ordenação do que para o caso com ordenação. É difícil explicar esse resultado, mas talvez para iluminar a situação possamos olhar os casos 9 a 13. Esses são submissões de alunos em uma disciplina de programação, e não apresentam plágio. Apesar disso, no caso **sem** ordenação vemos um resultado numericamente maior do que no caso com ordenação. Isso indica talvez que o LCS não seja um sistema inteiramente adequado para medir semelhança de programas; mesmo com ordenação, os resultados são imprevisíveis.

Essa imprevisibilidade talvez possa ser confirmada na tabela 6.1, onde o caso 5, que no conjunto anterior resultava em valor de semelhança maior para o caso sem ordenação, agora apresenta valor de semelhança maior no caso **com** ordenação.

Porém, na primeira coluna, em que vemos resultados quando se ignora expressões, os valores retornados são talvez menos surpreendentes, embora pareçam uniformemente altos. Nessa coluna, a semelhança entre os diversos programas é muito alta, sem dúvida devido à quantidade de informação 'perdida' quando expressões são ignoradas.

Outro resultado interessante é o do caso 9 na tabela 6.1. Esse foi um dos resultados do concurso de plágio, o que obteve menor índice de semelhança. E de fato, o LCS, nas três opções de pré-processamento não parece conseguir lidar com esse caso de teste. Como veremos adiante, os outros programas de detecção também tiveram problemas com esse caso, mas não na mesma proporção.

Especulamos que os resultados altos do LCS sobre os programas não ordenados se devem ao fato de o algoritmo LCS ser bastante sensível à reordenação. Pequenas mudanças espalhadas nas cadeias que não alteram sua estrutura geral terão menos impacto do que uma reordenação, especialmente quando considerando os problemas com a reordenação de expressões mencionados na seção 4.2.

	PIP	PIP-I	PIP-N
01	1.00	1.00	1.00
02	1.00	1.00	1.00
03	0.92	1.00	0.92
04	0.83	0.96	0.87
05	0.83	0.96	0.80
06	0.90	0.93	0.90
07	0.79	1.00	0.84
08	1.00	1.00	1.00
09	0.68	0.70	0.53
10	0.77	0.82	0.56
11	0.88	0.96	0.83
12	0.98	1.00	0.88
13	0.88	0.95	0.86

Tabela 6.1: Resultados do LCS com e sem reordenação sobre do conjunto 'pip'.

	HUFFMAN	HUFFMAN-I	HUFFMAN-N
01	1.00	1.00	1.00
02	1.00	1.00	1.00
03	1.00	1.00	0.99
04	0.89	0.99	0.97
05	0.88	0.99	0.96
06	0.90	0.90	0.91
07	0.97	1.00	0.98
08	1.00	1.00	1.00
09	0.64	0.83	0.74
10	0.63	0.88	0.74
11	0.68	0.87	0.74
12	0.67	0.83	0.76
13	0.66	0.80	0.71

Tabela 6.2: Resultados do LCS com e sem reordenação sobre o conjunto 'huffman'.

6.4 Resultados do conjunto 1 ('pip'), classificados por algoritmo de comparação

Na tabela 6.3 temos o primeiro conjunto de resultados. Primeiro, vale notar que para todas as tabelas, as linhas 1 e 2 deveriam ser sempre iguais: a primeira indica a comparação de um programa com ele mesmo (algo que o JPlag e MOSS não fazem); a segunda indica a comparação com uma cópia idêntica do programa original.

Considerando o processamento padrão, da primeira coluna, obtemos resultados bastante bons. Em particular, mesmo sem ignorar expressões, esse processamento consegue não ser enganado pela troca de ordem feita no caso 3 (tendo em mente a limitação descrita na seção 4.2). Isso valida o uso da ordenação, uma vez que uma comparação com a terceira coluna, em que não é feita ordenação, apresenta valores bastante mais baixos. E, como é de se esperar, na segunda coluna, em que se vê os resultados obtidos ignorando expressões, o valor de semelhança é igual a 1, que significa programas idênticos. O Jplag apresenta resultados semelhantes, uma vez que suspeitamos que o mesmo ignora expressões. Ele é visto na quarta coluna; mesmo sem ordenação, o fato de que expressões são ignoradas tornam os programas idênticos. Já o MOSS não parece ignorar expressões. Seus resultados nesse caso são decepcionantes, o que provavelmente seria um falso negativo.

O caso 4 é outro em que vemos resultados bons do *pygst*. O caso padrão apresenta o melhor resultado. O fato de que o valor é menor do que o anterior indica que nem todas as alterações feitas foram consideradas pelo pré-processamento. O interessante é que o caso que ignora expressões obteve valores um pouco mais baixos do que o padrão. Isso é porque as mudanças inseridas foram, em geral, novas expressões. Essas novas expressões são, em grande parte, tratadas pelo sistema de normalização e pré-processamento. Mas como essa opção de pré-processamento explicitamente ignora expressões, as mesmas não puderam ser tratadas, causando maior divergência. Nesse caso, apesar das mudanças serem bastante simples, o JPlag simplesmente não consegue lidar com elas, e nem o MOSS.

O caso 5 é uma amálgama das mudanças inseridas no caso 3 e 4, e os resultados coincidem com isso. Nem o MOSS nem o JPlag parecem conseguir lidar com essas alterações.

No caso sete já temos um resultado mais positivo. Esse apresenta o menor índice de semelhança entre todos os casos sintéticos. E, como esse caso não passa de nada além de mudanças nas expressões, não é grande surpresa que o JPlag e os casos **I** identifiquem esse programa como idêntico ao original. A única surpresa é o MOSS, que parece reagir bastante fortemente a mudanças de expressões, apresentando o mesmo valor de semelhança que o caso 3.

Como o último caso sintético, temos o caso 8. O JPlag não se comporta muito bem quando exposto a tantas chaves, obtendo um resultado muito baixo. Já o MOSS e todos os outros pré-processamentos ignoram chaves supérfluas, e indicam programas quase

idênticos.

Finalmente, temos os casos de 9 a 13. Nesses, o caso padrão obtém resultados bastante bons. Com exceção do caso 9, que parece enganar todos os sistemas (que inclusive foi o vencedor do concurso de plágio), todos os programas parecem apresentar níveis de semelhança altos. No caso 11, em que houve tentativa de plágio através de manipulação de expressões e certas estruturas de controle, pode-se ver resultados análogos semelhantes aos dos casos 3, 4 e 5. Particularmente, o caso **I** e o JPlag conseguem valores de semelhança particularmente altos.

Além disso, podemos concluir que a primeira coluna, (GST-05) parece produzir resultados mais consistentes para todos os casos.

	GST-05	GST-05-I	GST-05-N	GST-05-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	0.98
03	0.98	1.00	0.76	1.00	0.92	1.00	0.23
04	0.94	0.91	0.84	0.81	0.83	0.50	0.38
05	0.93	0.91	0.70	0.81	0.83	0.25	—
06	0.90	0.93	0.90	0.93	0.90	0.97	0.98
07	0.72	1.00	0.69	1.00	0.79	1.00	0.23
08	1.00	1.00	1.00	1.00	1.00	0.35	0.98
09	0.79	0.44	0.77	0.33	0.68	0.25	0.27
10	0.87	0.80	0.79	0.77	0.77	0.81	0.69
11	0.82	0.96	0.80	0.96	0.88	0.88	0.29
12	0.99	1.00	0.94	0.90	0.98	0.81	0.51
13	0.88	0.86	0.85	0.83	0.88	0.65	—

Tabela 6.3: Resultados GST sobre o conjunto de programas 'pip'.

	W-05	W-05-I	W-05-N	W-05-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	0.98
03	0.86	1.00	0.63	1.00	0.92	1.00	0.23
04	0.73	1.00	0.79	0.75	0.83	0.50	0.38
05	0.73	1.00	0.48	0.75	0.83	0.25	—
06	0.97	1.00	1.00	1.00	0.90	0.97	0.98
07	0.62	1.00	0.58	1.00	0.79	1.00	0.23
08	1.00	1.00	1.00	1.00	1.00	0.35	0.98
09	0.46	0.15	0.52	0.00	0.68	0.25	0.27
10	0.64	0.21	0.66	0.46	0.77	0.81	0.69
11	0.77	0.75	0.61	0.78	0.88	0.88	0.29
12	0.91	1.00	0.86	0.90	0.98	0.81	0.51
13	0.68	0.90	0.65	0.82	0.88	0.65	—

Tabela 6.4: Resultados da peneiragem sobre o conjunto de programas 'pip'.

