Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Techniques de détection automatique de plagiat de code source

Paheau, Philippe

Award date:
2009

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix
Faculté d'Informatique

Techniques de détection automatique de plagiat de code source

Philippe Paheau

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique

Année Académique 2008 - 2009

Résumé

Le plagiat de programmes dans un contexte scolaire n'a jamais été aussi aisé qu'aujourd'hui. Des études récentes établissent que le plagiat de code source par des étudiants en informatique dans les universités est un phénomène croissant. Depuis déjà une trentaine d'années, de nombreuses recherches ont été effectuées afin de concevoir des techniques de détection de plagiat de programmes. L'objectif de ce rapport est de présenter une description structurée du plagiat de code source, d'exposer les méthodes de détection de plagiat recensées dans la littérature et de comparer les principaux outils de détection de plagiat de code source actuels.

Mots-clés : Plagiat, plagiat de code source, similarités de code source, détection de plagiat.

Abstract

Programs plagiarism in school context has never been easier than it is today. Recent studies establish that source code plagiarism by students within computer sciences at university is an increasing phenomenon. Since already about thirty years, many researchs were carried out in order to conceive techniques of programs plagiarism detection. The objective of this report is to present a structured description of source code plagiarism, to expose the methods of plagiarism detection cited in the literature and to compare the current principal tools available for source code plagiarism detection.

Key-words: Plagiarism, source code plagiarism, source code similarities, plagiarism detection.

Remerciements :

Je remercie Monsieur Vanhoof, mon promoteur, pour sa disponibilité, sa patience et ses conseils avisés pendant l'élaboration de ce mémoire.

Je remercie également ma famille et mes amis pour leur soutien et leurs encouragements.

Table des matières

1 Introduction **10**

2 Description du plagiat de code source **15**

2.1 Introduction 15

2.2 Qu'est-ce que le plagiat de code source ? 15

2.2.1 Définition du plagiat de code source 16

2.2.1.1 Réutiliser du code source 16

2.2.1.2 Obtenir du code source 17

2.2.1.3 Ne pas citer correctement les sources 18

2.3 Techniques de plagiat 18

2.4 Conclusion 22

3 Classification des techniques de détection de plagiat de code source **23**

3.1 Introduction 23

3.2 Classifications traditionnelles 24

3.2.1 Systèmes à comptage d'attributs (attribute counting systems) 24

3.2.2 Systèmes à métrique de structure (structure metric systems) 27

3.2.3 Inconsistance des classifications traditionnelles 28

3.3 Lexèmisation 30

3.4 Techniques basées sur la génération d'empreintes 32

3.4.1 Empreintes quantitatives 33

3.4.2 Empreintes qualitatives 38
3.4.2.1 Algorithme de *Karp-Rabin* 40
3.4.2.2 Algorithme de sélection locale *winnowing* . . 43
3.5 Techniques basées sur la comparaison de chaînes de caractères 48
3.5.1 Algorithme *Running-Karp-Rabin Greedy-String-Tiling (RKS-GST)* 50
3.6 Techniques basées sur la comparaison d'arbres syntaxiques . . 59
3.7 Techniques basées sur la compression 68
3.8 Méthodes d'évaluation des techniques de détection de plagiat 75
3.9 Conclusion 77

4 Revue des outils actuels de détection de plagiat de code source 81
4.1 Introduction 81
4.2 Moteurs de détection modernes 82
4.2.1 *JPlag* 82
4.2.2 *MOSS* 87
4.2.3 *Sherlock* 89
4.2.4 *Copy/Paste Detector (CPD)* 94
4.3 Conclusion 96

5 Conclusion 102

Bibliographie 105

Table des figures

1.1 Méthodes de prévention et de détection de plagiat 12

2.1 Spectre de plagiat de Faidi et Robinson 20

3.1 Exemple de graphe de flux de contrôle 36
3.2 Liste des constructions proposées par McCabe 37
3.3 Diagramme de structure du programme original [1] 61
3.4 Diagramme de structure du programme copié [1] 61
3.5 Arbre de régions du programme original [1] 64
3.6 Arbre de régions du programme copié [1] 65
3.7 Dictionnaires des diagrammes de structure [1] 66
3.8 Contenu commun en information à des données 72
3.9 Architecture de *SID* 74
3.10 Évaluation des différents systèmes par l'étude de Mozgovoy . 77
3.11 Chronologie des différentes méthodes de détection de plagiat de code source 78
3.12 Techniques de détection de plagiat de code source 80

4.1 Sélection des soumissions dans *JPlag* 83
4.2 Processus de détection dans *JPlag* 84
4.3 Résultats affichés par *JPlag* 85
4.4 Comparaison côte à côte d'une paire de programmes dans *JPlag* 86
4.5 Soumission des programmes vers le serveur *MOSS* 87
4.6 Comparaison côte à côte d'une paire de programmes dans *MOSS* 88
4.7 *Sherlock* : écran initial 90

4.8 *Sherlock* : modification des paramètres 91
4.9 *Sherlock* : arbre de paires de programmes 92
4.10 *Sherlock* : écran de comparaison 93
4.11 *Sherlock* : graphe des résultats 94
4.12 *CDP* : écran initial 95
4.13 *CDP* : affichage des résultats 96

Liste des tableaux

3.1 Liste des opérandes de Halstead en Java 34
3.2 Liste des opérateurs de Halstead en Java 35
3.3 Comparaison des résultats fournis par *MOSS* et *BRASS [1]* . 67

4.1 Comparaison de fiabilité des systèmes *JPlag*, *MOSS* et *Sherlock* 97
4.2 Principales caractéristiques des différents moteurs modernes de détection de plagiat de code source 99
4.4 Principales caractéristiques des différents moteurs modernes de détection de plagiat de code source 100
4.6 Méthodes d'affichage des résultats des différents moteurs modernes de détection de plagiat de code source 101

Liste des algorithmes

3.1 Pseudo-code de l'algorithme *Karp-Rabin* 41
3.2 Pseudo Code de l'algorithme *Greedy String Tiling* 53
3.3 *Greedy String Tiling* 56

Chapitre 1

Introduction

Le plagiat n'est pas nouveau dans le monde académique, pourtant on assiste ces derniers temps à une forte recrudescence du phénomène. Internet, l'augmentation croissante de base de données de travaux scolaires et la standardisation de documents électroniques accélèrent l'ampleur du phénomène.

Aux États-Unis, une enquête menée par *The Center for Academic Integrity* [2] auprès de 60 000 étudiants du premier cycle universitaire en juin 2005 révèle que 80 % des étudiants de collège avouent avoir triché au moins une fois, 70 % des étudiants ont recours au plagiat à l'aide des technologies, 50 % des étudiants admettent avoir commis du plagiat électronique important à une ou plusieurs reprises dans le cadre de leurs travaux écrits, 77 % des étudiants ne croient pas qu'utiliser le 'copier – coller' sans citer les sources soit un geste répréhensible et 95 % des plagiaires signalent qu'ils ne se sont pas fait prendre.

Au Canada, une étude menée en 2006[2] auprès d'étudiants du post-secondaire affirme que 53 % des étudiants du premier cycle universitaire ont eu recours à un type ou un autre de plagiat électronique dans leurs travaux écrits et que, selon un des étudiants ayant participé à l'étude, 80 % des étudiants qui s'adonnent au plagiat à l'aide des technologies ne se font pas prendre. En outre, l'Université de Toronto enregistre chaque année 200 cas de plagiat et en 2002 l'Université d'Ottawa a sanctionné 100 étudiants reconnus coupables de plagiat.

En France, Pays-Bas et Belgique, une enquête réalisée par la société *Urkund*[3] sur 657 étudiants de 14 facultés d'économie, de commerce, de droit, de sciences humaines et de communication révèle que la part des étudiants admettant avoir triché au moins une fois au cours du dernier semestre varie de 16 à 67 %, que les trois quarts des étudiants ont été formés à l'utilisation correcte des sources et que 80 % déclarent savoir ce qu'est le plagiat. Pourtant, les deux tiers des étudiants considèrent le recours au plagiat comme étant intentionnel.

Les motivations de plagier peuvent être nombreuses. La méconnaissance des normes liées à la citation des sources et une gestion du temps inappropriée sont de loin les causes les plus répandues. Néanmoins, la désorganisation, une prise de note inadéquate, la sous-estimation de la charge de travail et le manque de compréhension de la matière sont également des facteurs motivants.

Le plagiat est un phénomène général qui touche l'ensemble des institutions éducatives, de l'école primaire jusqu'à l'université, n'épargnant aucune discipline, cependant les sciences se situent en tête de peloton, suivie des sciences de gestions, des sciences humaines et des langues. Il soulève également de nombreuses questions au niveau de l'éthique, de la prévention et de la détection, des conséquences sur l'apprentissage, de la législation et des sanctions à prendre. De plus, le plagiat interpelle également au plan des valeurs promues par l'école : l'honnêteté, l'effort et l'esprit critique[4].

De nos jours, de nombreuses méthodes ont été développées et utilisées pour combattre le plagiat. Ces méthodes peuvent être divisées selon deux catégories (figure 1.1) : les méthodes de prévention de plagiat et les méthodes de détection de plagiat. Les techniques de prévention, destinées à empêcher l'apparition du plagiat, demandent souvent beaucoup d'efforts mais ont un effet à long terme.

Généralement, la prévention du plagiat consiste en trois mesures :

- Rendre le plagiat plus difficile à réaliser techniquement par la préparation d'exercices individuels ou pour de petits groupes. Cette mesure est évidemment inapplicable au sein d'universités vu le nombre important d'étudiants.

- Sensibiliser les étudiants aux principes d'honnêteté intellectuelle, par exemple en enseignant les moyens de citer correctement les sources utilisées lors de travaux personnels.
- Publier des documents légaux visant à informer les étudiants des sanctions relatives à la découverte de cas de fraude.

Les méthodes de détection, quant à elles, sont destinées à découvrir le plagiat après son apparition et peuvent être réalisées selon deux manières :

- Comparer manuellement la source et la copie potentielle. Par exemple, des changements de style dans un document peuvent suggérer que le document analysé est une copie d'un original.
- Utiliser une méthode de détection automatique implémentée sous forme d'outils informatiques. Ce travail se focalise sur l'implémentation de ces outils informatiques, appelés moteurs de détection de plagiat.

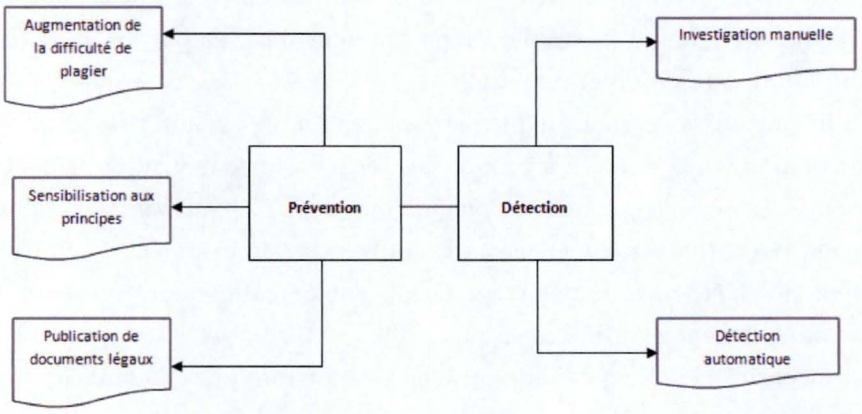

FIGURE 1.1 – Méthodes de prévention et de détection de plagiat

Ces moteurs sont des programmes qui comparent des documents avec d'autres sources possibles dans le but d'identifier des similarités et donc de découvrir des travaux d'étudiants qui auraient été plagiés. Deux familles de

détecteurs sont répertoriées dans la littérature : les moteurs de détection de plagiat de texte et les moteurs de détection de code source. Les premiers sont destinés à être utilisés pour détecter des cas de fraudes dans des thèses, mémoires, devoirs,... tandis que les seconds sont employés habituellement dans les laboratoires de programmation.

La différence majeure entre les deux systèmes se situe au niveau de la collection des documents à comparer. Dans le cas des outils de détection de plagiat de texte, le document potentiellement plagié doit être comparé avec de nombreuses sources localisées sur Internet, c'est pourquoi ces programmes fonctionnent à la manière de moteurs de recherche ou encore en maintenant de gigantesques bases de données de documents. L'approche est radicalement différente pour les moteurs de détection de plagiat de code source pour plusieurs raisons. D'une part il est difficile de trouver sur Internet le code source demandé par un professeur d'informatique lors d'un laboratoire de programmation. D'autre part, le partage de connaissance est le cas le plus fréquent de plagiat rencontré dans les travaux de programmation[5]. La collection de documents à comparer par le moteur de détection de plagiat de code source sera donc l'ensemble des programmes réalisés par les étudiants lors du laboratoire de programmation.

La littérature concernant le plagiat de code source est assurément abondante ; de nombreux articles à propos des techniques et des outils informatiques utilisés pour combattre le plagiat ont déjà été publiés. Cependant aucun de ces articles n'est en mesure de proposer une synthèse complète à propos des méthodes de dépistage de plagiat de code et des moteurs de détection actuellement utilisés.

La motivation principale de ce travail est donc de présenter un état de l'art des différentes recherches effectuées dans le cadre du plagiat de code source. Plus précisément, les objectifs de ce mémoire sont triples :

- La première partie tente d'expliquer ce qui constitue du plagiat en exposant une définition structurée de plagiat de code source. Les pratiques utilisées par les plagiaires y sont également identifiées et détaillées pour préciser les fonctionnalités attendues par les outils de détection de plagiat.

- La deuxième partie présente de façon détaillée chacune des différentes techniques de détection de plagiat de code source identifiées dans la littérature. Ce chapitre propose également une classification de ces techniques plus appropriée que celle habituellement citée dans la littérature.
- Dans le dernier chapitre, les principaux moteurs actuels de détection de plagiats sont exposés. Chaque moteur est décrit de façon détaillée d'un point de vue utilisateur et plusieurs tableaux comparatifs permettent de confronter les différents moteurs suivant une vingtaine de caractéristiques.