A tabela 6.4 apresenta resultados da peneiragem. Em geral, a peneiragem não parece se comportar tão bem quanto o GST. Em particular, no caso 4 e 5 os índices de semelhança são bastante baixos, ao menos quando comparados com os resultados anteriores. Interessantemente, nesses casos os resultados ainda parecem melhores do que os resultados obtidos pelo JPlag e MOSS.

No caso 3, vemos algo interessante sobre a peneiragem. Ela parece ser muito sensível a pequenas mudanças, sendo que esse é o caso de teste mais semelhante que temos sem ser absolutamente idêntico. Isso parece se confirmar no comportamento do MOSS.

Fora isso, o mais interessante vem dos casos **I**. Nesses casos, os resultados parecem um tanto estranhos. É possível que haja um 'comprimento crítico', um número mínimo de átomos para que se possa fazer a peneiragem, dados um tamanho de janela e comprimento de k-grama.

	S	S-I	S-N	S-NI	SG	SG-I	SG-N	SG-NI	LCS	Jplag	MOSS
01	0.89	0.78	0.89	0.76	0.94	0.79	0.96	0.70	1.00	—	—
02	0.89	0.78	0.89	0.76	0.94	0.79	0.96	0.70	1.00	1.00	0.98
03	0.66	0.78	0.41	0.76	0.74	0.79	0.44	0.70	0.92	1.00	0.23
04	0.56	0.58	0.49	0.57	0.51	0.46	0.44	0.59	0.83	0.50	0.38
05	0.51	0.58	0.29	0.57	0.45	0.46	0.25	0.59	0.83	0.25	—
06	0.81	0.66	0.86	0.59	0.79	0.57	0.88	0.48	0.90	0.97	0.98
07	0.38	0.78	0.38	0.76	0.31	0.79	0.34	0.70	0.79	1.00	0.23
08	0.89	0.78	0.89	0.76	0.94	0.79	0.96	0.70	1.00	0.35	0.98
09	0.42	0.25	0.41	0.27	0.30	0.20	0.33	0.20	0.68	0.25	0.27
10	0.48	0.32	0.47	0.38	0.40	0.29	0.48	0.30	0.77	0.81	0.69
11	0.51	0.47	0.39	0.52	0.51	0.43	0.30	0.39	0.88	0.88	0.29
12	0.83	0.78	0.67	0.52	0.80	0.79	0.63	0.55	0.98	0.81	0.51
13	0.51	0.49	0.43	0.45	0.49	0.43	0.34	0.33	0.88	0.65	—

Tabela 6.5: Resultados da informação compartilhada sobre o conjunto de programas 'pip'.

Como a última tabela desse caso de testes, temos a tabela 6.5 relativa à informação compartilhada. O mais interessante é que, mesmo para casos idênticos, o valor da semelhança não é igual a um. É interessante também ver que o valor da 'maior' semelhança também varia de caso para caso. Por exemplo, para o caso padrão, utilizando gzip, o valor é 0.89, enquanto que o caso padrão, utilizando *GenCompress* tem valor de semelhança 0.94.

Com isso em vista, podemos ver que pequenas diferenças parecem ter impacto bastante grande, maior ainda do que na peneiragem, uma vez que o caso 3 tem um índice de semelhança baixo em relação ao valor dos pares idênticos, lembrando que no caso 3 a única diferença é a ordem de certas expressões. O caso *I*, que ignora essas expressões completamente, apresenta índice de semelhança igual ao do caso idêntico, como é de se esperar. O caso 8, que é o de acréscimo de chaves supérfluas também se apresenta como idêntico, uma vez que o pré-processamento elimina essa diferença em todos os casos.

Embora seja difícil analisar os índices de semelhança nessa tabela, é provavelmente seguro dizer que os valores não parecem tão bons quanto os do GST. Além disso, apesar de que estamos utilizando um algoritmo de compressão desenvolvido especialmente para isso vemos alguns detalhes estranhos. O maior, obviamente, é que programas idênticos não tem valor de semelhança igual a um, como é o caso no GST. Porém, em geral os sistemas se comportam semelhantemente à peneiragem.

	GST-05	GST-05-I	GST-05-N	GST-05-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	0.97
03	1.00	1.00	0.97	1.00	1.00	1.00	0.77
04	0.94	0.99	0.94	0.93	0.89	0.94	0.88
05	0.94	0.99	0.90	0.93	0.88	0.54	0.62
06	0.90	0.89	0.90	0.90	0.90	0.91	0.96
07	0.93	1.00	0.97	1.00	0.97	1.00	0.77
08	1.00	1.00	1.00	1.00	1.00	0.46	0.97
09	0.66	0.80	0.69	0.56	0.64	0.61	0.27
10	0.69	0.76	0.68	0.60	0.63	0.65	0.29
11	0.69	0.76	0.73	0.62	0.68	—	0.28
12	0.73	0.69	0.70	0.67	0.67	0.55	0.26
13	0.67	0.74	0.66	0.57	0.66	0.59	0.26

Tabela 6.6: Resultados GST sobre o conjunto de programas 'huffman'.

6.5 Resultados do conjunto 2 ('huffman'), classificados por algoritmo de comparação

Na tabela 6.6 vemos os resultados do GST aplicado ao segundo conjunto de teste, o 'huffman'. De forma geral os resultados do caso padrão são excelentes, e semelhantes ao do caso de teste anterior. Não existe grande surpresa entre os resultados dos casos 1 a 8.

É mais interessante reparar nos casos 9 a 13, em que vemos verdadeiras submissões sem plágio. Ambos os JPlag e MOSS identificam esses casos corretamente, apresentando valores de semelhança relativamente baixos. Os nosses testes confirmam os mesmos resultados.

	W-05	W-05-I	W-05-N	W-05-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	0.97
03	0.94	1.00	0.92	1.00	1.00	1.00	0.77
04	0.90	0.91	0.88	0.88	0.89	0.94	0.88
05	0.87	0.91	0.79	0.88	0.88	0.54	0.62
06	0.97	0.85	0.96	0.94	0.90	0.91	0.96
07	0.81	1.00	0.88	1.00	0.97	1.00	0.77
08	1.00	1.00	1.00	1.00	1.00	0.46	0.97
09	0.62	0.51	0.60	0.58	0.64	0.61	0.27
10	0.54	0.69	0.57	0.80	0.63	0.65	0.29
11	0.66	0.64	0.63	0.59	0.68	—	0.28
12	0.62	0.67	0.59	0.74	0.67	0.55	0.26
13	0.50	0.61	0.48	0.60	0.66	0.59	0.26

Tabela 6.7: Resultados da peneiragem sobre o conjunto de programas 'huffman'.

Na tabela 6.7 novamente não há grandes surpresas. O mais interessante são, novamente, os casos **I**. Aqui vemos resultados com menos disparidade nesses casos, talvez porque o programa 'huffman' é um pouco mais complexo que o 'pip'.

Uma fraqueza da estratégia de ignorar expressões sem efetuar ordenação é que muita informação é perdida. Dessa forma, obtemos resultados surpreendemente altos para o caso 10 na coluna 4 (W-05-NI), que claramente não é plágio. Comparados aos outros resultados, e aos outros casos na mesma coluna, o valor (0.80) é alto. É tão alto que talvez seja mais próximo aos casos de plágio, especificamente aos casos 4 e 5 da mesma coluna. Esses dois casos são mais próximos ao 10 do que aos casos 9, 11 ou 13.

	S	S-I	S-N	S-NI	SG	SG-I	SG-N	SG-NI	LCS	Jplag	MOSS
01	0.92	0.86	0.92	0.83	0.98	0.79	0.97	0.79	1.00	—	—
02	0.92	0.86	0.92	0.83	0.98	0.79	0.97	0.79	1.00	1.00	0.97
03	0.83	0.86	0.76	0.83	0.91	0.79	0.80	0.79	1.00	1.00	0.77
04	0.69	0.58	0.76	0.62	0.65	0.65	0.70	0.67	0.89	0.94	0.88
05	0.66	0.58	0.59	0.62	0.63	0.65	0.57	0.67	0.88	0.54	0.62
06	0.85	0.69	0.87	0.75	0.87	0.61	0.90	0.70	0.90	0.91	0.96
07	0.63	0.86	0.65	0.83	0.68	0.79	0.73	0.79	0.97	1.00	0.77
08	0.92	0.86	0.92	0.83	0.98	0.79	0.97	0.79	1.00	0.46	0.97
09	0.30	0.33	0.32	0.35	0.29	0.26	0.32	0.18	0.64	0.61	0.27
10	0.32	0.34	0.31	0.37	0.32	0.18	0.33	0.24	0.63	0.65	0.29
11	0.32	0.38	0.33	0.34	0.30	0.22	0.31	0.20	0.68	—	0.28
12	0.31	0.36	0.34	0.37	0.29	0.26	0.32	0.23	0.67	0.55	0.26
13	0.26	0.31	0.26	0.33	0.26	0.29	0.25	0.18	0.66	0.59	0.26

Tabela 6.8: Resultados da informação compartilhada sobre o conjunto de programas 'huffman'.

Finalmente, na tabela 6.8 vemos os resultados da informação compartilhada. Novamente, vemos que o valor dos programas idênticos não é igual a 1. Porém, o valor da semelhança é melhor que o do 'pip'. Talvez essa métrica de semelhança tenha comportamento melhor quando temos programas maiores.

Vale também notar que, nesse conjunto e no anterior, o gzip parece ter comportamento inferior ao do GenCompress para efeitos de comparação. Mesmo assim, com um algoritmo desenvolvido justamente para esse propósito, o GenCompress, vemos que os resultados da informação compartilhada não são muito satisfatórios.

	GST-05	S	SG	W-05	LCS	Jplag	MOSS
01	1.00	0.89	0.94	1.00	1.00	—	—
02	1.00	0.89	0.94	1.00	1.00	1.00	0.98
03	0.98	0.66	0.74	0.86	0.92	1.00	0.23
04	0.94	0.56	0.51	0.73	0.83	0.50	0.38
05	0.93	0.51	0.45	0.73	0.83	0.25	—
06	0.90	0.81	0.79	0.97	0.90	0.97	0.98
07	0.72	0.38	0.31	0.62	0.79	1.00	0.23
08	1.00	0.89	0.94	1.00	1.00	0.35	0.98
09	0.79	0.42	0.30	0.46	0.68	0.25	0.27
10	0.87	0.48	0.40	0.64	0.77	0.81	0.69
11	0.82	0.51	0.51	0.77	0.88	0.88	0.29
12	0.99	0.83	0.80	0.91	0.98	0.81	0.51
13	0.88	0.51	0.49	0.68	0.88	0.65	—

Tabela 6.9: Comparação dos diversos resultados com reordenação sobre o conjunto 'pip'

6.6 Resultados do conjunto 1 ('pip'), classificados por pré-processamento

Nessa seção, e na seguinte, compararemos os diferentes algoritmos lado a lado. Em parte, essa análise foi feita nas duas seções anteriores, mas tentaremos dar mais foco aqui. Também daremos menos importância aos resultados do JPlag e do MOSS.

Na tabela 6.9, podemos efetuar a comparação entre os diversos sistemas diretamente, todos utilizando o pré-processamento padrão. É fácil ver que o GST consegue resultados mais consistentes. Em todos os casos de plágio, o GST tem valores superiores aos outros sistemas e, em um dos casos invalidados (o caso 7), o GST apresenta valores baixos o suficiente para distinguir entre plágio e não-plágio. Já o outro caso invalidado, o caso 6, é corretamente identificado como plágio por todos os sistemas.

	GST-05-N	S-N	SG-N	W-05-N	LCS	Jplag	MOSS
01	1.00	0.89	0.96	1.00	1.00	—	—
02	1.00	0.89	0.96	1.00	1.00	1.00	0.98
03	0.76	0.41	0.44	0.63	0.92	1.00	0.23
04	0.84	0.49	0.44	0.79	0.83	0.50	0.38
05	0.70	0.29	0.25	0.48	0.83	0.25	—
06	0.90	0.86	0.88	1.00	0.90	0.97	0.98
07	0.69	0.38	0.34	0.58	0.79	1.00	0.23
08	1.00	0.89	0.96	1.00	1.00	0.35	0.98
09	0.77	0.41	0.33	0.52	0.68	0.25	0.27
10	0.79	0.47	0.48	0.66	0.77	0.81	0.69
11	0.80	0.39	0.30	0.61	0.88	0.88	0.29
12	0.94	0.67	0.63	0.86	0.98	0.81	0.51
13	0.85	0.43	0.34	0.65	0.88	0.65	—

Tabela 6.10: Resultados sem reordenação sobre o conjunto de programas 'pip'.

Na tabela 6.10, que mostra resultados sem ordenação, vemos resultados semelhantes aos da tabela anterior. Novamente, o GST parece obter os melhores valores, com a informação compartilhada tendo os valores menos confiáveis. Já na tabela 6.11 temos resultados um pouco mais interessantes, uma vez que à primeira vista os resultados da peneiragem parecem ser melhores, em geral, do que o GST. Vale ressaltar que os valores para a peneiragem tomam pulos muito grandes, especialmente comparados aos outros algoritmos, isso talvez por causa das limitações da peneiragem quando lidando com poucos átomos, como foi mencionado anteriormente.

	GST-05-I	S-I	SG-I	W-05-I	LCS	Jplag	MOSS
01	1.00	0.78	0.79	1.00	1.00	—	—
02	1.00	0.78	0.79	1.00	1.00	1.00	0.98
03	1.00	0.78	0.79	1.00	0.92	1.00	0.23
04	0.91	0.58	0.46	1.00	0.83	0.50	0.38
05	0.91	0.58	0.46	1.00	0.83	0.25	—
06	0.93	0.66	0.57	1.00	0.90	0.97	0.98
07	1.00	0.78	0.79	1.00	0.79	1.00	0.23
08	1.00	0.78	0.79	1.00	1.00	0.35	0.98
09	0.44	0.25	0.20	0.15	0.68	0.25	0.27
10	0.80	0.32	0.29	0.21	0.77	0.81	0.69
11	0.96	0.47	0.43	0.75	0.88	0.88	0.29
12	1.00	0.78	0.79	1.00	0.98	0.81	0.51
13	0.86	0.49	0.43	0.90	0.88	0.65	—

Tabela 6.11: Resultados ignorando expressões sobre o conjunto de programas 'pip'.

	GST-05-NI	S-NI	SG-NI	W-05-NI	LCS	Jplag	MOSS
01	1.00	0.76	0.70	1.00	1.00	—	—
02	1.00	0.76	0.70	1.00	1.00	1.00	0.98
03	1.00	0.76	0.70	1.00	0.92	1.00	0.23
04	0.81	0.57	0.59	0.75	0.83	0.50	0.38
05	0.81	0.57	0.59	0.75	0.83	0.25	—
06	0.93	0.59	0.48	1.00	0.90	0.97	0.98
07	1.00	0.76	0.70	1.00	0.79	1.00	0.23
08	1.00	0.76	0.70	1.00	1.00	0.35	0.98
09	0.33	0.27	0.20	0.00	0.68	0.25	0.27
10	0.77	0.38	0.30	0.46	0.77	0.81	0.69
11	0.96	0.52	0.39	0.78	0.88	0.88	0.29
12	0.90	0.52	0.55	0.90	0.98	0.81	0.51
13	0.83	0.45	0.33	0.82	0.88	0.65	—

Tabela 6.12: Resultados sem reordenação e ignorando expressões sobre o conjunto de programas 'pip'.

Na tabela 6.12 vemos os resultados sem ordenação e ignorando expressões, sendo provavelmente a classe de pré-processamento menos interessante, uma vez que ignora o trabalho feito até então. Também é a tabela que parece fornecer os resultados mais inconsistentes. Isto não é de todo estranho, uma vez que os resultados dessa tabela talvez também sejam os menos interessantes, pois provavelmente representam o que o JPlag faz na sua análise. Entretanto, é difícil comparar o JPlag ao GST diretamente, já que mesmo ignorando expressões e sem reordenação, existem passos tomados no pré-processamento que o JPlag não toma, como o achatamento de comandos compostos e normalização de estruturas de laço (ver seção 4.2). Outra razão para essa falta de interesse decorre de que, ao ignorar expressões, perdemos informação demais. Isso é mais evidente no caso 7, em que temos mudanças de expressões, mas que o sistema acredita ser idêntico ao programa original.

Novamente deve-se frisar que programas maiores provavelmente produziriam resultados mais coerentes. Isso é algo a se pensar para trabalhos futuros.

	GST-05	S	SG	W-05	LCS	Jplag	MOSS
01	1.00	0.92	0.98	1.00	1.00	—	—
02	1.00	0.92	0.98	1.00	1.00	1.00	0.97
03	1.00	0.83	0.91	0.94	1.00	1.00	0.77
04	0.94	0.69	0.65	0.90	0.89	0.94	0.88
05	0.94	0.66	0.63	0.87	0.88	0.54	0.62
06	0.90	0.85	0.87	0.97	0.90	0.91	0.96
07	0.93	0.63	0.68	0.81	0.97	1.00	0.77
08	1.00	0.92	0.98	1.00	1.00	0.46	0.97
09	0.66	0.30	0.29	0.62	0.64	0.61	0.27
10	0.69	0.32	0.32	0.54	0.63	0.65	0.29
11	0.69	0.32	0.30	0.66	0.68	—	0.28
12	0.73	0.31	0.29	0.62	0.67	0.55	0.26
13	0.67	0.26	0.26	0.50	0.66	0.59	0.26

Tabela 6.13: Resultados dos diversos sistemas sobre o conjunto de programas 'huffman'.