Assurément, le plagiat de code source n'est pas limité au contexte scolaire ; les techniques ainsi que les outils informatiques de détection sont parfaitement applicables à des cas de plagiat dans un contexte industriel. Certains moteurs de détection, comme *JPlag*[6], ont déjà été utilisés avec succès dans ce sens. Cependant, les techniques et outils de détection de plagiat de code source sont généralement le fruit de recherches universitaires et sont avant tout développés pour détecter des similitudes parmi des travaux de programmation. Il en résulte que la littérature nécessaire à l'élaboration de ce travail est axée principalement sur la détection de plagiat de code source appliquée à un contexte scolaire, ce qui implique que ce travail le soit aussi et ce qui n'empêche nullement l'utilisation des mêmes méthodes de détection dans un cadre industriel.

Chapitre 2

Description du plagiat de code source

2.1 Introduction

Les moteurs de détection de plagiat de code source sont des programmes qui comparent des fichiers de code source dans le but d'identifier des similarités et donc de découvrir des travaux d'étudiants qui auraient été plagiés. L'implémentation de ces outils est directement liée à la définition du plagiat. Définir le plagiat de code source permet donc de préciser les fonctionnalités d'un détecteur de plagiat. Ce chapitre tente de présenter les bases d'une définition structurée du plagiat de code source ainsi que les techniques utilisées par les plagiaires pour modifier du code source.

2.2 Qu'est-ce que le plagiat de code source ?

Ordinairement le plagiat consiste à s'inspirer d'un modèle que l'on omet délibérément de désigner. Certaines définitions, citées dans la littérature, sont plus précises mais restent informelles :

- Selon Faidhi et Robinson[7] *« le plagiat se produit quand des travaux de programmation sont copiés ou transformés avec peu d'efforts par les*

étudiants ».

- Joy et Luck[8] définissent le plagiat comme « *copier des documents et des programmes sans référencer leur source* ».
- D'après Parker et Hamblen[9] un programme plagié est « *un programme produit à partir d'un autre avec peu de transformations et sans vraiment en comprendre le fonctionnement en détail* ». Ces transformations varient des plus simples (tel que modifier des commentaires) aux plus complexes (tel que modifier des structures de contrôle). Ces modifications peuvent par ailleurs être classées selon leur degré d'altération.
- Manber[10] affirme que « *deux fichiers sont similaires si ceux-ci contiennent un nombre significatif de chaînes de caractères communes qui ne sont pas trop courtes* ».

Malheureusement ces définitions sont relativement limitées et peu explicites. Une étude, réalisée en 2006 par Cosma et Joy[11] dans les universités anglaises, présente une description détaillée du plagiat de code source et ce, d'après les réponses de 59 professeurs d'informatique. Cette enquête permet de comprendre de manière plus formelle en quoi constitue le plagiat de code source dans un contexte scolaire.

2.2.1 Définition du plagiat de code source

Selon l'étude menée par Cosma et Joy, « *le plagiat de code source dans les projets de programmation survient quand un étudiant réutilise du code source obtenu avec ou sans la permission de l'auteur original et en ne citant pas correctement, intentionnellement ou non, la source empruntée* ».

2.2.1.1 Réutiliser du code source

La réutilisation de code source inclut les cas suivants :

1. Reproduire ou copier du code source sans aucune modification.
2. Reproduire ou copier du code source avec des modifications mineures ou modérées, le code soumis contenant toujours des parties du code source original.
3. Convertir l'intégralité ou une partie de code source produit par quelqu'un d'autre dans un langage de programmation différent. Le plagiat dépend alors de la similarité entre les deux langages et de l'effort requis pour effectuer cette conversion.
4. Générer automatiquement du code source en utilisant un logiciel approprié si l'utilisation d'un tel outil n'est pas autorisée.
5. Réutiliser son propre code source, produit par exemple lors d'un autre devoir de programmation, sans citer correctement ce fait, constitue de l'auto plagiat ou une infraction au règlement de l'université. Il est à noter que la majorité des professeurs interrogés considèrent que la réutilisation de code source sans citer les références constitue du plagiat alors qu'une minorité en disconvient en argumentant que la réutilisation de code est une pratique encouragée dans la programmation orientée objet[11].
6. Réutiliser du code produit par un autre en référençant correctement les sources ne constitue pas du plagiat mais peut conduire à une infraction au règlement de l'université.

2.2.1.2 Obtenir du code source

L'obtention de code source avec ou sans la permission de l'auteur original inclut les cas suivants :

1. Payer une autre personne pour produire une partie ou l'intégralité du code source.
2. Voler le code source d'un autre étudiant.
3. Collaborer sur un projet de programmation individuel avec un ou plusieurs autres étudiants et produire du code source similaire.

4. Échanger des parties de code source lors de travaux de programmation.

2.2.1.3 Ne pas citer correctement les sources

La non citation correcte de sources, inclut les cas suivants :

1. Ne pas citer la source utilisée et son auteur dans le code source produit (par un commentaire) et, éventuellement, dans la documentation associée.
2. Fournir de fausses références. Ces références peuvent être inventées par l'étudiant ou ne pas correspondre avec la source citée.

La définition du plagiat de code source donnée par Cosma et Joy est sans aucun doute la plus complète trouvée dans la littérature. Cependant les détecteurs de plagiat ne peuvent gérer toutes les situations citées précédemment. En effet, la détection de plagiat ne peut s'opérer que sur des documents se trouvant dans la même collection à analyser. Le cas où un étudiant paierait une tierce personne pour produire un devoir de programmation est donc indétectable. De plus, aucune information permettant d'identifier les modifications qu'un étudiant pourrait apporter à du code source original n'est spécifiée. Il est donc utile d'énumérer ces transformations dans la suite du chapitre. En outre, la manière de citer correctement une ou plusieurs sources n'est pas explicitée de façon claire. La citation, sous forme de commentaire, devrait spécifier distinctement qu'une ou plusieurs parties de codes ont été empruntées et devrait mentionner le nom de l'auteur, la référence bibliographique ou l'adresse Internet du code reproduit. L'étudiant devrait également indiquer de façon précise où débute la partie de code empruntée et où celle-ci se termine.

2.3 Techniques de plagiat

En admettant que la détection de plagiat dans le code source se borne à identifier un simple travail de recopiage entre deux programmes, la tâche

serait triviale. Un éditeur basique de fichier permettrait de déceler que les deux fichiers de code source présentent des similitudes. Malheureusement la nature même du code source rend difficile la détection de plagiat basée uniquement sur le texte du programme car dans la plupart des cas le code copié est typiquement altéré pour justement éviter cette détection.

Whale[12] identifie 12 techniques utilisées pour modifier des programmes. Ces pratiques sont listées ci-dessous et triées selon leurs degrés de sophistication. Pour certaines, des exemples sont cités entre parenthèses.

1. Modifier les commentaires. (ajout, suppression, modification)
2. Modifier le formatage du programme. (indentation)
3. Modifier les identifiant. (renommage)
4. Modifier l'ordre des opérandes dans les expressions.
5. Modifier le type des données. (par un entier vers un réel)
6. Remplacer des expressions par des expressions équivalentes. ('while found = false do ... ' par 'while not found do...')
7. Ajouter et/ou supprimer des instructions ou variables redondantes. (initialisations et/ou ajout d'instructions de sortie superflues)
8. Modifier l'ordre des instructions. (Réordonner des clauses en Prolog)
9. Modifier la structure des instructions d'itération. (Utiliser l'instruction 'repeat' à la place de 'while', 'while' à la place de 'for')
10. Modifier la structure des instruction de sélection. (Utiliser 'if' à la place de 'case')
11. Remplacer un appel de procédure par le corps de la procédure appelée.
12. Combiner des fragments du programme original avec ceux du programme copié.

Faidhi et Robinson[7] ,quant à eux, ont développé un spectre de plagiat détaillant 6 niveaux de modifications qui peuvent être apportées au code source

d'un programme. La gamme de transformations s'étend de la simple modification des commentaires et/ou l'indentation jusqu'à la transformation de la logique de contrôle. L'idée sous-jacente est d'appliquer ces 6 niveaux de modifications dans l'ordre pour produire une refonte complète du programme original.

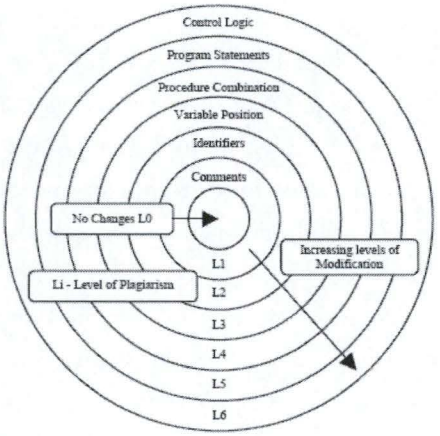

FIGURE 2.1 – Spectre de plagiat de Faidi et Robinson

Les 6 niveaux de transformations du spectre présenté à la figure 2.1 sont définis comme suit :

- L0 : Aucun changement ; il s'agit du programme original.
- L1 : Représente les changements effectués au niveau des commentaires et/ou de l'indentation.
- L2 : Représente les changements effectués dans le niveau 1 et les changements apportés au niveau des identifiants.
- L3 : Représente les changements effectués dans le niveau 2 et les changements apportés aux déclarations (i.e. déclarer de nouvelles constantes, permuter des déclarations de variables, inverser des fonctions, etc.).
- L4 : Représente les changements effectués dans le niveau 3 et les changements apportés aux modules du programme (i.e. combiner deux fonc-

tions pour n'en former qu'une seule, créer de nouvelles fonctions, etc.).
- L5 : Représente les changements effectués dans le niveau 4 et les changements apportés aux instructions du programme (i.e. utiliser une boucle '*for*' à la place d'une boucle '*while*', etc.).
- L6 : Représente les changements effectués dans le niveau 5 et les changements apportés à la logique de contrôle (i.e. remplacer des boucles ou des instructions conditionnelles sans altérer le résultat du programme).

Les listes de techniques de plagiat proposées par Whale, Faidhi et Robinson ne sont pas exhaustives ; l'identification de tous les changements possibles que les plagiaires sont susceptibles d'employer pour déguiser leurs copies serait infaisable. Il apparaît cependant les transformations pouvant être apportées à du code source se divisent en deux catégories [8] :

- Les modifications lexicales : réalisées avec un simple éditeur de fichier texte et ne requérant pas une connaissance avancée du langage de programmation utilisé. Ces changements peuvent inclure l'ajout et/ou la suppression de commentaires, le reformatage et le renommage de variable ; respectivement les niveaux 1 à 3 du spectre de Faidhi et Robinson[7].
- Les modifications structurelles : celles-ci exigent une connaissance effective du langage de programmation utilisé et peuvent inclure le remplacement de structures de contrôle par des structures équivalentes et/ou le réordonnancement des variables ; respectivement les niveaux 4 à 6 du spectre de Faidhi et Robinson[7].

En général, les outils de détection de plagiat traitent sans difficulté les transformations lexicales (par exemple en ignorant les commentaires), les manipulations structurelles sont plus difficiles à gérer mais ce degré de complexité est directement proportionnel à la connaissance du langage requise pour effectuer ces modifications. Différentes techniques peuvent être combinées, ce qui rendrait la détection de plagiat plus difficile, néanmoins les efforts fournis pourraient être alors plus importants que ceux nécessaires à la réalisation du

programme original.

Il faut encore remarquer que les techniques de transformations identifiées par Whale, Faidhi et Robinson, Cosma et Joy sont fortement liées aux langages de programmation impérative et ne peuvent être appliquées intégralement à d'autres paradigmes de programmation. Cette constatation peut être expliquée par le fait que les recherches menées dans le but d'identifier les techniques utilisées par des plagiaires ont été menées dans un cadre scolaire au début des années nonante et que la programmation impérative était majoritairement la plus enseignée au sein des universités lors de ces recherches.

2.4 Conclusion

Même si le concept de plagiat de code source reste assez vague dans la littérature, il semble qu'il existe un accord commun au sein de la communauté scientifique sur la classification des modifications apportées au code source par les plagiaires et donc sur les aptitudes requises par les outils de détection de plagiat de code source. Néanmoins, chaque moteur de détection de plagiat implémente sa propre définition du plagiat de code source, la fiabilité de l'outil utilisé dépend donc fortement de cette définition. Celle-ci est en général explicitée dans la documentation du moteur employé.

Comme expliqué précédemment, certains cas de plagiat ne peuvent être détectés. La localisation de la source du programme original et de la copie dans la même collection à traiter par le détecteur de plagiat est la condition nécessaire pour la tâche soit réalisable. La suite de ce travail se réfère donc au plagiat de code source géré par les outils informatiques.

Chapitre 3

Classification des techniques de détection de plagiat de code source

3.1 Introduction

Ce chapitre décrit les différentes techniques utilisées par les moteurs de détection de plagiat de code source recensées dans la littérature. Selon de nombreux articles[13, 14, 12] deux approches majeures ont émergées : les systèmes à comptages d'attributs (*attributes-counting systems*) et les systèmes à métriques de structure (*structure metrics systems*). Même si cette classification semble désuette[15, 16], la frontière entre ces systèmes apparaissant floue, elle présente néanmoins un intérêt chronologique permettant de comprendre l'évolution des systèmes. C'est pourquoi la première section du chapitre décrit de façon générale les deux systèmes précédemment cités et explique pourquoi il est difficile de classer les différentes méthodes de détection de plagiat de manière formelle dans ces deux catégories. La partie suivante, quant-à-elle, utilise la taxinomie évoquée par Mozgorov[15] qui paraît plus appropriée pour expliciter les techniques de détection recensées dans la littérature. Celui-ci affirme que les moteurs de détection de plagiat de code source actuels appartiennent à l'une de ces trois catégories : l'approche

basée sur la génération d'empreinte, l'approche basée sur la comparaison de chaînes de caractères et celle basée sur la comparaison d'arbre. Néanmoins Mozgorov omet de citer une approche recensée régulièrement dans la littérature, celle basée sur les techniques de compression. Il est donc nécessaire d'ajouter cette approche dans la taxonomie de Mozgorov. La dernière partie du chapitre traite des diverses procédures d'évaluation des techniques de détection de plagiat de code source.