6.7 Resultados do conjunto 2 ('huffman'), classificados por pré-processamento

Como na primeira tabela da seção anterior, a tabela 6.13 não traz grandes surpresas. Novamente podemos concluir que o GST oferece os melhores resultados para essa situação, confirmando resultados anteriores. Em particular, a informação compartilhada tem resultados bastante ruins. Ela consegue identificar os casos sem plágio, mas pequenas perturbações no código (casos 4, 5, 7) geram grandes alterações na distância computada.

	GST-05-N	S-N	SG-N	W-05-N	LCS	Jplag	MOSS
01	1.00	0.92	0.97	1.00	1.00	—	—
02	1.00	0.92	0.97	1.00	1.00	1.00	0.97
03	0.97	0.76	0.80	0.92	1.00	1.00	0.77
04	0.94	0.76	0.70	0.88	0.89	0.94	0.88
05	0.90	0.59	0.57	0.79	0.88	0.54	0.62
06	0.90	0.87	0.90	0.96	0.90	0.91	0.96
07	0.97	0.65	0.73	0.88	0.97	1.00	0.77
08	1.00	0.92	0.97	1.00	1.00	0.46	0.97
09	0.69	0.32	0.32	0.60	0.64	0.61	0.27
10	0.68	0.31	0.33	0.57	0.63	0.65	0.29
11	0.73	0.33	0.31	0.63	0.68	—	0.28
12	0.70	0.34	0.32	0.59	0.67	0.55	0.26
13	0.66	0.26	0.25	0.48	0.66	0.59	0.26

Tabela 6.14: Resultados sem reordenação sobre o conjunto de programas 'huffman'.

A tabela 6.14 apresenta valores semelhantes aos da tabela 6.10. Os casos em que a ordenação poderia ajudar (casos 3, 4 e 5) obtêm valores de semelhança inferiores, em geral. Essa tabela é menos interessante que a anterior, simplesmente porque a ordenação é crucial ao bom funcionamento dos diversos sistemas. E novamente, em geral o GST tem resultados melhores do que os outros sistemas, novamente confirmando resultados anteriores.

	GST-05-I	S-I	SG-I	W-05-I	LCS	Jplag	MOSS
01	1.00	0.86	0.79	1.00	1.00	—	—
02	1.00	0.86	0.79	1.00	1.00	1.00	0.97
03	1.00	0.86	0.79	1.00	1.00	1.00	0.77
04	0.99	0.58	0.65	0.91	0.89	0.94	0.88
05	0.99	0.58	0.65	0.91	0.88	0.54	0.62
06	0.89	0.69	0.61	0.85	0.90	0.91	0.96
07	1.00	0.86	0.79	1.00	0.97	1.00	0.77
08	1.00	0.86	0.79	1.00	1.00	0.46	0.97
09	0.80	0.33	0.26	0.51	0.64	0.61	0.27
10	0.76	0.34	0.18	0.69	0.63	0.65	0.29
11	0.76	0.38	0.22	0.64	0.68	—	0.28
12	0.69	0.36	0.26	0.67	0.67	0.55	0.26
13	0.74	0.31	0.29	0.61	0.66	0.59	0.26

Tabela 6.15: Resultados ignorando expressões sobre o conjunto de programas 'huffman'.

Na tabela 6.15 vemos os resultados quando se ignoram expressões. Novamente sem surpresas, os casos que tinham seu plágio dependendo da reordenação de átomos falham, como é o caso 3. Como no conjunto de dados anterior, essa tabela talvez seja a menos interessante, uma vez que é a classe de pré-processamento com menos 'inteligência'. Finalmente, na tabela 6.16 temos o pré-processamento sem ordenação e ignorando expressões. Novamente vemos algum comportamento estranho na peneiragem, embora não tanto quanto no caso de teste 'pip'. Em geral, os valores para o conjunto 'huffman' não divergem tanto.

	GST-05-NI	S-NI	SG-NI	W-05-NI	LCS	Jplag	MOSS
01	1.00	0.83	0.79	1.00	1.00	—	—
02	1.00	0.83	0.79	1.00	1.00	1.00	0.97
03	1.00	0.83	0.79	1.00	1.00	1.00	0.77
04	0.93	0.62	0.67	0.88	0.89	0.94	0.88
05	0.93	0.62	0.67	0.88	0.88	0.54	0.62
06	0.90	0.75	0.70	0.94	0.90	0.91	0.96
07	1.00	0.83	0.79	1.00	0.97	1.00	0.77
08	1.00	0.83	0.79	1.00	1.00	0.46	0.97
09	0.56	0.35	0.18	0.58	0.64	0.61	0.27
10	0.60	0.37	0.24	0.80	0.63	0.65	0.29
11	0.62	0.34	0.20	0.59	0.68	—	0.28
12	0.67	0.37	0.23	0.74	0.67	0.55	0.26
13	0.57	0.33	0.18	0.60	0.66	0.59	0.26

Tabela 6.16: Resultados sem reordenação e ignorando expressões sobre o conjunto de programas 'huffman'.

Capítulo 7

Conclusão

Plágio em tarefas escolares é um problema que parece estar aumentando com o tempo. Embora seja muito difícil desenvolver ferramentas generalizadas para detecção de plágio, fazer o mesmo para o problema de plágio em tarefas de programação é bastante mais simples, uma vez que é mais fácil identificar sua estrutura do que a de uma dissertação, por exemplo. Inclusive, existem vários sistemas gratuitos que oferecem esse serviço, utilizando diversos algoritmos diferentes, como o JPlag, o MOSS ou o SID.

No entanto, esses sistemas têm vulnerabilidades. É possível fazer hipóteses sobre o funcionamento desses sistemas e elaborar conjuntos de plágio que os enganam: conseguimos forçar os sistemas a gerar ambos falsos-positivos (pares de programas diferentes mas com um índice de semelhança grande) ou falsos-negativos (pares de programas semelhantes mas com um índice de semelhança baixo).

Acreditamos que esses problemas não são provindos dos algoritmos em si, e sim da etapa de pré-processamento. Além disso, acreditamos que a etapa de pré-processamento pode ser até mais importante do que a aplicação do algoritmo em si. Para testar isso, desenvolvemos três programas que utilizam os algoritmos dos sistemas mencionados acima: o *pygst* (utiliza o algoritmo do JPlag), o *pywinnow* (MOSS) e o *pyshared* (SID).

Também desenvolvemos um programa de pré-processamento com diferentes níveis de funcionamento. Nesse programa, exploramos a idéia de que ordenar os átomos gerados na análise léxica de maneira sintática conseguiria expor melhor a estrutura básica do programa, e ofereceria uma comparação melhor. Os resultados foram positivos, já que os conjuntos de programas ordenados ofereciam uma taxa de semelhança para programas plagiados maior do que quando não eram ordenados.

Outra perspectiva analisada é a de que talvez seja interessante manter mais informação sobre o programa. Em específico, se é interessante manter os dados das expressões que pertencem a um comando, ou se é melhor armazenar apenas o comando. Os resultados nesse caso foram menos certos, mas dentro do esperado. Conjuntos de programas que

variavam apenas nas expressões apresentavam índices de semelhança diferentes quando expressões são consideradas e quando não são.

Nos resultados obtidos para os casos em que é feita a ordenação, o valor de semelhança em geral é maior do que os casos em que a mesma não é feita (existe uma exceção para o LCS). Em particular, para certos tipos de plágio, a ordenação consegue identificar programas como quase idênticos, sem perda de informação. Isso confirma nossa segunda idéia, a de que é interessante manter mais informação.

Nos testes em que informação é perdida, ou seja, em casos em que expressões são ignoradas e apenas comandos são considerados, certas classes de plágio, as que tentam enganar o sistema com simples reordenação de expressões, são detectadas corretamente. Mas, por outro lado, essa estratégia também força casos de falsos-positivos, em particular em programas que tem estrutura de comandos semelhante mas expressões bastante diferentes.

Além disso, pudemos constatar que, dadas as nossas condições, e as limitações da nossa implementação (por exemplo, na falta de um algoritmo de compressão bem adequado à comparação), o RKRGST parece ser o que obtém melhores resultados. É claro que, ao contrário dos outros sistemas, sua complexidade é maior: sendo quadrático não só no número de programas comparados (como todos os sistemas são) mas também no comprimento de cada cadeia comparada. Para programas pequenos isso não é grande empecilho, mas para grandes conjuntos de textos talvez seja mais plausível utilizar a peneiragem, que é linear em relação ao comprimento dos programas.

Apesar dos resultados obtidos, seria interessante obter um conjunto maior de testes, já que esse foi um grande problema nesse trabalho. Apesar de tentativas de obter conjuntos de programas plagiados, tivemos que recorrer a exemplos sintéticos. Além disso, seria talvez interessante estudar algoritmos de compressão, para melhor utilizar a medida de informação compartilhada, porque talvez seja a mais interessante, e a mais aplicável, uma vez que é universal. Mas, talvez, até mesmo no melhor caso, a informação compartilhada, como métrica, seja limitada demais, uma vez que até mesmo os valores do SID, utilizando o algoritmo especial de compressão não conseguem identificar programas idênticos como tendo uma distância de zero, que seria o resultado ideal. O fato do *overhead* pesar tanto no resultado, mesmo no algoritmo otimizado, sugere que a medida obtida não é mais do que uma aproximação um tanto grosseira.

Para o futuro, seria interessante talvez expandir os algoritmos utilizados para possibilitar a busca de trechos de código plagiados, uma vez que isso também é bastante comum em submissões de alunos: copiar apenas a parte mais difícil de uma tarefa de outro aluno. Nos sistemas existentes, esse tipo de plágio provavelmente não seria detectado. Outra expansão para o futuro seria a de facilitar a comparação de um programa contra uma base de dados de submissões anteriores. Nos casos atuais isso é possível, mas

é necessário, ter todos os casos anteriores à mão para se testar. Outra expansão futura, claro, é a de ampliar as linguagens atendidas. Isso é um pouco mais difícil do que soa já que a linguagem das tarefas está bastante vinculada ao pré-processamento. Seria então um objetivo desvincular essa dependência, talvez quebrando os vários componentes do pré-processamento em módulos (análise sintática/léxica e construção da árvore sintática, achatamento e outras normalizações, ordenação).

Outras idéias para o futuro incluem efetuar uma normalização das expressões, como mencionada na seção 4.2. O sistema, como está implementado, poderia ser enganado se o plagiador manipular expressões criativamente. Parênteses, sinais negativos ou mesmo manipulação da associatividade de operadores podem ser usados para gerar expressões equivalentes, mas que poluiriam a normalização. Isso pode ser evitado ignorando expressões, embora para programas muito pequenos isso possa ser problemático. Por isso, levando a qualidade do resultado em mente, talvez fosse interessante explorar um processamento extra para normalizar expressões.

Um problema visto nos nossos resultados provém dos conjuntos de testes escolhidos. Em retrospecto, teria sido melhor se houvesse mais um caso de teste, para programas mais longos. Por outro lado, essa ferramenta tem como propósito primário encontrar plágio em submissões de alunos em disciplinas elementares de programação. Em geral, tais programas não chegam a ser longos, e portanto os conjuntos de testes utilizados atendem bem a esse propósito. Tendo programas mais longos, porém, ajudaria a confirmar a qualidade dos métodos de comparação. Para trabalhos futuros será necessário obter programas mais longos, especialmente se desejamos efetuar detecção de plágio sobre trechos de programas, como mencionado acima.

Finalmente, é fácil ver que o melhor resultado sem dúvida vêm não de um tratamento apenas, mas da submissão de um programa a uma bateria de testes, uma vez que todos os mecanismos parecem ter certos pontos fracos e pontos fortes. Nem todos os sistemas analisados são interessantes para esse propósito. Em particular, a peneiragem e o GST são os mais interessantes. Também provavelmente não há interesse em usar todos os mecanismos de pré-processamento; apenas o caso padrão e o caso que ignora expressões produzem resultados interessantes.

Bibliografia

[1] Ali Alemozafar. Online software battles plagiarism at Stanford. *The Stanford Daily, February 12, 2003*.

[2] ANSI C language, Bison/YACC file, `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`. Visited on 07/07/2007.

[3] ANSI C language, flex/lex file, `http://www.lysator.liu.se/c/ANSI-C-grammar-l.html`. Visited on 07/07/2007.

[4] Berkeley University of California, Student Conduct, Sanctions, `http://students.berkeley.edu/osl/sja.asp?id=1004`. Visited on 05/07/2007.

[5] Steven Burrows, Seyed M. M. Tahaghoghi, and Justin Zobel. Efficient and Effective Plagiarism Detection For Large Repositories. In *Proceedings of the Second Australian Undergraduate Students' Computing Conference*, 2004.

[6] Xin Chen, Brent Francia, Ming Li, Brian McKinnon, and Amit Seker. Shared Information and Program Plagiarism Detection. *IEEE Transactions on Information Theory*, 50(7), 2003.

[7] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for DNA sequences based on approximate matching. In R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology (RECOMB)*, page 107, Tokyo, Japan, April 8–11 2000. Association for Computing Machinery.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[9] Alair Pereira do Lago and Imre Simon. *Tópicos em Algoritmos sobre Seqüências*. IMPA, Edição do 24o Colóquio Brasileiro de Matemática edition, 2003.

[10] David Gitchell and Nicholas Tran. Sim: A Utility For Detecting Similarity in Computer Programs. In *SIGCSE '99*, 2004.

[11] JPlag webpage, `https://www.ipd.uni-karlsruhe.de/jplag/`. Visited on 04/09/2007.

[12] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 1987.

[13] Ming Li, Jonathan H. Badger, Xin Chen, Sam Kwong, Paul Kearney, and Haoyong Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2), 2000.

[14] Hermann Maurer, Frank Kappa, and Bilal Zaka. Plagiarism - A Survey. *Journal of Universal Computer Science*, 12(8), 2006.

[15] MOSS webpage, `http://theory.stanford.edu/~aiken/moss/`. Visited on 04/09/2007.

[16] Dave Muha. New Study Confirms Internet Plagiarism Is Prevalent, `http://urwebsrv.rutgers.edu/medrel/viewArticle.html?ArticleID=3408`, Visited 05/07/2007. *Rutgers University Press Release, August 8, 2003.*

[17] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, 8(11), 2002.

[18] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD 2003*, 2003.

[19] SID webpage, `http://genome.math.uwaterloo.ca/SID/`. Visited on 04/09/2007.

[20] Kristina L. Verco and Michael J. Wise. Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. In *First Australian Conference on Computer Science Education*, 1996.

[21] Michael J. Wise. Detection of Similarities in Student Programs: YAP'ing may be preferable to Plague'ing. In *SIGCSE 92*, 1992.

[22] Michael J. Wise. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. Unpublished, 1993.

[23] Michael J. Wise. Yap3: Improved detection of similarities in computer program and other texts. In *SIGCSE 96*, 1996.

[24] YAP webpage, `http://luggage.bcs.uwa.edu.au/~michaelw/YAP.html`. Visited on 04/09/2007.

[25] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Apêndice A

Tabelas completas

A seguir vemos tabelas completas, com todos os casos gerados. São semelhantes às do capítulo de comparação, mas com mais colunas. Em particular, vemos os resultados GST-10 e W-10 em suas diversas variantes. Também vemos uma variante da informação compartilhada, a que usa *bzip2*, que removemos do texto principal por não oferecer nada mais interessante do que o caso utilizando simplesmente *gzip*.

	GST-05	GST-05-I	GST-05-N	GST-05-NI	GST-10	GST-10-I	GST-10-N	GST-10-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98
03	0.98	1.00	0.76	1.00	0.91	1.00	0.70	1.00	0.92	1.00	0.23
04	0.94	0.91	0.84	0.81	0.91	0.74	0.74	0.81	0.83	0.50	0.38
05	0.93	0.91	0.70	0.81	0.79	0.74	0.41	0.81	0.83	0.25	—
06	0.90	0.93	0.90	0.93	0.90	0.77	0.90	0.93	0.90	0.97	0.98
07	0.72	1.00	0.69	1.00	0.53	1.00	0.61	1.00	0.79	1.00	0.23
08	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.35	0.98
09	0.79	0.44	0.77	0.33	0.67	0.00	0.71	0.00	0.68	0.25	0.27
10	0.87	0.80	0.79	0.77	0.75	0.60	0.65	0.58	0.77	0.81	0.69
11	0.82	0.96	0.80	0.96	0.74	0.67	0.58	0.82	0.88	0.88	0.29
12	0.99	1.00	0.94	0.90	0.92	1.00	0.84	0.75	0.98	0.81	0.51
13	0.88	0.86	0.85	0.83	0.74	0.86	0.69	0.79	0.88	0.65	—

Tabela A.1: Resultados GST sobre o conjunto de programas ‘pip’.