3.2 Classifications traditionnelles

3.2.1 Systèmes à comptage d'attributs (attribute counting systems)

Les systèmes à comptage d'attributs apparurent dans les années septante et sont considérés comme les premiers moteurs de détection de plagiat de code source. L'idée maîtresse de cette technique est d'assigner à chaque programme un simple nombre (métrique) ou un ensemble de nombres (vecteur de métriques) représentant une simple analyse quantitative des caractéristiques du programmes (attributs). Après avoir été collectées, ces mesures sont combinées dans un vecteur de métriques, encore appelé profil ou empreinte, et ceci pour chaque programme à comparer. Ensuite une mesure de distance, par exemple la distance euclidienne, est utilisée pour calculer la différence entre ces profils. Les programmes analysés sont considérés comme similaires, et donc potentiellement plagiés, si la distance entre leur profil est faible. La majorité de ces systèmes utilisèrent notamment les métriques de Halstead[17], de McCabe[18], ou encore un nombre de portée[19]. Ces différentes métriques sont généralement utilisées pour déterminer la complexité d'un programme mais présentent néanmoins un intérêt quant à la détection de similarités au sein de code source.

Ottenstein[20] fut le premier à développer un système utilisant les métriques de Halstead pour détecter des similitudes au sein de programmes écrits en FORTRAN. Cette technique s'est révélée très inefficace car elle ne résistait pas à des altérations basiques du programme à analyser.

Un bon exemple de système à comptage d'attributs est le programme *Accuse*[21]. D'une part, *Accuse* utilise des paramètres similaires à ceux employés par la plupart des systèmes à comptages d'attributs, et d'autre part, *Accuse* fût réimplémenté par Verco and Wise[22] dans le but d'être évalué. Le profil généré par *Accuse* est déterminé par sept paramètres. Les quatre premiers sont directement dérivés des métriques de Halstead[17] et définis par le nombre d'opérandes (les variables) et d'opérateurs (les symboles) comptés dans le fichier de code source. Les métriques de Halstead sont expliqués plus précisement dans la suite du chapitre. Les autres paramètres utilisés par *Accuse* sont le nombre de lignes de code en excluant les lignes vides ou commentées, le nombre de variables déclarées et le nombre total d'instructions de contrôle.

Ces sept métriques sont calculées et placées dans un vecteur durant la première phase de l'algorithme implémenté dans *Accuse* et ceci pour chaque programme source. Dans la seconde phase les paires de vecteurs de métriques sont comparées pour générer un score de similarité. Ce score, initialement fixé à 0, est calculé par une fonction, appelée fonction de correspondance. Une constante est utilisée dans le calcul en assignant à chaque métrique une valeur reflétant son importance.

- Si A_i est la valeur du i^{eme} métrique du programme A et que B_i est la valeur du i^{eme} métrique du programme B
- alors la corrélation entre A et B peut être définie par : $correlation_{AB} += importance_i - |A_i - B_i|$

Une fois les scores de correspondance calculés, *Accuse* les affiche dans un ordre décroissant, les paires de programmes ayant un score de similarité élevé, et donc potentiellement plagiés, apparaissant au sommet de la liste.

Selon Verco and Wise[22], les résultats obtenus par *Accuse* sont néanmoins très peu convainquants. *Accuse* résiste relativement bien aux transformations lexicales qu'un plagiaire pourrait apporter à du code source mais reste complètement insensible aux transformations structurelles.

Plus tard des d'autres systèmes à comptage d'attributs furent conçus en

introduisant un plus grand nombre de métriques.

Faidhi et Robinson[7] utilisèrent jusqu'à 24 métriques, les 10 premières étant susceptibles d'être modifiées par un programmeur novice et le reste par un plagiaire plus expérimenté. Ce système fût également réimplémenté par Verco et Wise[22] et les résultats obtenus par ces derniers sont également peu persuasifs. Leur conclusion souligne le fait qu'aucune métrique ou ensemble de métriques ne parvient à capturer l'information structurelle nécessaire à la détection de similitudes de code source.

La tentative la plus récente est celle de Jones[23]. Le système implémenté par Jones crée trois profils ou empreintes différentes. La première empreinte est un vecteur décrivant le programme en terme d'attributs physiques : nombre de lignes, nombre de mots et nombre de caractères. La deuxième empreinte contient trois des métriques de Halstead et une combinaison des deux premiers profils forme la troisième empreinte. Une fois collectés, les profils sont normalisés et un score de similarité est généré en utilisant la distance euclidienne. Les recherches de Jones sont néanmoins limitées et les performances de son système sont peu concluantes.

Whale[12] a démontré que les systèmes à comptage d'attributs sont incapables de donner de bons résultats car trop d'information structurelle est simplement ignorée. Cette déficience ne peut cependant pas être contournée en ajoutant des métriques. De tels systèmes deviendraient soit insensibles (et donc faciles à tromper par de simples transformations du programme original) soit trop sensibles (et détecteraient un trop grand nombre de faux positifs).

La majeure partie du problème s'avère être la difficulté de choisir des métriques appropriées, ce qui s'avère être un problème insoluble, c'est pourquoi les moteurs actuels de détection de plagiat de code source n'utilisent plus cette technique.

Cependant, même si les techniques à comptage d'attributs ne sont plus employées actuellement, il apparaît nécessaire de pouvoir les classer dans la taxonomie de Mozgovoy décrite dans l'introduction de ce chapitre. Comme ces techniques génèrent une empreinte (vecteur de métriques) basée sur des statistiques logicielles, celles-ci seront appelées, dans la suite du travail, tech-

niques à génération d'empreintes quantitatives, ce qui permet donc de les classer dans la catégorie des techniques à génération d'empreinte définie selon la terminologie de Mozgovoy. Les méthodes de détection de plagiat par génération d'empreintes sont détaillées dans la suite du chapitre.

3.2.2 Systèmes à métrique de structure (structure metric systems)

Selon la littérature [12, 22, 14], la seconde approche utilisée par les moteurs de détection de plagiat est de comparer des programmes selon leur structure plutôt que par un ensemble d'attributs. Les métriques de structure permettent de mesurer une propriété d'un document mais nécessitent une connaissance de la structure du programme à analyser. Les programmes sont typiquement convertis dans une représentation de chaînes de caractères afin de les comparer en supposant que les programmes plagiés partageront la plus longue chaîne de caractères. En utilisant, par exemple, un algorithme de comparaison de chaînes de caractères il devient donc possible de calculer la similarité entre deux programmes. Les paires de programmes dont la similarité excède un certain seuil seront marqués comme potentiellement plagiés. Clough[14] donne, par exemple, un algorithme basique de détection de plagiat basé sur la comparaison de chaînes de caractère :

1. Supprimer tous les commentaires.
2. Ignorer les espaces, retours à la ligne.
3. Effectuer une comparaison de chaînes de caractères entre les fichiers avec les outils UNIX diff, grep et wc.
4. Calculer et enregistrer le pourcentage de caractères identiques pour ces deux fichiers.
5. Exécuter les instructions 3 et 4 pour toutes les paires de programmes possibles.
6. Créer une liste contenant les paires de fichiers comparés avec le pourcentage de caractères identiques trié en ordre décroissant.

Les systèmes à métrique de structure ont l'avantage de fournir aussi bien des résultats quantitatifs, par exemple un score de similarité, que qualitatifs, par exemple une représentation visuelle des fragments de code présentant des similarités. Une grande majorité des moteurs de détection actuels utilisent dès lors ce type de comparaison.

Cependant, les définitions des systèmes à métrique de structure recensées dans la littérature apparaissent relativement imprécises et incomplètes, ce qui implique que la terminologie des systèmes à métrique de structure ne peut être utilisée pour répertorier les différentes techniques de détection de plagiat de code source. La section suivante explique de manière détaillée la faiblesse de ces définitions.

3.2.3 Inconsistance des classifications traditionnelles

Les descriptions des deux systèmes précédemment cités ne sont pas formelles, de nombreuses inconsistances sont relevées dans la littérature.

Selon Verco et Wise[22], *« les systèmes de comptage d'attributs sont ceux qui permettent de mesurer une ou plusieurs propriétés d'un programme »*.

Les systèmes utilisant des métriques de structure des programmes sont quant à eux présentés *« comme recherchant des similarités dans une représentation de deux morceaux de code source en plus d'utiliser les techniques de comptage d'attributs »*.

Verco et Wise reconnaissent que leur propre moteur (*YAP3*), un système à métrique de structure, n'utilise pas de comptage d'attributs. Cette affirmation apparaît donc directement contradictoire avec la définition donnée précédemment, *YAP3* ne pouvant être classé dans aucun des deux groupes de moteurs de détection de plagiat.

Culwin, MacLeod, et Lancaster[24] fournissent des définitions vaguement similaires à celles données par Verco et Wise.

Selon eux, les systèmes de comptage d'attributs décrivent uniquement *« les moteurs qui examinent de manière superficielle le programme à analyser sans connaissance de sa structure tandis que les systèmes à métrique de structure utilisent uniquement des techniques de lexèmisation pour recher-*

cher des chaînes de caractères communes aux programmes analysés ». Ceci est directement inconsistant avec les affirmations de Verco et Wise. En effet, un moteur utilisant simultanément des techniques de comptage d'attributs et de métrique de structure serait répertorié par Verco et Wise comme un système à métrique de structure alors qu'il n'appartiendrait à aucune catégorie conformément aux classifications définies par Culwin, MacLeod, et Lancaster.

Plus récemment Jones[23] échoue quant à la classification de son propre système. Ce dernier affirme que son propre moteur, un système de comptage d'attributs, « *fonctionne selon les mêmes principes que YAP3* », un moteur à métrique de structure. Ceci démontre qu'une certaine confusion règne au sein même de la communauté des développeurs de moteurs de détection de plagiat de code source.

Il devrait être possible de redéfinir ces classifications plus formellement pour les rendre plus consistantes. Néanmoins certains systèmes de détection de plagiat ne peuvent appartenir à aucun des deux groupes de moteurs. Le prototype de détection de plagiat développé par Saxon[25] en est un exemple pertinent. Ce système permet de rechercher des similarités basées sur la compressibilité du code source et n'utilise ni technique de comptage d'attributs ni technique de métrique de structure. Par conséquent il ne peut être répertorié dans aucune catégorie existante.

Il est donc impossible de classer les différentes techniques de détection de plagiat de code source selon les classifications habituellement citées dans la littérature. Ce travail utilise la terminologie évoquée par Mozgovoy, plus conforme pour expliciter les techniques de détection recensées dans la littérature. Ces techniques appartiennent dès lors à l'une des quatres familles suivantes :

- Comparaison d'empreintes
  - Empreintes quantitatives
  - Empreintes qualitatives
- Comparaison de chaînes de caractères
- Comparaison d'arbres syntaxiques
- Compression de fichiers

La suite du chapitre explique de façon détaillée le fonctionnement de ces techniques.

3.3 Lexèmisation

La plupart des moteurs de détection de plagiat de code source actuels utilisent une phase de pré-traitement appelée lexèmisation (*tokenization*). Cette phase est généralement commune à toutes les techniques de détection ; seule la technique basée sur la comparaison d'empreintes quantitatives échappe à cette règle. Cette méthode est vue en détail dans la suite du chapitre à la section 3.4.1.

La lexèmisation permet d'obtenir une unité d'abstraction du code source à partir d'une séquence de lexèmes (*token*). Un analyseur lexical permet de générer cette séquence. Durant l'analyse lexicale, le code source subit une série de transformations bénéfiques pour la détection de similitudes.

Par exemple, ces deux portions de code source Java, sémantiquement identiques, ont clairement subit une suite de transformations lexicales. Le tableau A a été remplacé par B et l'indice i par l'indice j.

```
0. int [ ] A = {1,2,3,4} ;
1. for (int i = 0 ; i < A.length ; i++) {
2.  A[i] = A[i] + 1 ;
3. }
```

```
0. int [ ] B = {1,2,3,4} ;
1. for (int j = 0 ; j < B.length ; j++) {
2.  B[j] = B[j] + 1 ;
3. }
```

La séquence de lexèmes produite par un analyseur lexical est néanmoins identique pour les deux portions de code.

0. LITERAL_int LBRACK RBRACK IDENT ASSIGN LCURLY NUM_INT COMMA NUM_INT COMMA NUM_INT COMMA NUM_INT RCURLY SEMI

1. LITERAL_for LPAREN LITERAL_int IDENT ASSIGN NUM_INT SEMI IDENT LT IDENT DOT IDENT SEMI IDENT INC RPAREN LCURLY

2. IDENT LBRACK IDENT RBRACK ASSIGN IDENT LBRACK PLUS NUM_INT SEMI

3. RCURLY

Concrètement, les identifiants des tableaux A et B et les identifiants des indices i et j ont été remplacés par le même lexème *IDENT*. Les deux portions de code peuvent donc être considérées comme similaires. Il est également utile de spécifier que les espaces, commentaires et casse des caractères ne sont pas traités lors de l'analyse lexicale.

La lexèmisation permet donc clairement d'éliminer les transformations des niveaux L0 à L2 (changements apportés aux commentaires et/ou aux identifiants) du spectre de Faidhi et Robinson[7]. Il est également possible de substituer des structures de contrôle différentes par un même lexème. Par exemple, les instructions de contrôle (while, for, ...) peuvent être substituées par la structure *BEGIN_LOOP. . .END_LOOP* .

Le flux de lexèmes peut également contenir de l'information liée au code source tel que le numéro de ligne relatif au lexème utilisé. Cette information pourra servir à l'utilisateur pour pouvoir comparer visuellement les portions de code classées comme similaires par le moteur de détection de plagiat.

Il faut encore remarquer que, généralement, les lexèmes sont remplacés par des caractères ou par des entiers avant d'être comparés par les algorithmes de détection.

Le fonctionnement général de la phase d'analyse lexicale est commun à tous les moteurs de détection mais l'implémentation peut être particulière

à un moteur. Par exemple, l'algorithme de lexèmisation de l'outil *YAP3*[26] fonctionne de cette manière :

- Supprimer les commentaires et constantes.
- Transformer les majuscules en minuscules.
- Substituer les synonymes par leur forme commune (par exemple, *strncmp* est transformé en *strcmp* et *function* en *procedure*).
- Réordonner les fonctions dans l'ordre de leurs appels. Le premier appel à une fonction est substitué par des lexèmes représentant la séquence complète, les appels suivants sont remplacés par le lexème *FUN*.
- Supprimer tous les lexèmes ne provenant pas du lexique du langage cible, i.e. tous les lexèmes qui ne sont pas des mots réservés.