	W-05	W-05-I	W-05-N	W-05-NI	W-10	W-10-I	W-10-N	W-10-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98
03	0.86	1.00	0.63	1.00	0.96	1.00	0.52	1.00	0.92	1.00	0.23
04	0.73	1.00	0.79	0.75	0.79	1.00	0.69	1.00	0.83	0.50	0.38
05	0.73	1.00	0.48	0.75	0.76	1.00	0.30	1.00	0.83	0.25	—
06	0.97	1.00	1.00	1.00	1.00	1.00	0.89	1.00	0.90	0.97	0.98
07	0.62	1.00	0.58	1.00	0.51	1.00	0.41	1.00	0.79	1.00	0.23
08	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.35	0.98
09	0.46	0.15	0.52	0.00	0.48	0.07	0.43	0.00	0.68	0.25	0.27
10	0.64	0.21	0.66	0.46	0.63	0.08	0.61	0.25	0.77	0.81	0.69
11	0.77	0.75	0.61	0.78	0.72	0.86	0.37	0.86	0.88	0.88	0.29
12	0.91	1.00	0.86	0.90	0.90	1.00	0.90	1.00	0.98	0.81	0.51
13	0.68	0.90	0.65	0.82	0.63	1.00	0.47	1.00	0.88	0.65	—

Tabela A.2: Resultados da peneiragem sobre o conjunto de programas ‘pip’.

	S	S-I	S-N	S-NI	SBZ	SBZ-I	SBZ-N	SBZ-NI	SG	SG-I	SG-N	SG-NI	LCS	Jplag	MOSS
01	0.89	0.78	0.89	0.76	0.71	0.86	0.73	0.86	0.94	0.79	0.96	0.70	1.00	—	—
02	0.89	0.78	0.89	0.76	0.71	0.86	0.73	0.86	0.94	0.79	0.96	0.70	1.00	1.00	0.98
03	0.66	0.78	0.41	0.76	0.65	0.86	0.44	0.86	0.74	0.79	0.44	0.70	0.92	1.00	0.23
04	0.56	0.58	0.49	0.57	0.58	0.75	0.52	0.75	0.51	0.46	0.44	0.59	0.83	0.50	0.38
05	0.51	0.58	0.29	0.57	0.57	0.75	0.37	0.75	0.45	0.46	0.25	0.59	0.83	0.25	—
06	0.81	0.66	0.86	0.59	0.72	0.86	0.71	0.77	0.79	0.57	0.88	0.48	0.90	0.97	0.98
07	0.38	0.78	0.38	0.76	0.43	0.86	0.43	0.86	0.31	0.79	0.34	0.70	0.79	1.00	0.23
08	0.89	0.78	0.89	0.76	0.71	0.86	0.73	0.86	0.94	0.79	0.96	0.70	1.00	0.35	0.98
09	0.42	0.25	0.41	0.27	0.47	0.61	0.44	0.59	0.30	0.20	0.33	0.20	0.68	0.25	0.27
10	0.48	0.32	0.47	0.38	0.50	0.72	0.46	0.72	0.40	0.29	0.48	0.30	0.77	0.81	0.69
11	0.51	0.47	0.39	0.52	0.54	0.77	0.49	0.74	0.51	0.43	0.30	0.39	0.88	0.88	0.29
12	0.83	0.78	0.67	0.52	0.65	0.86	0.62	0.79	0.80	0.79	0.63	0.55	0.98	0.81	0.51
13	0.51	0.49	0.43	0.45	0.56	0.75	0.50	0.70	0.49	0.43	0.34	0.33	0.88	0.65	—

Tabela A.3: Resultados da informação compartilhada sobre o conjunto de programas ‘pip’.

	GST-05	GST-05-I	GST-05-N	GST-05-NI	GST-10	GST-10-I	GST-10-N	GST-10-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
03	1.00	1.00	0.97	1.00	1.00	1.00	0.97	1.00	1.00	1.00	0.77
04	0.94	0.99	0.94	0.93	0.93	0.95	0.95	0.93	0.89	0.94	0.88
05	0.94	0.99	0.90	0.93	0.93	0.95	0.90	0.93	0.88	0.54	0.62
06	0.90	0.89	0.90	0.90	0.90	0.89	0.90	0.90	0.90	0.91	0.96
07	0.93	1.00	0.97	1.00	0.92	1.00	0.94	1.00	0.97	1.00	0.77
08	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.46	0.97
09	0.66	0.80	0.69	0.56	0.57	0.44	0.55	0.39	0.64	0.61	0.27
10	0.69	0.76	0.68	0.60	0.57	0.65	0.52	0.43	0.63	0.65	0.29
11	0.69	0.76	0.73	0.62	0.55	0.62	0.57	0.39	0.68	—	0.28
12	0.73	0.69	0.70	0.67	0.57	0.50	0.56	0.38	0.67	0.55	0.26
13	0.67	0.74	0.66	0.57	0.48	0.62	0.47	0.38	0.66	0.59	0.26

Tabela A.4: Resultados GST sobre o conjunto de programas ‘huffman’.

	W-05	W-05-I	W-05-N	W-05-NI	W-10	W-10-I	W-10-N	W-10-NI	LCS	Jplag	MOSS
01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	—	—
02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
03	0.94	1.00	0.92	1.00	0.92	1.00	0.86	1.00	1.00	1.00	0.77
04	0.90	0.91	0.88	0.88	0.85	0.86	0.89	0.69	0.89	0.94	0.88
05	0.87	0.91	0.79	0.88	0.81	0.86	0.76	0.69	0.88	0.54	0.62
06	0.97	0.85	0.96	0.94	0.93	0.86	0.95	0.85	0.90	0.91	0.96
07	0.81	1.00	0.88	1.00	0.73	1.00	0.82	1.00	0.97	1.00	0.77
08	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.46	0.97
09	0.62	0.51	0.60	0.58	0.49	0.57	0.55	0.41	0.64	0.61	0.27
10	0.54	0.69	0.57	0.80	0.53	0.67	0.59	0.46	0.63	0.65	0.29
11	0.66	0.64	0.63	0.59	0.58	0.55	0.63	0.48	0.68	—	0.28
12	0.62	0.67	0.59	0.74	0.62	0.50	0.52	0.62	0.67	0.55	0.26
13	0.50	0.61	0.48	0.60	0.41	0.52	0.43	0.40	0.66	0.59	0.26

Tabela A.5: Resultados da peneiragem sobre o conjunto de programas ‘huffman’.

	S	S-I	S-N	S-NI	SBZ	SBZ-I	SBZ-N	SBZ-NI	SG	SG-I	SG-N	SG-NI	LCS	Jplag	MOSS
01	0.92	0.86	0.92	0.83	0.67	0.72	0.65	0.67	0.98	0.79	0.97	0.79	1.00	—	—
02	0.92	0.86	0.92	0.83	0.67	0.72	0.65	0.67	0.98	0.79	0.97	0.79	1.00	1.00	0.97
03	0.83	0.86	0.76	0.83	0.64	0.72	0.57	0.67	0.91	0.79	0.80	0.79	1.00	1.00	0.77
04	0.69	0.58	0.76	0.62	0.57	0.65	0.55	0.58	0.65	0.65	0.70	0.67	0.89	0.94	0.88
05	0.66	0.58	0.59	0.62	0.55	0.65	0.48	0.58	0.63	0.65	0.57	0.67	0.88	0.54	0.62
06	0.85	0.69	0.87	0.75	0.64	0.71	0.62	0.62	0.87	0.61	0.90	0.70	0.90	0.91	0.96
07	0.63	0.86	0.65	0.83	0.49	0.72	0.51	0.67	0.68	0.79	0.73	0.79	0.97	1.00	0.77
08	0.92	0.86	0.92	0.83	0.67	0.72	0.65	0.67	0.98	0.79	0.97	0.79	1.00	0.46	0.97
09	0.30	0.33	0.32	0.35	0.28	0.50	0.28	0.44	0.29	0.26	0.32	0.18	0.64	0.61	0.27
10	0.32	0.34	0.31	0.37	0.30	0.55	0.31	0.47	0.32	0.18	0.33	0.24	0.63	0.65	0.29
11	0.32	0.38	0.33	0.34	0.32	0.56	0.30	0.43	0.30	0.22	0.31	0.20	0.68	—	0.28
12	0.31	0.36	0.34	0.37	0.32	0.57	0.31	0.46	0.29	0.26	0.32	0.23	0.67	0.55	0.26
13	0.26	0.31	0.26	0.33	0.28	0.53	0.28	0.45	0.26	0.29	0.25	0.18	0.66	0.59	0.26

Tabela A.6: Resultados da informação compartilhada sobre o conjunto de programas ‘huffman’.