La production d'une séquence de lexèmes permet donc d'éliminer une partie des transformations lexicales et structurelles qu'un plagiaire pourrait apporter à du code source.

Habituellement, la phase de lexèmisation s'exécute en temps linéaire, O(n) où n est la longueur d'un fichier. Comme les algorithmes de détection ont une complexité plus élevée, la lexèmisation n'a pas un impact majeur sur la complexité générale du système.

Deux inconvénients importants sont cependant à souligner[15]. D'une part, un analyseur lexical est dépendant du langage utilisé, ce qui nécessite un travail supplémentaire lors de l'ajout d'un nouveau langage supporté par l'outil de détection de plagiat. D'autre part, le fait de fournir une séquence de lexèmes à l'algorithme de détection de plagiat rend celui-ci plus paranoïaque puisque la lexèmisation augmente le degré de similitude existant entre deux portions de code source.

3.4 Techniques basées sur la génération d'empreintes

Les techniques basées sur la génération d'empreintes permettent une représentation plus compacte du code source afin d'effectuer une comparaison. Cette représentation est appelée empreinte et constitue un condensé d'infor-

mations caractéristiques du code analysé. Deux types d'empreintes peuvent être générées. Les empreintes quantitatives, assimilées aux systèmes de comptage d'attributs, sont calculées à partir de métriques représentant certaines propriétés du code source.

Les empreintes qualitatives sont, quant à elles, générées à partir de valeurs calculées par une fonction de hachage. Plus la quantité d'empreintes communes aux fichiers comparés est importante, plus la probabilité de plagiat est forte. Contrairement aux empreintes quantitatives, la génération d'une empreinte qualitative permet de capturer la structure complète du document.

3.4.1 Empreintes quantitatives

La génération d'empreintes quantitatives est la technique la plus ancienne utilisée pour la détection de plagiat de code source. Comme expliqué dans la section traitant des systèmes à comptage d'attribut, l'idée générale de cette approche est de comptabiliser certaines caractéristiques ou métriques du code source à analyser. Une mesure de distance est ensuite appliquée sur ces métriques, permettant la génération d'un score reflétant la similarité des programmes étudiés.

La distance utilisée (d) pour mesurer la similarité doit respecter les propriétés classiques d'une distance :

- $d(a,b) \geq 0$
- $d(a,b) = 0$ si et seulement si $a=b$
- $d(a,b) = d(b,a)$ (symétrie)
- $d(a,c) \leq d(a,b) + d(b,c)$ (inégalité triangulaire)

Cette approche présente l'avantage d'être facile à implémenter mais offre des résultats peu persuasifs. Des études empiriques réalisées sur les techniques de génération d'empreintes quantitatives ont largement démontré que ces systèmes ne sont pas adaptés pour la détection de plagiat de code source[22, 12].

Même si cette technique n'est plus utilisée par les moteurs de détection de plagiat modernes, elle présente toutefois un intérêt chronologique. C'est pourquoi trois des métriques les plus utilisées par cette approche sont présentées brièvement dans ce chapitre.

Volume

Le volume peut être quantifié en utilisant les métriques de Halstead[17] et représente la taille de l'implémentation d'un algorithme. L'idée est d'identifier et de compter les opérandes et opérateurs dans le code étudié. Halstead définit les opérandes (table 3.1) comme des variables et des constantes tandis que les opérateurs (table 3.2) sont déterminés comme des symboles ou une combinaison de symboles.

IDENTIFIER	Tous les identifiants
TYPENAME	Mots réservés spécifiant le type : bool, byte, int, float, char, double, long, short, signed, unsigned, void
CONSTANT	Constantes numériques, de caractères ou de chaînes de caractères

TABLE 3.1 – Liste des opérandes de Halstead en Java

TYPE_QUAL	Qualificateurs de type : const, friend, volatile, final, transient
RESERVED	Mots réservés : break, case, continue, default, do, if, else, enum, for, goto, if, new, return, asm, operator, private, protected, public, sizeof, struct, switch, union, while, this, namespace, namespace, using, try, catch, throw, throws, finally, strictfp, instanceof, interface, extends, implements, abstract, concrete, const_cast, static_cast, dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename
OPERATOR	! != % %= & && || &= ( ) { } [ ] * *= + ++ += , - – -= -> . ... / /= : : : < << <<= <= = == > >= >> >>> >>= >>>= ? ^ ^= | |= ~ ; =& “ “ ‘ ‘ # ## ~

TABLE 3.2 – Liste des opérateurs de Halstead en Java

Le volume (V) est donc calculé comme suit :

$V = N.log_2(n)$

avec :

n_1, le nombre d'opérateurs uniques
n_2, le nombre d'opérandes uniques
N_1, le nombre total d'opérateurs
N_2, le nombre total d'opérandes
$n = n_1 + n_2$, la taille du vocabulaire
$N = N_1 + N_2$, la longueur du programme

Flux de contrôle

Le flux de contrôle se réfère à l'ordre dans lequel les instructions d'un programme sont évaluées et exécutées. La complexité cyclomatique de McCabe mesure le nombre de chemins d'exécution d'un programme et donne une indication de la complexité du code analysé[18]. La métrique de McCabe ($V(G)$) est calculée à partir d'un graphe de flux de contrôle (figure 3.1).

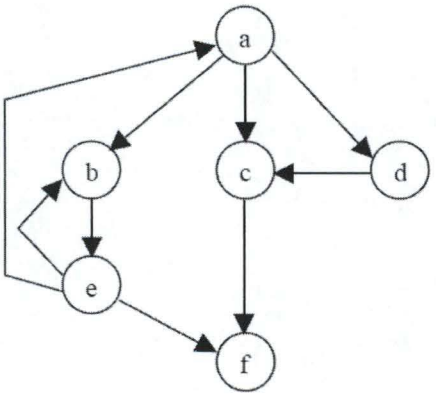

FIGURE 3.1 – Exemple de graphe de flux de contrôle

Dans ce graphe, les blocs de code d'instructions séquentielles sont représentés par des noeuds et les chemins d'exécution par des arêtes, chaque noeud devant être accessible à partir du noeud d'entrée et le noeud de sortie devant être accessible à chaque noeud. Pour construire un tel graphe, McCabe propose une liste des constructions utilisées dans un programme (figure 3.2).

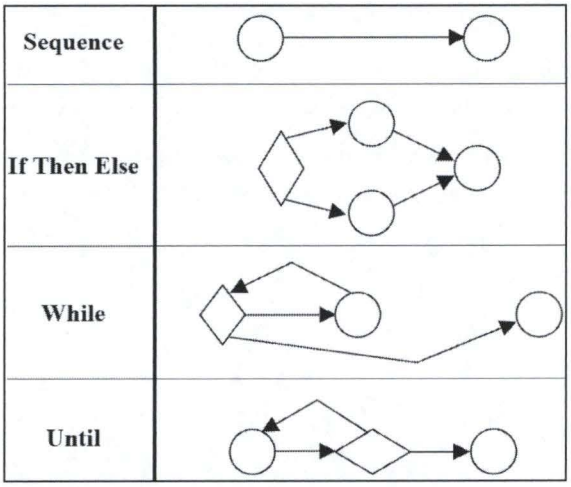

FIGURE 3.2 – Liste des constructions proposées par McCabe

Une fois le graphe réalisé (G), il devient possible de calculer le volume ($V(G)$) :

$$V(G) = e - n + 2p$$

avec

e = le nombre d'arêtes du graphe G

n = le nombre de noeuds du graphe G

p = le nombre de composants connectés du graphe G

Niveau d'imbrication

Le niveau d'imbrication[27] moyen d'un programme est une mesure simple qui se calcule en assignant à chaque ligne de code une valeur indiquant son niveau d'imbrication. Cette mesure se calcule de la manière suivante :

– Le niveau 1 est assigné à la première instruction exécutable.

- Si l'instruction a est au niveau 1 et que l'instruction b suit simplement de manière séquentielle l'exécution de l'instruction a, alors le niveau d'imbrication de l'instruction b est de 1 également.
- Si l'instruction a est au niveau 1 et et que l'instruction b se trouve dans la portée d'une boucle ou d'une instruction conditionnelle contrôlée par l'instruction a, alors le niveau d'imbrication de l'instruction b est incrémenté de 1.

La somme de tous les niveaux d'imbrication calculés pour l'ensemble des instructions est divisé par le nombre total des instructions pour produire le niveau moyen d'imbrication.

3.4.2 Empreintes qualitatives

La génération d'empreinte (*fingerprint*) de documents est une technique utilisée dans de nombreux domaines informatiques. Une empreinte représente un condensé de l'information contenue dans un document et identifie celui-ci de manière unique. La génération de l'empreinte est effectuée en utilisant une fonction de hachage qui crée une suite de nombres représentant le contenu du fichier, appelée valeurs de hachage. Par exemple, dans les problèmes de sécurité informatique, une empreinte peut être générée sur un document de manière à garantir son intégrité. En expédiant un message accompagné de sa valeur de hachage, le destinataire peut vérifier que le message n'a pas été altéré (intentionnellement ou de manière fortuite) durant la communication. Lors de la réception du message, il suffit au destinataire de calculer le haché du message reçu et de le comparer avec le haché accompagnant le document. Si le message (ou le haché) a été falsifié durant la communication, les deux empreintes ne correspondront pas.

Les techniques de génération d'empreintes par fonction de hachage peuvent également être utilisées pour la détection de plagiat de documents. Ces documents peuvent contenir aussi bien du texte que du code source. Pour chaque document à traiter, une empreinte est générée à partir d'une fonction de

hachage. Ensuite les empreintes sont comparées et les documents dont l'empreinte est similaire sont alors répertoriés comme potentiellement plagiés.

La majorité des techniques de génération d'empreintes utilisent la notion de k-gram, un k-gram est une sous-chaîne de caractères de longueur k. Chaque document à comparer doit d'abord être considéré comme une suite de caractères dans une forme canonique. Le document à analyser subit donc un pré-traitement consistant à enlever toute forme de formatage, de ponctuation ou de casse de caractères. Dans le cas de fichiers de code source, cette transformation est effectué par la phase de lexèmisation, vue précédemment.

Les documents sont ensuite découpés en une séquence de k-grams et une fonction de hachage est utilisée sur chaque k-gram pour produire une valeur de hachage.

L'idée sous-jacente de cette technique est que si deux documents partagent une ou plusieurs valeurs de hachage alors ces documents partagent probablement un ou plusieurs k-grams, et présentent donc des similitudes.

Par exemple[28], la forme canonique obtenue sur la phrase '*A do run run run, a do run run*' serait :

adorunrunrunadorunrun

Si la valeur de k vaut 5 alors le découpage en 5-grams de la forme canonique de la phrase de départ donne :

adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun

Une fonction de hachage est alors appliquée sur chaque k-gram obtenu :

77 72 42 17 98 50 17 98 8 88 67 39 77 72 42 17 98

Deux difficultés sont néanmoins à souligner. La première concerne la

fonction de hachage utilisée. Celle-ci doit être choisie de manière à ce que la probabilité de collision soit faible. Une collision survient quand deux k-grams différents produisent la même valeur de hachage, ce qui risquerait d'augmenter le nombre de faux positifs détectés. De plus, pour des raisons évidentes de performance, les valeurs de hachage doivent être rapidement calculées. L'algorithme de *Karp-Rabin*[29] solutionne ce problème en utilisant une fonction de hachage incrémentale qui permet de calculer des valeurs de hachage linéairement tout en garantissant une faible probabilité de collisions.

La deuxième difficulté concerne le nombre important de valeurs de hachage générées. En effet, le découpage du document en k-gram produit pratiquement autant de k-grams (et donc de valeurs de hachage) que de caractères contenus dans le document. Pour des raisons d'efficacité, seulement un sous-ensemble des valeurs de hachage doit être sélectionné pour pouvoir effectuer une comparaison d'empreintes. Cette sélection doit impérativement s'effectuer en garantissant la capture de l'information nécessaire à la détection de similitudes. La technique de sélection d'empreinte par l'algorithme *winnowing* (tamisage) utilisée par Aiken dans l'outil de détection de plagiat de code source *MOSS (Measure Of Similarity Software)* garantit cette propriété.

La suite de cette section présente l'algorithme de *Karp-Rabin* [29] qui permet de calculer les valeurs de hachage pour les k-grams en temps constant. La sélection de ces valeurs est ensuite expliquée avec l'algorithme de sélection *winnowing*.

3.4.2.1 Algorithme de *Karp-Rabin*

L'algorithme de Karp et Rabin est apparu en 1987 dans les applications bio-informatiques et plus particulièrement dans la comparaison de séquences ADN. Le problème était de trouver l'occurrence d'une chaîne de caractères, le motif, dans une chaîne de caractères plus longue, le texte. Plutôt que d'utiliser une comparaison naïve, c'est à dire caractère par caractère, Karp et Rabin exploitèrent le fait qu'une comparaison de valeurs de hachage (condensé des chaînes de caractères à comparer) est plus rapide qu'une comparaison des chaînes de caractères dans leurs intégralités. La conversion des caractères

vers des entiers est effectuée grâce à une fonction de hachage et la comparaison des valeurs de hachage se base sur le fait que si deux chaînes de caractères partagent une même valeur de hachage alors ces chaînes *peuvent* être identiques. A l'inverse, si ces deux chaînes possèdent des valeurs de hachage différentes, alors elles ne sont définitivement pas égales.

Une valeur de hachage est donc calculée pour le motif $p[1..m]$. Grâce à un mécanisme de fenêtre glissante de gauche à droite, une valeur de hachage est similairement calculée pour chaque sous-chaîne de caractères de longueur m dans le texte$t[1..n]$ et comparée avec la valeur de hachage calculée pour le motif. Si les valeurs testées sont égales, une comparaison caractère par caractère est effectuée dans le but d'éviter une collision, i.e. deux chaînes de caractères différentes mais partageant la même valeur de hachage.

Algorithm 3.1 Pseudo-code de l'algorithme *Karp-Rabin*

hash_p = valeurs de hachage du motif p
hash_t = valeur de hachage des m premiers caractères du texte t
Faire
 Si (hash_p = = hash_t) alors comparaison caractère par caractère
 hash_t = valeur de hachage de la prochaine sous-chaîne de t
Tant que (fin du texte t)

Pour éviter de calculer n fois la valeur de hachage des sous-chaînes de caractères du texte $t[1..n]$, l'algorithme utilise une simple relation de récurrence, appelée fonction de hachage incrémentale.

Si une sous-chaîne de caractères de longueur m est considérée comme un nombre de longueur m en base b, où b est le nombre de lettres de l'alphabet utilisé, alors, en utilisant la règle de Horner, cette sous-chaîne peut être définie par :

$$x(i) = t[i].b^{m-1} + t[i+1].b^{m-2} + ... + t[i+m-1]$$

A partir de $x(i)$ il devient possible de calculer $x(i+1)$pour la prochaine sous-chaîne *t[i+1..i+m]* en temps constant :

$$x(i+1) = t[i+1].b^{m-1} + t[i+2].b^{m-2} + ... + t[i+m]$$
$$x(i+1) = x(i).b - t[i].b^{m} + t[i+m]$$

De cette manière, la valeur de hachage existante est simplement ajustée lorsque la fenêtre glisse d'un caractère vers la droite et donc, la nouvelle valeur de hachage n'est jamais explicitement calculée.

Par exemple, si les chaînes de caractères à comparer sont constituées des dix premières lettres de l'alphabet et que la valeur d'un caractère est déterminée par sa position dans l'alphabet, la base vaut donc 10.

a	b	c	d	e	f	g	h	i	j
1	2	3	4	5	6	7	8	9	10

Si le motif est la chaîne de caractères 'cah' est à rechercher dans le texte 'abdcahefg'. La valeur du motif se calcule par :

$$3.10^{3-1} + 1.10^{3-2} + 8.10^{3-3} = 318$$

L'exécution de l'algorithme se poursuit en calculant les valeurs de hachage pour les sous-chaînes comprises dans le texte.

Lorsque la fenêtre glisse de 'dca' à 'cah', la valeur de 'dca' est de 431,

a	b	***d***	***c***	***a***	h	e	f	g
			413					

->

a	b	d	***c***	***a***	***h***	e	f	g
				318				

et la relation de récurrence définie plus haut permet de déduire la valeur de hachage de 'cah' à partir de 431 :

$$431.10 - 4.10^3 + 8 = 318$$

Cependant, si m est grand, alors les valeurs manipulées risquent d'être énormes. La fonction *modulo* q est dès lors appliquée sur toutes les opérations mathématiques, q étant un nombre premier suffisamment grand pour éviter les dépassements de capacité d'entier et également réduire le risque de collisions. La fonction de hachage peut donc être définie par :

$$h(i+1) = (h(i).b - t[i].b^m + t[i+m]) \; mod \; q$$

Dans le pire des cas, la complexité en temps de l'algorithme de *Karp-Rabin* est de O(mn) avec m la longueur du motif et n la longueur du texte. Cette possibilité est cependant très peu probable et n'est rencontrée que si toutes les valeurs de la fenêtre sont identiques à la valeur du motif, dès lors que la comparaison caractère par caractère doit être effectuée pour toutes les valeurs. En pratique, le temps consacré est O(m+n).

3.4.2.2 Algorithme de sélection locale *winnowing*

Comme expliqué précédemment, il n'est pas envisageable de conserver toutes les valeurs de hachage produites sur les k-grams d'un document pour générer une empreinte. Par conséquent, un algorithme sélectionnant un sous-ensemble de ces empreintes doit être utilisé. Plusieurs approches de sélections existent. La plus populaire est de sélectionner la i^{eme} valeur de hachage de l'ensemble des valeurs générées. Cette stratégie apparaît tout de suite incorrecte puisqu'elle ne résiste pas à l'insertion, la suppression et/ou le réordonnancement de code. Par exemple, l'ajout d'un simple caractère à un fichier de code source ferait glisser d'une position tous les k-grams, conséquemment l'empreinte du code source original ne correspondrait plus avec l'empreinte du fichier modifié.

Une autre manière de procéder[10] est de choisir les valeurs de hachage tel que 0 modulo p (p étant une constante), i.e. de ne conserver que les valeurs de hachages divisibles par p. Le sous-ensemble des valeurs sélectionnées

est donc constitué de $1/p$ de l'ensemble des valeurs de hachage produites.

Par exemple, avec une sélection 0 modulo 4 sur les valeurs de hachage suivantes :

77 72 42 17 98 50 17 98 8 88 67 39 77 72 42 17 98

L'empreinte obtenue est :

72 8 88 72

Le désavantage de cette sélection est que la distance séparant deux valeurs de hachage peut être importante, dès lors la quantité d'information capturée par l'empreinte n'est pas assez représentative du code source analysé, ce qui aurait comme conséquence que des similitudes importantes entre deux fichiers pourraient passer inaperçues. Lors de tests de sélection d'empreintes générées sur des pages *HTML*, Schleimer, Wilkerson et Aiken démontrent que la distance entre deux valeurs de hachage sélectionnées *0* modulo p , avec $p = 50$, peut être plus importante que la taille moyenne d'une page *HTML*.

Pour remédier à ce genre de problème, Schleimer, Wilkerson et Aiken proposent une nouvelle méthode de sélection d'empreintes grâce à l'algorithme *winnowing*. Celui-ci garantit, pour une collection de documents à comparer, deux propriétés :

- Si une *correspondance* caractérise le fait que plusieurs documents partagent une chaîne de caractères et que la longueur de cette correspondance est aussi longue que le *seuil de garantie* t alors cette *correspondance* est détectée.

- Les *correspondances* de longueurs inférieures au *seuil de bruit* k ne sont pas détectées, cette propriété est immédiatement garantie par le fait

que seul les k-grams sont considérés. Il est primordial de choisir une valeur appropriée pour la constante k. Plus la valeur de k augmente et plus la probabilité de *correspondances* accidentelles diminue, c'est à dire la détection de mots communs au langage dans lequel sont écrits les documents à comparer. Les *correspondances* de petites longueurs sont à proscrire puisqu'elles augmentent la détection de faux positifs, i.e. des documents marqués comme plagiés alors qu'ils ne le sont pas. A l'inverse, choisir pour k une valeur trop importante implique une diminution de la sensibilité au réordonnancement du texte (ou du code source), puisque les chaînes de caractères plus petites que k ne sont plus détectées. Schleimer, Wilkerson et Aiken affirment avoir choisi un seuil de bruit égal à 50 pour leurs expérimentations de leur algorithme sur des pages Web, cependant aucune précision n'est donnée sur la valeur à donner à k pour la détection de plagiat de code source dans l'outil *MOSS*.

L'algorithme *winnowing* utilise un mécanisme de fenêtre glissante de longueur w pour sélectionner les empreintes. Dès lors, si la taille de la fenêtre est fixée à $w = t - k + 1$ et que chaque position $1 \leq i \leq n - w + 1$ de la séquence de valeurs de hachage générée par document définit une fenêtre de valeurs de hachage $h_i ... h_{i+w-1}$ alors il est suffisant de sélectionner une valeur de chaque fenêtre pour constituer l'empreinte du document. Pour chaque fenêtre, la plus petite valeur de hachage trouvée est sélectionnée, si plusieurs occurrences minimales sont trouvées alors l'algorithme sélectionne l'occurrence se trouvant le plus à droite de la fenêtre. Il est important de remarquer que le nombre de valeurs sélectionnées est beaucoup moins important que le nombre de fenêtres, ce qui est dû au fait que la même valeur est sélectionnée par plusieurs fenêtres se chevauchant.
En reprenant l'exemple précédemment utilisé :

A do run run run, a do run run

adorunrunrunadorunrun

adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun

77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98

Lorque la taille de la fenêtre (w) est fixée à 4, les fenêtres de hachage de longueurs égales à w sont obtenues par :

(77, 74, 42, ***17****) (74, 42, 17, 98)*
(42, 17, 98, 50) (17, 98, 50, ***17****)*
(98, 50, 17, 98) (50, 17, 98, ***8****)*
(17, 98, 8, 88) (98, 8, 88, 67)
(8, 88, 67, ***39****) (88, 67, 39, 77)*
(67, 39, 77, 74) (39, 77, 74, 42)
(77, 74, 42, ***17****) (74, 42, 17, 98)*

Chaque valeur sélectionnée est montrée en gras, mais seulement dans la première fenêtre qui sélectionne la plus petite valeur de hachage. Lors de chaque avancement de la fenêtre vers la droite, l'algorithme ne sélectionne pas nécessairement une nouvelle valeur de hachage. La sélection d'une nouvelle valeur de hachage intervient uniquement lorsque la valeur minimale actuelle sort de la fenêtre ou lorsque la nouvelle valeur introduite est plus petite que toutes les valeurs contenues dans la fenêtre courante.

Les valeurs sélectionnées deviennent donc :

17 17 8 39 17

L'information contenant la position des valeurs de hachage dans le document traité (l'index des positions commence à 0) est également mémorisée, ce qui est utile pour représenter les portions de code source considérées comme similaires à l'utilisateur.

[17,3] [17,6] [8,8] [39,11] [17,15]

Les auteurs de *winnowing* généralisent leur algorithme en introduisant la famille d'algorithme de sélection locale dont *winnowing* est un membre. Un algorithme de sélection locale est caractérisé par l'utilisation d'une fenêtre glissante pour réaliser la sélection des valeurs de hachage et base son choix uniquement sur le contenu de cette fenêtre, d'où l'épithète de locale. Pour évaluer leur algorithme, ces derniers analysent la densité de *winnowing*. La densité d'un algorithme de génération d'empreintes est égale au rapport entre le nombre d'empreintes sélectionnées et le nombre total d'empreintes calculées pour un document. La densité s'exprime donc comme un compromis entre le seuil de garantie et le nombre d'empreintes sélectionnées, l'objectif étant de minimiser la densité tout en maintenant un seuil de garantie élevé. Schleimer, Wilkerson et Aiken montrent que la densité moyenne théorique de *winnowing* est de $d = \frac{2}{w+1}$ où w exprime la taille de la fenêtre. Ceux-ci montrent également qu'un algorithme de sélection locale obtient une densité de $d = \frac{1,5}{w+1}$, ce qui prouve que l'algorithme *winnowing* n'est pas optimal, sa densité n'étant pas parfaite, et que de nouvelles méthodes de sélection d'empreintes plus performantes restent à découvrir.

L'algorithme *winnowing* est implémenté par Aiken dans le moteur de détection de plagiat de code source *MOSS*[30]. *MOSS* accepte une collection de fichiers de code source à analyser et retourne un ensemble de pages *HTML* montrant les portions de code source considérées comme similaires. Les fichiers à analyser sont tout d'abord convertis en une chaîne de caractères standard via une phase de lexèmisation. Cette chaîne de caractères est ensuite passée en paramètre à l'algorithme générant les empreintes.

La première étape de l'algorithme consiste à construire un index qui fait correspondre chaque empreinte générée avec sa position dans les fichiers de code source. Par exemple, si l'ensemble des empreintes générées pour le fichier *a* correspond à *[17,3] [17,6] [8,8] [39,11] [17,15]* et que l'ensemble des empreintes générées pour le fichier *b* correspond à *[9,3] [17,5] [23,8] [56,13] [17,16]* alors l'index généré correspond au tableau ci-dessous :

Empreinte	Fichier(s) et position
8	a(8)
9	b(3)
17	a(3,6,15) b(5,16)
23	b(8)
39	a(11)
56	b(13)

Ensuite, pour chaque fichier, l'algorithme construit une liste de paires d'empreintes correspondant avec des empreintes de valeurs égales trouvées pour les autres fichiers. Dans le cas de l'exemple utilisé, l'empreinte de valeur *17* est commune aux fichiers a et b et la paire générée pour le fichier a correspond à ***a(17[3,6,17],b 17[5,16])***.

Ces paires sont ensuite ordonnées par nombre d'empreintes partagées et sont affichées. Puisque ces paires contiennent également les positions des empreintes sélectionnées, l'utilisateur a la possibilité de visionner et de comparer les portions de code source analysées comme similaires.

Complexité

Aucune précision n'est donnée par Aiken quant à la complexité de son moteur de détection *MOSS*. Cependant même si la génération et la sélection des empreintes se fait en temps constant O(nN), où n est la taille du fichier de code source et N le nombre de fichiers, l'étape consistant à générer et à classer les paires de documents s'exécute en $O(N^2)$.

3.5 Techniques basées sur la comparaison de chaînes de caractères

Les techniques basées sur la comparaison de chaînes de caractères caractérisent le fait que des programmes potentiellement plagiés partageront des chaînes de caractères plus ou moins longues. A l'instar des techniques

basées sur la génération d'empreintes qualitatives, ces techniques appliquent une phase de lexèmisation sur les fichiers à comparer. Ces fichiers sont donc typiquement convertis en une séquence de lexèmes. Ce type de comparaison peut fournir aussi bien des résultat qualitatifs, en général un score de similarité, que des résultats quantitatifs, par exemple une représentation visuelle des portions de code considérées comme plagiées.

Plus formellement, la similarité entre deux programmes P et Q peut être définie par[31] :

Soit un programme P constitué des éléments $p_1, p_2, ..., p_n$. Les éléments sont les lexèmes résultants de la phase de pré-traitement. P est représenté comme l'ensemble contenant ces éléments soit $P = \{p_1, p_2, ..., p_n\}$.

Soit un programme Q constitué des éléments $q_1, q_2, ..., q_m$. De la même manière, $Q = \{q_1, q_2, ..., q_m\}$.

Un *match (correspondance)* entre $p_i, q_j (0 \leq i < n, 0 \leq j < m)$, i.e. un lexème est commun à P et Q, est noté (p_i, q_j). L'ensemble des *matches* est appelé la *correspondance* Rs où $Rs \subseteq P \times Q$.