	GST-05	GST-10	S	SBZ	SG	W-05	W-10	LCS	Jplag	MOSS
01	1.00	1.00	0.89	0.71	0.94	1.00	1.00	1.00	—	—
02	1.00	1.00	0.89	0.71	0.94	1.00	1.00	1.00	1.00	0.98
03	0.98	0.91	0.66	0.65	0.74	0.86	0.96	0.92	1.00	0.23
04	0.94	0.91	0.56	0.58	0.51	0.73	0.79	0.83	0.50	0.38
05	0.93	0.79	0.51	0.57	0.45	0.73	0.76	0.83	0.25	—
06	0.90	0.90	0.81	0.72	0.79	0.97	1.00	0.90	0.97	0.98
07	0.72	0.53	0.38	0.43	0.31	0.62	0.51	0.79	1.00	0.23
08	1.00	1.00	0.89	0.71	0.94	1.00	1.00	1.00	0.35	0.98
09	0.79	0.67	0.42	0.47	0.30	0.46	0.48	0.68	0.25	0.27
10	0.87	0.75	0.48	0.50	0.40	0.64	0.63	0.77	0.81	0.69
11	0.82	0.74	0.51	0.54	0.51	0.77	0.72	0.88	0.88	0.29
12	0.99	0.92	0.83	0.65	0.80	0.91	0.90	0.98	0.81	0.51
13	0.88	0.74	0.51	0.56	0.49	0.68	0.63	0.88	0.65	—

Tabela A.7: Resultados dos diversos sistemas sobre o conjunto de programas ‘pip’.

	GST-05-N	GST-10-N	S-N	SBZ-N	SG-N	W-05-N	W-10-N	LCS	Jplag	MOSS
01	1.00	1.00	0.89	0.73	0.96	1.00	1.00	1.00	—	—
02	1.00	1.00	0.89	0.73	0.96	1.00	1.00	1.00	1.00	0.98
03	0.76	0.70	0.41	0.44	0.44	0.63	0.52	0.92	1.00	0.23
04	0.84	0.74	0.49	0.52	0.44	0.79	0.69	0.83	0.50	0.38
05	0.70	0.41	0.29	0.37	0.25	0.48	0.30	0.83	0.25	—
06	0.90	0.90	0.86	0.71	0.88	1.00	0.89	0.90	0.97	0.98
07	0.69	0.61	0.38	0.43	0.34	0.58	0.41	0.79	1.00	0.23
08	1.00	1.00	0.89	0.73	0.96	1.00	1.00	1.00	0.35	0.98
09	0.77	0.71	0.41	0.44	0.33	0.52	0.43	0.68	0.25	0.27
10	0.79	0.65	0.47	0.46	0.48	0.66	0.61	0.77	0.81	0.69
11	0.80	0.58	0.39	0.49	0.30	0.61	0.37	0.88	0.88	0.29
12	0.94	0.84	0.67	0.62	0.63	0.86	0.90	0.98	0.81	0.51
13	0.85	0.69	0.43	0.50	0.34	0.65	0.47	0.88	0.65	—

Tabela A.8: Resultados sem reordenação sobre o conjunto de programas ‘pip’.

	GST-05-I	GST-10-I	S-I	SBZ-I	SG-I	W-05-I	W-10-I	LCS	Jplag	MOSS
01	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	—	—
02	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	1.00	0.98
03	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.92	1.00	0.23
04	0.91	0.74	0.58	0.75	0.46	1.00	1.00	0.83	0.50	0.38
05	0.91	0.74	0.58	0.75	0.46	1.00	1.00	0.83	0.25	—
06	0.93	0.77	0.66	0.86	0.57	1.00	1.00	0.90	0.97	0.98
07	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.79	1.00	0.23
08	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	0.35	0.98
09	0.44	0.00	0.25	0.61	0.20	0.15	0.07	0.68	0.25	0.27
10	0.80	0.60	0.32	0.72	0.29	0.21	0.08	0.77	0.81	0.69
11	0.96	0.67	0.47	0.77	0.43	0.75	0.86	0.88	0.88	0.29
12	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.98	0.81	0.51
13	0.86	0.86	0.49	0.75	0.43	0.90	1.00	0.88	0.65	—

Tabela A.9: Resultados ignorando expressões sobre o conjunto de programas ‘pip’.

	GST-05-I	GST-10-I	S-I	SBZ-I	SG-I	W-05-I	W-10-I	LCS	Jplag	MOSS
01	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	—	—
02	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	1.00	0.98
03	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.92	1.00	0.23
04	0.91	0.74	0.58	0.75	0.46	1.00	1.00	0.83	0.50	0.38
05	0.91	0.74	0.58	0.75	0.46	1.00	1.00	0.83	0.25	—
06	0.93	0.77	0.66	0.86	0.57	1.00	1.00	0.90	0.97	0.98
07	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.79	1.00	0.23
08	1.00	1.00	0.78	0.86	0.79	1.00	1.00	1.00	0.35	0.98
09	0.44	0.00	0.25	0.61	0.20	0.15	0.07	0.68	0.25	0.27
10	0.80	0.60	0.32	0.72	0.29	0.21	0.08	0.77	0.81	0.69
11	0.96	0.67	0.47	0.77	0.43	0.75	0.86	0.88	0.88	0.29
12	1.00	1.00	0.78	0.86	0.79	1.00	1.00	0.98	0.81	0.51
13	0.86	0.86	0.49	0.75	0.43	0.90	1.00	0.88	0.65	—

Tabela A.10: Resultados sem reordenação e ignorando expressões sobre o conjunto de programas ‘pip’.

	GST-05	GST-10	S	SBZ	SG	W-05	W-10	LCS	Jplag	MOSS
01	1.00	1.00	0.92	0.67	0.98	1.00	1.00	1.00	—	—
02	1.00	1.00	0.92	0.67	0.98	1.00	1.00	1.00	1.00	0.97
03	1.00	1.00	0.83	0.64	0.91	0.94	0.92	1.00	1.00	0.77
04	0.94	0.93	0.69	0.57	0.65	0.90	0.85	0.89	0.94	0.88
05	0.94	0.93	0.66	0.55	0.63	0.87	0.81	0.88	0.54	0.62
06	0.90	0.90	0.85	0.64	0.87	0.97	0.93	0.90	0.91	0.96
07	0.93	0.92	0.63	0.49	0.68	0.81	0.73	0.97	1.00	0.77
08	1.00	1.00	0.92	0.67	0.98	1.00	1.00	1.00	0.46	0.97
09	0.66	0.57	0.30	0.28	0.29	0.62	0.49	0.64	0.61	0.27
10	0.69	0.57	0.32	0.30	0.32	0.54	0.53	0.63	0.65	0.29
11	0.69	0.55	0.32	0.32	0.30	0.66	0.58	0.68	—	0.28
12	0.73	0.57	0.31	0.32	0.29	0.62	0.62	0.67	0.55	0.26
13	0.67	0.48	0.26	0.28	0.26	0.50	0.41	0.66	0.59	0.26

Tabela A.11: Resultados dos diversos sistemas sobre o conjunto de programas ‘huffman’.

	GST-05-N	GST-10-N	S-N	SBZ-N	SG-N	W-05-N	W-10-N	LCS	Jplag	MOSS
01	1.00	1.00	0.92	0.65	0.97	1.00	1.00	1.00	—	—
02	1.00	1.00	0.92	0.65	0.97	1.00	1.00	1.00	1.00	0.97
03	0.97	0.97	0.76	0.57	0.80	0.92	0.86	1.00	1.00	0.77
04	0.94	0.95	0.76	0.55	0.70	0.88	0.89	0.89	0.94	0.88
05	0.90	0.90	0.59	0.48	0.57	0.79	0.76	0.88	0.54	0.62
06	0.90	0.90	0.87	0.62	0.90	0.96	0.95	0.90	0.91	0.96
07	0.97	0.94	0.65	0.51	0.73	0.88	0.82	0.97	1.00	0.77
08	1.00	1.00	0.92	0.65	0.97	1.00	1.00	1.00	0.46	0.97
09	0.69	0.55	0.32	0.28	0.32	0.60	0.55	0.64	0.61	0.27
10	0.68	0.52	0.31	0.31	0.33	0.57	0.59	0.63	0.65	0.29
11	0.73	0.57	0.33	0.30	0.31	0.63	0.63	0.68	—	0.28
12	0.70	0.56	0.34	0.31	0.32	0.59	0.52	0.67	0.55	0.26
13	0.66	0.47	0.26	0.28	0.25	0.48	0.43	0.66	0.59	0.26

Tabela A.12: Resultados sem reordenação sobre o conjunto de programas ‘huffman’.