La similarité S entre le programme P et le programme Q est donc définie par :

$$S \equiv \frac{\mid \{\exists\, p_i \mid (p_i, q_j) \in Rs\} \mid + \mid \{\exists\, q_j \mid (p_i, q_j) \in Rs\}}{|P| + |Q|}$$

avec $|P|$ le nombre de lexèmes contenus dans P et $|Q|$ le nombre de lexèmes contenus dans Q.

La mesure de similarité S reflète donc la portion de lexèmes partagés par

les deux programmes et est calculée en utilisant un algorithme de comparaison de chaînes de caractères. La suite de cette section présente l'algorithme le plus utilisé dans les moteurs de détection de plagiat de code source : l'algorithme *Running-Karp-Rabin-Greedy-String-Tiling* implémenté notamment par le système *JPlag[6]* .

3.5.1 Algorithme *Running-Karp-Rabin Greedy-String-Tiling (RKS-GST)*

A l'origine, l'algorithme *Greedy String Tiling* fût implémenté par Wise pour le système *Neweyes*[32]. *Neweyes* est un programme utilisé pour aligner des bio-séquences de nucléotides ou d'acides aminées. Ce système est capable de détecter des sous-chaînes d'une bio-séquence réarrangées dans une autre bio-séquence, tout en garantissant un temps d'exécution linéaire. Par la suite, Wise utilisera l'algorithme pour l'implémenter dans son propre moteur de détection de plagiat de code source *YAP3*[26].

Plusieurs moteurs de détection de plagiat actuels utilisent l'algorithme de Wise. On peut citer entre autres *Jplag*[6], *Plaggie*[33] et *Sherlock.*[8].

De nombreux algorithmes existent pour détecter la transposition d'une chaîne de caractères en une autre. Les deux les plus employés pour obtenir une mesure de similarité sont la *distance de Levenshtein* et l'algorithme *LCS (Longest Common Subsequences)[34].*

Par exemple :

Soit $T = PortezCeVieuxWhiskyAuJugeBlondQuiFume$, le texte.
Soit $P = WhiskyVieux$, le motif à rechercher dans le texte.

L'algorithme *LCS* trouve la longueur maximum de la sous-chaîne commune à T et P. Ainsi, dans l'exemple, *LCS* renvoie une longueur de 6. Cette valeur est la longueur de la sous-chaîne *Whisky* qui est la plus longue sous-chaîne partagée par T et P. En revanche, la distance de Levenshtein trouve le nombre minimal de caractères qu'il faut supprimer, insérer, ou remplacer

pour transformer T en P. Des chaînes identiques obtiennent une *distance de Levenshtein* égale à zéro. Dans l'exemple présenté, la *distance de Levenshtein* sera la longueur de T sans la plus longue sous-chaîne *Whisky*, qui donne 37 - 6 = 31.

Ces deux techniques présentent un inconvénient majeur pour la détection de plagiat car elles préservent l'ordre des sous-chaînes. Il est clair, que dans l'exemple proposé, *WhiskyVieux* est une version transposée de la sous-chaîne *VieuxWhisky*. L'algorithme *LCS* et la *distance de Levenshtein* trouveront seulement la sous-chaîne *Whisky* et pas la version transposée complète, *VieuxWhisky*. Or, la transposition est un phénomène commun dans la détection du plagiat. Par conséquent, l'algorithme *Greedy String Tiling* [14] a été implémenté pour résoudre ces problèmes de transposition liés à l'utilisation de l'algorithme *LCS* et de la *distance de Levenshtein*.

L'algorithme *GST* appliqué sur l'exemple précédemment décrit, détermine que les chaînes *Whisky* et *Vieux* sont communes et donc que la longueur de la plus longue chaîne de caractères commune à T et à P est de 11.

Selon [34, 6], un algorithme approprié de comparaison de séquences de caractères possède les caractéristiques suivantes :

- Chaque caractère de la séquence T ne peut correspondre qu'une seule fois avec un caractère de la séquence P. Ceci implique que les doublons du code source sont ignorés dans le programme plagié.
- La position d'une sous-chaîne devrait avoir un effet minimal sur la valeur de similitude. Ceci implique que le réarrangement du code source n'affecte pas les résultats de l'algorithme de comparaison.
- Une *correspondance* (*match*) indique le fait qu'une sous-chaîne de caractère soit partagée par T et P. Pour des raisons de fiabilité, les *correspondances* les plus longues sont préférées aux *correspondances* plus courtes.

L'algorithme *GST* fonctionne en deux phases (algorithme 3.2) :

- Dans la phase initiale de l'algorithme, les plus longues *correspondances* contiguës sont recherchées. Les deux premières boucles effectuent une comparaison caractère par caractère sur les deux chaînes A et B. Lorsqu'un caractère commun est trouvé, la troisième boucle tente d'étendre la *correspondance* aussi loin que possible. La première phase de l'algorithme collecte l'ensemble des plus longues sous-chaînes de caractères communes aux deux chaînes A et B.
- La deuxième phase marque toutes les *correspondances* trouvées lors de la première phase. Ceci signifie que tous les caractères appartenant à ces sous-chaînes sont marqués (par exemple à l'aide d'un changement de valeur d'une variable booléenne associée à ce caractère) et qu'ils deviennent indisponibles pour les futures *correspondances* de la première phase. Dans la terminologie de Wise, le fait de marquer ces caractères revient à la formation d'un *recouvrement* (*tile*).

Ces deux phases sont ensuite répétées jusqu'à ce qu'il ne soit plus possible de trouver de *correspondances*. L'algorithme est garantit de se terminer puisque la longueur des *correspondances* trouvées décroît de 1 à chaque étape. Pour éviter que des *correspondances* trop petites et donc non fiables ne soient trouvées, une limite inférieure pour la longueur d'une *correspondance* est déterminée. Cette valeur est appelée *MinimumMatchLength*. Les tests empiriques réalisés sur *JPlag* utilisent des *MinimumMatchLength* de valeurs de 3, 4, 5, 7, 9, 11, 14, 17, 20, 25, 30, 40 et démontrent que les valeurs les plus robustes se situent entre 7 et 11, où une *MinimumMatchLength* de valeur égale à 9 apparaît comme étant le meilleur paramètre expérimenté[6].

L'algorithme 3.2 montre le pseudo-code de l'algorithme *GST*. L'opérateur $\oplus$ utilisé à la ligne 12 indique qu'une *correspondance* est ajouté à l'ensemble des *correspondances* (*matches*). Le triplet *match(a, b, l)* définit une *correspondance*, i.e. une sous chaîne commune aux chaînes A et B, commençant aux position A_a et B_b de longueur l.

La première phase (lignes 5 à 18) recherche pour chaque caractère A_a non marqué de la chaîne A, un caractère identique B_b non marqué dans la

Algorithm 3.2 Pseudo Code de l'algorithme *Greedy String Tiling*

```
0 Greedy-String-Tiling(String A, String B) {
1   tiles = {};
2   do {
3       maxmatch = MinimumMatchLength;
4       matches = {};
5       Forall unmarked tokens Aa in A {
6         Forall unmarked tokens Bb in B {
7         j= 0;
8         while (Aa+j == Bb+j &&
9               unmarked(Aa+j ) && unmarked(Bb+j ))
10          j ++;
11        if (j == maxmatch)
12          matches ⊕ matches match(a, b, j);
13        else if (j > maxmatch) {
14         matches = {match(a, b, j)};
15         maxmatch = j;
16        }
17       }
18      }
19    Forall match(a, b, maxmatch) ∈ matches {
20      For j = 0...(maxmatch - 1) {
21       mark(Aa+j );
22       mark(Bb+j );
23      }
24    tiles = tiles ∪ match(a, b, maxmatch);
25     }
26    }while (maxmatch > MinimumMatchLength);
27   return tiles;
28 }
```

chaîne B. Si une *correspondance* est trouvée, celle-ci est étendue au maximum. Dès qu'un caractère diffère ou est considéré comme marqué alors la *correspondance* est interrompue et sa longueur est mémorisée dans la variable j. Ensuite si cette *correspondance* de longueur j est aussi longue que les *correspondances* trouvées précédemment, celle-ci est ajoutée à l'ensemble des *correspondances*. Si la *correspondance* de longueur j est la plus longue des *correspondances* alors l'ensemble des *correspondances* est réinitialisé pour ne contenir que cette *correspondance* et la valeur de j est considérée comme la valeur de la plus longue *correspondance*. Après cette phase, l'ensemble *matches* contient toutes les *correspondances*. L'ensemble est vide si aucune *correspondance* de longueur supérieure à *MinimumMatchLength* n'a été trouvée.

La deuxième phase commence à la ligne 19. L'ensemble des *correspondances (matches)* est itéré pour marquer chaque caractère appartenant à une *correspondance*. Les *correspondances* marquées sont ensuite ajoutées à l'ensemble des *recouvrements* (*tiles*). L'algorithme itère ensuite sur les deux phases et lorsqu'il n'est plus possible de trouver de *correspondance* de longueur supérieure à *MinimumMatchLength*, celui-ci se termine en renvoyant comme résultat l'ensemble des correspondances marquées (*tiles*).

La similarité reflète donc la portion de caractères communs aux programmes A et B et se calcule par :

$$sim(A,B) = \frac{2.coverage(tiles)}{|A|+|B|}$$

où *coverage(tiles)* est la somme des longueurs des correspondances (*match*) appartenant à l'ensemble des recouvrements (*tiles*).

L'algorithme 3.3 est une implémentation du *Greedy String Tiling* en langage Java. La valeur de *MinimumMatchLength* y est fixée à 3. Avant d'être passées en paramètres à l'algorithme, les chaînes de caractères à comparer sont converties en tableau d'objet Token. L'objet Token maintient la valeur et la position d'un caractère de la chaîne ainsi qu'une variable booléenne

utilisée pour marquer l'objet durant la création des *tiles*. L'ensemble des recouvrements (*tiles*) est implémenté grâce à un objet de type Map qui permet de stocker les correspondances, implémentées par l'objet Match, trouvées par longueur. A la fin de son exécution, l'algorithme affiche les recouvrements (*tiles*) trouvés et calcule la valeur de similarité existant entre les deux chaînes de caractères à comparer.

Lorsque les chaînes *PortezCeVieuxWhiskyAuJugeBlondQuiFume* et *WhiskyVieux* sont passées en paramètres à l'algorithme, les deux premières boucles effectuent la comparaison caractère par caractère. L'algorithme rentre dans la troisième boucle quand le caractère *W* est rencontré dans les deux chaînes. La variable *j* est alors incrémentée tant que les caractères correspondent dans les deux chaînes, i.e. jusqu'à ce que la chaîne *Whisky* soit découverte dans son intégralité. Un objet Match ensuite créé et ajouté à l'ensemble des correspondances (*matches*). Lors de la seconde phase de l'algorithme, un recouvrement est créé en marquant les objets Match instanciés lors de la première phase et ajouté à la liste des recouvrements (*tiles*). La découverte de la chaîne *Vieux* s'exécute de manière similaire. A la fin de son exécution, lorsqu'il n'est plus possible de découvrir de nouvelles correspondances, l'algorithme affiche les recouvrements trouvés et le score de similarité :

Tile : 6 8 5
Tile : 0 13 6
Similarité : 0.4583333333333333

Le premier recouvrement (*Tile : 6 8* 5) correspond à la chaîne *Vieux* de longueur 5 et commençant respectivement à la position 6 dans le motif et à la position 8 dans le texte. Le deuxième recouvrement (*Tile : 0 13 6*) correspond à la chaîne *Whisky* de longueur 6 et commençant respectivement à la position 0 dans le motif et à la position 13 dans le texte. La similarité entre le motif et le texte est dès lors d'environ 45 % ((2*(5+6)/48)*100).

Algorithm 3.3 *Greedy String Tiling*

```
public void greedyStringTiling(Token[]textTokens,
                               Token[]patternTokens){
    int minimalLenght = 3;
    int maxmatche;
    List<Match> matches;
    Map<Integer,List<Match>> tiles =
                new TreeMap<Integer, List<Match>>();
    do {
        matches = new ArrayList<Match>();
        maxmatche = minimalLenght;
        for(int m = 0; m < patternTokens.length; m++){
          for (int n = 0; n < textTokens.length; n++) {
          int j = 0;
          while (((j + m) < patternTokens.length)
                &&((j + n) < textTokens.length)
                && (patternTokens[j + m].getValue()
                == textTokens[j+n].getValue())
                && (!patternTokens[j + m].isMarked())
                && (!textTokens[j + n].isMarked())) {
                  j++;
                }
```

```
            if (j == maxmatche) {
              Match match = new Match(m, n, j);
              matches.add(match);
            }else if (j > maxmatche) {
              Match match = new Match(m, n, j);
              matches.clear();
              matches.add(match);
              maxmatche = j;
            }
          }
        }
        for (int i = 0; i < matches.size(); i++) {
            Match m = (Match) matches.get(i);
            for (int j = 0; j < m.getLenght(); j++) {
              Token t = textTokens[j + m.getStartInText()];
              t.setMarked(true);
              t = patternTokens[j + m.getStartInPattern()];
              t.setMarked(true);
              }
            tiles.put(new Integer(maxmatche), matches);
        }
    }while (maxmatche > minimalLenght);
        double coverage = 0;
        Iterator<List<Match>> it = tiles.values().iterator();
        while (it.hasNext()) {
           List<Match> list = it.next();
           for (int i = 0; i < list.size(); i++) {
              Match m = (Match) list.get(i);
              coverage = coverage + m.getLenght();
              System.out.println("Tile : " + m.toString());
        }
      }
       System.out.println("Similarité : "
            +(2*coverage/(patternTokens.length
            + textTokens.length)));
    }
```

Complexité

Wise[34] démontre que, dans le pire des cas, la complexité de l'algorithme est de $O(n^3)$. Pour simplifier les calculs, on assume que $n = |A| = |B|$. Par exemple, si A = aaaaaaaaaaaaa et que B = aaabaaabaaab, l'algorithme trouve à chaque itération une correspondance plus courte d'un caractère que celle trouvée à l'itération précédente, ce qui signifie que les trois boucles de la première phase de l'algorithme sont exécutées en intégralité.

Dans le meilleur des cas, même si les deux chaînes de caractères à comparer sont complètement différentes, l'algorithme doit comparer au moins chaque caractère de A avec tous les caractères de B, ce qui implique que $|A|.|B|$ comparaisons sont nécessaires et donc un temps d'exécution en $O(n^2)$.

Optimisations

Même si la complexité ne peut être diminuée pour le pire des cas (scénario fort peu probable), quelques transformations peuvent être appliquées au *Greedy String Tiling* pour réduire son temps d'exécution à pratiquement O(n). L'idée, à l'instar de l'algorithme *winnowing*, est d'utiliser des valeurs de hachage calculées pour chaque sous-chaîne de caractères à comparer. Encore une fois, l'algorithme de *Karp-Rabin*[29] est employé pour générer ces valeurs en un temps linéaire.

L'algorithme modifié est appelé *Running-Karp-Rabin-Greedy-String-Tiling* et subit les transformations suivantes :

- Les valeurs de hachage sont calculées pour toutes les sous-chaînes de caractères de longueurs égales à *MinimalMatchLenght*. Ces calculs peuvent être réalisé en temps $O(|A| + |B|)$ et permettent une comparaison plus rapide de ces sous-chaînes.
- Chaque valeur de hachage de A est donc comparée avec chaque valeur de B. Si deux valeurs sont identiques, alors une correspondance possible est trouvée. Pour éviter une collision de valeurs de hachage, cette correspondance est vérifiée caractère par caractère. L'algorithme tente alors d'étendre la correspondance aussi loin que possible en y ajoutant

un caractère à la fois.

- Les valeurs de hachage générées sont manipulées via une table de hachage. Cette structure de données permet une association clé-élément, ce qui permet de réduire l'effort demandé pour localiser une sous-chaîne de caractères en fonction de sa valeur de hachage.

Ces transformations ne permettent pas de diminuer la complexité $O(n^3)$ dans le pire des cas, puisque toutes les sous-chaînes doivent subir une comparaison caractère par caractère. Néanmoins, en pratique un temps d'exécution de moins de $O(n^2)$ est observé. Les tests empiriques effectués par Wise démontrent qu'en général la complexité moyenne est de $O(n^{1.12})$ ce qui reste très proche d'un temps d'exécution linéaire. La complexité totale du système est calculée en temps $O(N^2 n^{1.12})$ puisque N fichiers doivent être comparés.

3.6 Techniques basées sur la comparaison d'arbres syntaxiques

Le premier moteur de détection de plagiat à utiliser la comparaison d'arbres syntaxiques fût vraisemblablement le moteur de détection de plagiat *SIM*[35], développé à l'Université libre d'Amsterdam par Dick Grune. Plutôt que d'analyser directement le code source d'un programme, *SIM* étudie les arbres générés par l'analyse syntaxique de programmes. Ces arbres syntaxiques sont convertis en chaînes de caractères pour être ensuite comparés en utilisant un algorithme d'alignement de chaînes de caractères comparable à celui utilisé dans l'outil *UNIX diff*. Un score de similarité est alors calculé pour chaque paire de programmes analysés. *SIM* est considéré comme une approche hybride puisque les techniques employées sont basées sur la comparaison d'arbres syntaxiques et sur la comparaison de chaînes de caractères.

Le procédé complet de la comparaison d'arbres syntaxiques a été implémenté dans le système *BRASS*[1] développé à l'Université de Tulane. Il est à noter que *BRASS* est un système privé, non disponible en téléchargement. L'algorithme de détection de plagiat utilisé par *BRASS* effectue les cinq opé-

rations suivantes :

1. Représentation graphique du code source

Le code source des programmes à comparer est transformé en ce que les auteurs de *BRASS* qualifient de diagramme de structure. Un tel diagramme permet d'exprimer un fichier de code source à un niveau d'abstraction plus élevé. Un diagramme de structure est constitué de deux composants : un arbre représentant le code source et un dictionnaire répertoriant les variables et structures de données identifiées dans le code source. Les figures 3.3 et 3.4 montrent les diagramme de structure d'un programme original et d'une copie tandis que les dictionnaires correspondants sont représentés à la figure 3.7. La racine de l'arbre est le header du code source, le langage cible de *BRASS* étant le langage *C*, tandis que les noeuds enfants spécifient les instructions et structures de contrôle. Ces noeuds sont ordonnés de la gauche vers la droite pour permettre de capturer l'ordre de séquencement des instructions. Au final, les diagrammes de structure générés sont encodés au format *DaVinci*[36]. DaVinci est un format d'encodage permettant de visualiser des graphes de manière hiérarchique en organisant les noeuds sur des niveaux différents de manière à ce que les noeuds parents se trouvent au-dessus des noeuds enfants.

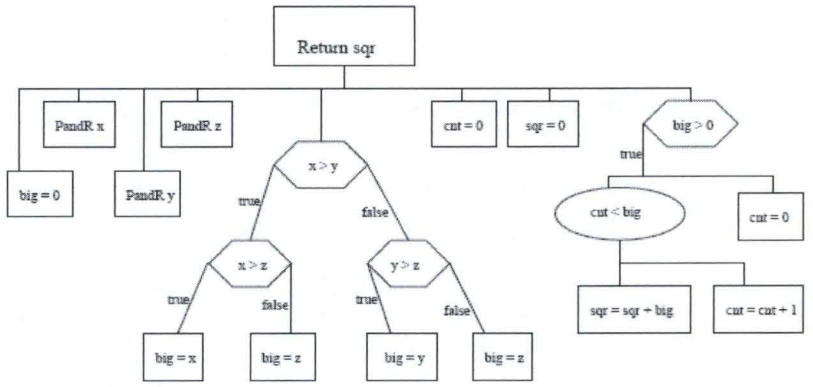

FIGURE 3.3 – Diagramme de structure du programme original [1]

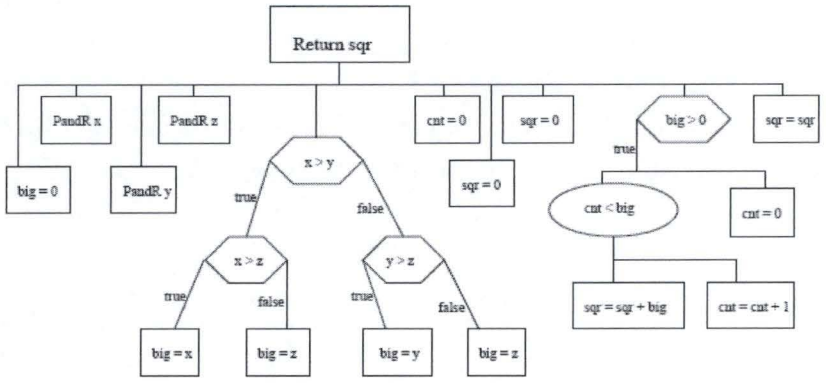

FIGURE 3.4 – Diagramme de structure du programme copié [1]

2. Partitionnement des diagrammes de structure

L'algorithme du système *BRASS* partitionne ensuite récursivement chaque diagramme de structure en trois type de régions : groupe d'instructions sé-

quentielles, instructions de sélection, instructions de répétition. Le résultat de cette phase est appelé arbre de régions. Par exemple[1], le partitionnement de l'ensemble des instructions suivantes

```
while (x < 0){
        y = 5;
        x = x + 1;
        while (y > 0)
          y = y - 1;
        }
```

serait constitué de quatre régions :

- Région 1 (instruction de répétition) pour l'instruction *while (x < 0)*
- Région 2 (instructions séquentielles) pour les instructions $y = 5$ et $x = x + 1$
- Région 3 (instruction de répétition) pour l'instruction *while (y > 0)*
- Région 4 (instruction séquentielle) pour l'instruction $y = y - 1$

L'arbre de régions généré aurait donc la région 1 comme racine, les régions 2 et 3 comme noeuds enfants de la région 1 et la région 4 comme noeud enfant de la région 3.

3. Comparaison des régions

Les noeuds racines de chaque sous-arbre de régions sont ensuite marqués selon leur appartenance à une catégorie syntaxique (répétition, sélection, entrée/sortie, séquence, ...). Une séquence ordonnée de types de régions est alors générée pour chaque arbre de régions en commençant au niveau le plus élevé, le but étant de comparer ces séquences ordonnées de manière à déterminer la plus longue séquence commune à deux arbres de régions. Ce processus est ensuite répété jusqu'à atteindre le niveau le plus bas de

l'arbre des régions. La longueur des séquences communes est alors utilisée pour générer un pourcentage de similarité.

La dernière étape de la comparaison de régions se concentre sur la comparaison des types de noeuds. En utilisant les séquences précédemment générées, les types des paires de noeuds correspondantes sont comparées. A chaque égalité, un compteur est incrémenté de manière à générer un score de similarité reflétant le nombre de noeuds correspondants contre le nombre total de noeuds.

Les figures 3.5 et 3.6 montrent les arbres de régions d'un programme original et d'une copie de celui-ci. Par exemple, les séquences générées par l'algorithme de *BRASS* sur ces arbres à la première itération seraient :

$S_1 = < A_1 : (séquence) ; A_2 : (sélection) ; A_{16} : (séquence) ; A_{17} : (sélection) >$

$S_2 = < B_1 : (séquence) ; B_2 : (sélection) ; B_{16} : (séquence) ; B_{17} : (sélection) ; B_{22} : (séquence) >$

La comparaison des séquences S_1 et S_2 montre que $(A_1, A_2, A_{16}, A_{17})$ correspond à $(B_1, B_2, B_{16}, B_{17})$ et que B_{22} n'a aucune correspondance. La plus longue chaîne de régions correspondantes entre les deux arbres est donc $(B_1, B_2, B_{16}, B_{17})$. A l'itération suivante, comme les régions A_1 et B_1 n'ont pas de sous-régions, aucune comparaison n'est effectuée. Par contre, les séquences pour A_2 et B_2 sont générées et comparées :

$S_{A2} =< A_4 : (sélection) ; A_{10} : (sélection) >$

$S_{B2} =< B_4 : (sélection) ; B_{10} : (sélection) >$

La plus longue séquence (B_4, B_{10}) est alors identifiée et le processus est répété jusqu'à ce que toutes les séquences soient générées et comparées.

Ensuite la comparaison selon le type de noeuds commence. Pour A_1 et B_1, les listes de types de noeuds suivantes sont construites :

A_1 : *(header, assign, I/O, I/O, I/O)*
B_1 : *(header, assign, I/O, I/O, I/O)*

La comparaison de ces deux listes implique donc dans ce cas une similarité totale.

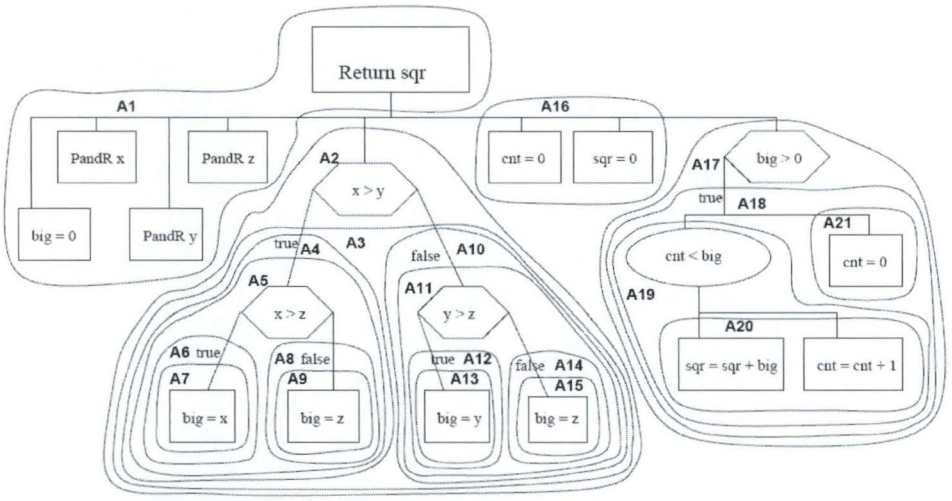

FIGURE 3.5 – Arbre de régions du programme original [1]

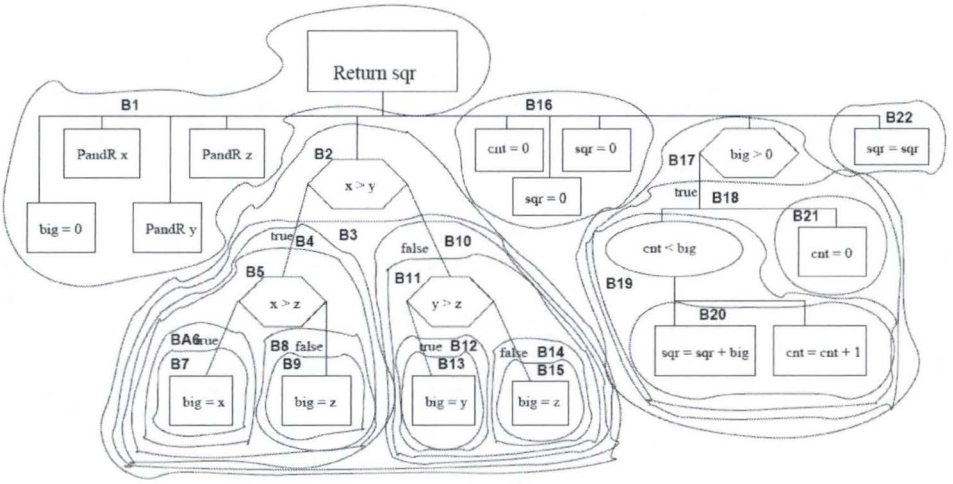

FIGURE 3.6 – Arbre de régions du programme copié [1]

4. Comparaison des arbres d'évaluation

A ce stade, l'algorithme de *BRASS* transforme chaque noeud, un noeud étant constitué d'une seule instruction, en ce que les auteurs du système nomment un arbre d'évaluation. Aucune précision n'est donnée sur ces arbres d'évaluation mais on peut supposer qu'il s'agit d'arbres syntaxiques classiques où la forme d'un noeud est représentée différemment selon le type d'opérateur contenu à l'intérieur de ce noeud. La structure de ces sous-arbres est alors comparée de deux manières. D'une part un algorithme d'isomorphisme d'arbres analyse la forme des sous-arbres d'évaluation pour y détecter d'éventuelles similarités. D'autre part les opérandes contenues dans les feuilles de ces sous-arbres sont examinées.