	GST-05-I	GST-10-I	S-I	SBZ-I	SG-I	W-05-I	W-10-I	LCS	Jplag	MOSS
01	1.00	1.00	0.86	0.72	0.79	1.00	1.00	1.00	—	—
02	1.00	1.00	0.86	0.72	0.79	1.00	1.00	1.00	1.00	0.97
03	1.00	1.00	0.86	0.72	0.79	1.00	1.00	1.00	1.00	0.77
04	0.99	0.95	0.58	0.65	0.65	0.91	0.86	0.89	0.94	0.88
05	0.99	0.95	0.58	0.65	0.65	0.91	0.86	0.88	0.54	0.62
06	0.89	0.89	0.69	0.71	0.61	0.85	0.86	0.90	0.91	0.96
07	1.00	1.00	0.86	0.72	0.79	1.00	1.00	0.97	1.00	0.77
08	1.00	1.00	0.86	0.72	0.79	1.00	1.00	1.00	0.46	0.97
09	0.80	0.44	0.33	0.50	0.26	0.51	0.57	0.64	0.61	0.27
10	0.76	0.65	0.34	0.55	0.18	0.69	0.67	0.63	0.65	0.29
11	0.76	0.62	0.38	0.56	0.22	0.64	0.55	0.68	—	0.28
12	0.69	0.50	0.36	0.57	0.26	0.67	0.50	0.67	0.55	0.26
13	0.74	0.62	0.31	0.53	0.29	0.61	0.52	0.66	0.59	0.26

Tabela A.13: Resultados ignorando expressões sobre o conjunto de programas ‘huffman’.

	GST-05-NI	GST-10-NI	S-NI	SBZ-NI	SG-NI	W-05-NI	W-10-NI	LCS	Jplag	MOSS
01	1.00	1.00	0.83	0.67	0.79	1.00	1.00	1.00	—	—
02	1.00	1.00	0.83	0.67	0.79	1.00	1.00	1.00	1.00	0.97
03	1.00	1.00	0.83	0.67	0.79	1.00	1.00	1.00	1.00	0.77
04	0.93	0.93	0.62	0.58	0.67	0.88	0.69	0.89	0.94	0.88
05	0.93	0.93	0.62	0.58	0.67	0.88	0.69	0.88	0.54	0.62
06	0.90	0.90	0.75	0.62	0.70	0.94	0.85	0.90	0.91	0.96
07	1.00	1.00	0.83	0.67	0.79	1.00	1.00	0.97	1.00	0.77
08	1.00	1.00	0.83	0.67	0.79	1.00	1.00	1.00	0.46	0.97
09	0.56	0.39	0.35	0.44	0.18	0.58	0.41	0.64	0.61	0.27
10	0.60	0.43	0.37	0.47	0.24	0.80	0.46	0.63	0.65	0.29
11	0.62	0.39	0.34	0.43	0.20	0.59	0.48	0.68	—	0.28
12	0.67	0.38	0.37	0.46	0.23	0.74	0.62	0.67	0.55	0.26
13	0.57	0.38	0.33	0.45	0.18	0.60	0.40	0.66	0.59	0.26

Tabela A.14: Resultados sem reordenação e ignorando expressões sobre o conjunto de programas ‘huffman’.

Apêndice B

Enunciado dos conjuntos de teste

No apêndice a seguir veremos os enunciados dos dois conjuntos de teste utilizados nos experimentos. O enunciado do caso 1, ('pip') está em inglês, e portanto foi traduzido.

B.1 Enunciado do caso 1, 'pip'

Encontrado em `http://acm.uva.es/p/v107/10701.html`.

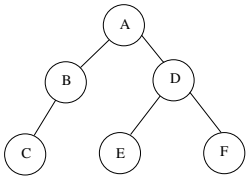
Pré, in e pós.

Tempo limite: 1 segundo

Um problema comum em estruturas de dados é determinar a travessia de uma árvore binária. Existem três maneiras clássicas de efetuar o mesmo:

- **Pré-ordem**:Você deve visitar a raiz, a sub-árvore da esquerda e a sub-árvore da direita, nessa ordem.
- **In-ordem**:Você deve visitar a sub-árvore da esquerda, a raiz e a sub-árvore da direita, nessa ordem.
- **Pós-ordem**:Você deve visitar a sub-árvore da esquerda, a sub-árvore da direita e a raiz, nessa ordem.

Veja a o diagrama a seguir:



A travessia pré, in e pós-ordem são, respectivamente, ABCDEF, CBAEDF e CBEFDA. Nesse problema, você deve computar a travessia em pós-ordem de uma árvore binária dada sua travessia em in- e pré-ordem.

Entrada

O conjunto de entrada consiste de um número positivo $C \leq 2000$, que consiste de um número de casos de entrada e C linhas, uma para cada caso de teste. Cada caso de teste começa com um número $1 \leq N \leq 52$, o número de nós nessa árvore binária. Depois, seguirão duas cadeias S_1 e S_2 que descrevem a travessia em pré-ordem e in-ordem da árvore. Os nós da árvore recebem nomes de $a..z$ e $A..Z$. N, S_1 e S_2 são separados por um espaço em branco.

Saída

Para cada conjunto de entrada voce deve imprimir uma linha contendo a travessa em pós-ordem da árvore.

Exemplo de entrada

```
3
3 xYz Yxz
3 abc cba
6 ABCDEF CBAEDF
```

Exemplo de saída

```
Yzx
cba
CBEFDA
```

B.2 Enunciado do caso 2, 'huffman'

Instituto de Computação da UNICAMP
Disciplina MC202: Estruturas de Dados
Segundo Semestre de 2006

Laboratório Nº 11
Profs. Tomasz Kowaltowski e Orlando Lee

O objetivo deste laboratório é a implementação e teste de algumas rotinas que implementam o método de compressão de Huffman. A implementação deve seguir as idéias expostas em aula. O arquivo huffman.c já contém as declarações de tipos e de algumas funções. Os comentários indicam a finalidade de cada um. Alguns comentários importantes:
*

- O arquivo de entrada pode conter qualquer caractere (um byte tem 8 bits e, portanto, 256 possibilidades).
- A função `ConstroiHuffman` deve construir, a partir de um texto original, um vetor de freqüências dos caracteres e dois objetos globais (as variáveis já estão declaradas): a árvore de Huffman `Arvore` e o vetor de apontadores para as folhas `Folhas` da mesma árvore. Estas estruturas são usadas por outras funções. As posições não utilizadas do vetor `Folhas` devem ser preenchidas com apontadores nulos, para permitir a detecção de caracteres inválidos.
- A construção da árvore deverá utilizar uma fila de prioridades (heap) para determinar, em cada passo, a árvore parcial de peso mínimo. Para isto, deve ser construído um vetor `floresta`, inicialmente igual ao vetor `Folhas` mencionado acima e transformado em seguida numa fila de prioridade com o elemento de peso mínimo na raiz. A cada remoção de duas subárvores será realizada uma inserção da nova árvore que as combina.
- A função `Comprime` deve percorrer a árvore, para cada letra do texto dado, a partir da folha correspondente, seguindo na direção da raiz através dos campos pai. Conseqüentemente, a cadeia de bits obtida deverá ser invertida.
- O programa principal testa as funções lendo do arquivo padrão de entrada o tamanho do alfabeto e as freqüências das letras. Após a criação das estruturas necessárias, é lido um texto constituído de várias linhas de letras que são concatenadas numa única cadeia. Esta cadeia é usada para compressão. A descompressão é testada com a aplicação ao texto comprimido.
- A implementação do laboratório pode seguir uma simplificação, na qual ao invés de verdadeiros bits, deve produzir uma seqüência de caracteres '0's e '1's que simulam

os bits. Esta opção é garantida por uma constante declarada no arquivo `config.h` (exibido sob o nome `config-original.h`):

`#`define PSEUDO_BITS 1

Uma implementação com bits verdadeiros pode ser escolhida removendo-se (ou comentando) esta linha e vale até dois pontos adicionais na nota final. Note que, neste caso, os bits da representação comprimida devem ser gerados ao longo do processo de compressão. Uma solução que primeiro gera uma seqüência auxiliar de pseudo-bits e depois a converte para bits verdadeiros não ganhará pontos extra. Para os que implementarem esta opção, uma dica é o uso do comando abaixo no gdb. Ele imprime uma variável em binário.

```
(gdb) p /t bits[0]
$1 = 110001
```