5. Comparaison des dictionnaires

Finalement, les dictionnaires générés lors de la représentation graphique

du code source sont analysés. D'abord le nombre de type d'éléments est calculé. Pour les dictionnaires représentés à la figure 3.7, trois *ints* et deux *floats* sont comptés dans le premier dictionnaire contre deux *ints* et quatre *floats* dans le second. Ensuite, pour chaque entrée des dictionnaires à comparer, des paires *type-id* sont générées et confrontées. Deux pourcentages sont ainsi calculés : un représentant les types identiques et un autre indiquant les paires *type-id* similaires.

Dictionnaire 1	Dictionnaire 2
int x	int x
int y	int y
int z	int z
int sqr	int sqr
int cnt	int cnt
int big	int big

FIGURE 3.7 – Dictionnaires des diagrammes de structure [1]

Performance et complexité

Pour évaluer leur moteur, les concepteurs de *BRASS* comparent les résultats fournis par celui-ci avec les résultats produits par le moteur *MOSS* sur le même code source. Les programmes fournis en entrée aux deux systèmes ont subi les modifications identifiées par Whale[12] et pouvant être appliquées par un plagiaire dans le but de transformer du code source. La tableau 3.3 montre la comparaison des résultats produits par les deux systèmes.

Modification	*MOSS*	*BRASS*
Commentaires	X	X
Formatage	X	X
Identifiants	V	V-CAE
Ordre des opérandes	V	V-CD
Type des données	V	V-CAE
Expressions	V	V
Instructions redondantes	V	V-CAE
Ordre des instructions	V	V-CAE
Instructions d'itération	V	V-CAE
Instruction de sélection	V	V
Appel de procédure	V	V-CAE
Combinaison	V	V

TABLE 3.3 – Comparaison des résultats fournis par *MOSS* et *BRASS [1]*

Le symbole X signifie que le système ne notifie pas le changement au niveau du programme copié, tandis que le symbole V signifie la détection d'une modification. *V-CAE* dénote le fait que *BRASS* détecte un changement dès la phase de comparaison des arbres d'évaluation tandis que pour *V-CD*, le changement est signalé à la phase de comparaison des dictionnaires. Suivant ces résultats, les auteurs de *BRASS* affirment que leur système fournit une information plus détaillée à l'utilisateur : celui-ci pourrait d'abord examiner le pourcentage des arbres de régions correspondantes, pour ensuite analyser plus particulièrement les résultats donnés par la comparaison des arbres d'évaluation ou des dictionnaires. Toutefois, il est raisonnable de supposer que *BRASS* ou *SIM* ne sont pas plus rapides que les moteurs à comparaison d'empreinte (*MOSS*) ou que les moteurs à comparaison de chaînes de caractères (*JPlag*). Aucune information n'est apportée par les concepteurs de *BRASS* sur la complexité de celui-ci, par-contre la complexité de l'algorithme utilisé par *SIM* est estimée à $O(s^2)$ où s représente la taille maximale des arbres syntaxiques. Puisque *SIM* utilise également un algorithme d'alignement de chaînes de caractères et que la taille d'un arbre de fichier de code source est proportionnelle la taille de ce fichier (n), la complexité totale du procédé de détection de plagiat est évaluée à $O(s^2n^2)$[15].

3.7 Techniques basées sur la compression

L'approche basée sur la compression est radicalement différente des techniques vues précédemment. Plutôt que de se concentrer sur une application spécifique comme la détection de plagiat, cette technique étudie une mesure basée sur l'information partagée par deux séquences. Cette approche se détache des autres méthodes par le fait que ces séquences peuvent être de n'importe quel type : séquence ADN, séquence de texte, séquence de fichiers musicaux, séquence linguistique et séquence de programmes. Cette mesure, appelée distance informationnelle car elle dépend du contenu de l'information, fût inventée au cours des années nonante par Charles Bennet[37]. Elle fût ensuite utilisée en 1998 par l'équipe de bio-informatique du laboratoire d'informatique fondamentale de Lille pour classer des séquences génétiques[38] : la distance informationnelle entre deux chaînes de caractères x et y est définie par la taille du plus court programme permettant de transformer x en y et y en x. Peu après, cette méthode, reprise par des chercheurs de l'Université d'Amsterdam [39, 40, 41], fut perfectionnée et simplifiée sous le nom de *distance de similarité.*

Récemment, cette distance a été adaptée à toutes sortes de problèmes :

- En linguistique, une expérience a permis d'élaborer automatiquement un arbre de classification des 52 langues indo-européennes principales sur base des similarités existant entre ces langues. L'arbre obtenu est conforme à celui obtenu par des experts linguistes.
- Toujours dans le domaine de la linguistique, une autre étude a permis de classer des écrivains russes sur base d'extraits de leurs ouvrages.
- Dans le domaine musical, une expérience a permis de classer automatiquement des morceaux de musiques codés dans le format MIDI sur base de leurs caractéristiques musicales.
- En génétique, un arbre phylogénétique des mammifères placentaires à partir des génomes complets des différentes espèces a été produit automatiquement. L'arbre obtenu par la méthode de distance de similarité

est conforme à l'arbre admis par les spécialistes.

- Dans la détection de plagiat de texte, plusieurs outils implémentent la technique basée sur la distance de similarité, notamment les logiciels *FindFraud*[42] et *Baldr*[43].

La distance de similarité trouve ses origines dans la complexité de Kolmogorov inventée dans les années soixante[44], l'objectif étant de trouver une notion d'information contenue dans une chaîne binaire en utilisant la longueur d'un programme informatique générant cette chaîne. La complexité de Kolmogorov désigne donc la quantité de complexité d'un objet composé de données et s'exprime par le nombre de bits du programme qui peut générer un tel objet.

Par conséquent, la complexité de Kolmogorov d'une chaîne de caractères x ,et notée $\mathrm{K}(x)$, exprime donc le montant d'information absolue contenue dans la séquence x. $\mathrm{K}(x)$ est donc la longueur, en bits, du plus court programme qui imprime x.

Soit x = *'01010101010101'* et y = *'01101000010100'*. La complexité de Kolmogorov pour ces deux séquences peut s'écrire $\mathrm{K}(x) < \mathrm{K}(y)$ puisque le programme nécessaire pour générer la séquence x est plus simple (il suffit de répéter 7 fois la sous-séquence *'01'*) que celui nécessaire pour générer la séquence y (dans laquelle les sous-séquences *'0'* et *'1'* sont positionnées aléatoirement).

Théoriquement la complexité de Kolmogorov est universelle. Cette universalité garantit que, dans sa version idéale, n'importe quelle différence entre deux séquences sera détectée. Néanmoins, la complexité de Kolmogorov est incalculable[44] mais il est possible de s'en rapprocher en utilisant un algorithme de compression de données. L'objectif d'un algorithme de compression est d'éliminer les redondances d'informations dans les données, c'est-à-dire les répétitions surabondantes de séquences de symboles. Plus les répétitions sont importantes et plus la compression sera importante. Les algorithmes utilisés doivent être sans perte, c'est à dire qu'aucune perte de données sur l'information d'origine n'est permise, l'information étant seulement réécrite d'une manière plus concise. Pour s'approcher au maximum de la complexité

de Kolmogorov, les compresseurs, i.e. les algorithmes de compression, doivent garantir les propriétés suivantes :

- Idempotence : $C(C(x)) = C(x)$ où $C(x)$ dénote la compression sur la séquence x, et $C(\$) = 0$ où \$ dénote la séquence vide.
- Monotonie : $C(xy) \geq C(x)$.
- Symétrie : $C(xy) = C(yx)$.
- Distributivité : $C(xy) + C(z) \leq C(xz) + C(yz)$.

En appliquant un algorithme de compression à deux séquences x et y, l'information partagée par x et y peut être calculée par[45] :

$$C(x) + C(y) - C(xy)$$

où $C(xy)$ exprime la version comprimée des séquences x et y concaténées

Lors du calcul de $C(xy)$, les informations communes à x et y ne sont comptées qu'une seule fois. Après avoir comprimé x, le compresseur comprime y et élimine l'information redondante qui était déjà dans x. Les informations propres à x et celles propres à y ne sont donc comptées qu'une fois dans $C(xy)$. En revanche, lors du calcul de $C(x)$ et de $C(y)$, les informations propres à x et les informations propres à y sont comptées une fois, tandis que les informations communes à x et y sont comptées deux fois (une fois lors du calcul de $C(x)$ et une fois lors du calcul de $C(y)$).

Le calcul de la différence $C(x) + C(y) - C(xy)$ se simplifie et il n'en reste finalement qu'un seul terme : le contenu commun en information de x et y, c'est à dire l'économie d'espace obtenu en comprimant xy en une fois comparée à des compressions séparées de x et de y. $C(x) + C(y) - C(xy)$ est donc une mesure du contenu commun en information de x et y.

On définit alors la distance de similarité entre les séquences x et y par :

$$d(x,y) = 1 - \frac{C(x) + C(y) - C(xy)}{C(x)} \; si \; C(y) \leq C(x)$$

$$d(x,y) = 1 - \frac{C(x) + C(y) - C(xy)}{C(y)} \; si \; C(x) \leq C(y)$$

Les dénominateurs apparaissant dans les formules sont des facteurs de normalisation, qui ne jouent un rôle important que quand on manipule des données x et y de tailles très différentes. La quantité d(x, y) est comprise entre 0 et 1 et possède les propriétés classiques d'une distance, à savoir :

- d(x, y) = 0 si et seulement si $x = y$.
- d(x, y) = d(y, x).
- d(x, y) $\leq$ d(x, z) + d(z, y).

L'interprétation de la quantité d'information soit d(x, y) s'obtient comme suit :

- Si x et y sont sans rapport (par exemple deux séquences complètement aléatoires ou deux textes sans liens dans des langues différentes), alors le contenu commun en information de x et y est nul ; C(xy) = C(x) + C(y) et donc d(x, y) = 1, la valeur maximale de la distance.
- Si $x = y$, alors C(x) = C(xy) = C(y) et donc d(x, y) = 0.

Ainsi, plus le contenu commun en information de x et y est grand, plus petite est la distance d(x, y). A l'inverse, plus les séquences x et y sont indépendantes (sans corrélation), plus d(x, y) s'approche de 1.

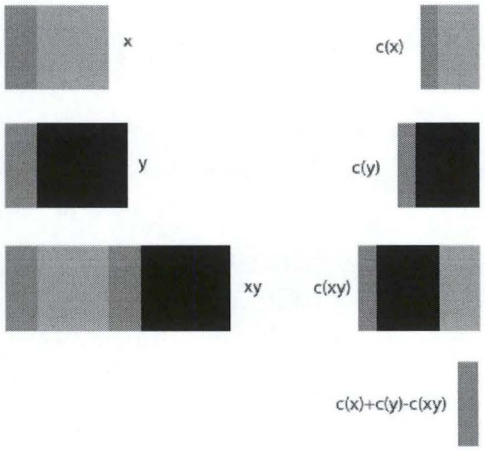

FIGURE 3.8 – Contenu commun en information à des données

La figure 3.8 montre l'évaluation du contenu commun en information à deux ensembles de données[45]. Ces données peuvent être de n'importe quel type : deux textes, deux morceaux de musique ou deux séquences d'ADN, deux fichiers de code source) . L'ensemble des données x comporte des informations qui lui sont spécifiques (en gris clair) et des informations partagées avec l'ensemble de données y (en gris foncé). Les informations de y se décomposent en une partie spécifique à y (en noir) et la partie commune (en gris foncé). Les versions comprimées de x et de y comportent similairement des parties spécifiques et des parties communes. Lorsque la concaténation de x et y est comprimée, la partie commune à x et y n'est pas dupliquée, car la compression supprime les redondances, et donc la version comprimée de xy comporte une partie correspondant à la version comprimée de la partie colorée en gris clair (spécifique à x), une autre pour la version comprimée de la partie colorée en noir (spécifique à y) et une dernière pour la version comprimée de la partie colorée en gris foncé. Les longueurs des données comprimées C(x), C(y) et C(xy) mesurent les contenus en information de x, de y et de xy. Le schéma illustre que l'expression C(x) + C(y) - C(xy) se simplifie et que le résultat obtenu est une mesure du contenu commun en

information à deux ensembles de données. La distance de similarité vaudra *0* (approximativement) lorsque x et y auront le même contenu en information et prendra une valeur proche de 1 lorsque x et y seront peu corrélés.

Puisque les séquences à comparer peuvent être de tout type, l'approche basée sur la compression peut également être appliquée à la détection de code source. Saxon[25] est le premier à implémenter cette technique pour détecter des similarités au sein de programme. L'algorithme de compression utilisé dans son système est construit en utilisant les fonctions *gzip* standards des librairies Java. Saxon affirme que les résultats obtenus par son système sont prometteurs et que l'approche basée sur la compression se révèle plus efficace que les méthodes de détection standards. Les expériences menées par Campbell et Culwin sont moins encourageantes. Ces derniers comparent les résultats obtenus par le système de Saxon, en le réimplémentant, avec ceux obtenus par *JPlag* et *MOSS* et concluent que l'étude menée n'a pas été en mesure de valider les conclusions de Saxon. Campbell et Culwin soulèvent néanmoins l'hypothèse que le système réimplémenté n'est pas une copie parfaite de celui de Saxon, ce qui pourrait être la cause de leur échec.

L'outil le plus prometteur utilisant une technique basée sur la compression est le système *SID (Software Integrity Diagnosis system ou encore Shared Information Distance)*. Développé et utilisé à l'université de Santa Barbara, *SID* est disponible librement[46].

SID s'exécute en trois phases[47] :

- Dans la première phase, les fichiers de code source sont traités par un analyseur lexical standard pour générer des lexèmes codés sur un byte.
- Durant la deuxième phase, un algorithme de compression est utilisé pour approcher au maximum la complexité de Kolmogorov. L'algorithme, appelé *TokenCompress*, n'est pas expliqué en détail dans la littérature mais les auteurs de *SID* affirment qu'il s'agit d'une version modifiée de l'algorithme de compression de Lempel-Ziv[48]. L'algorithme *TokenCompress* calcule une distance de similarité pour chaque

paires de fichiers de code source et enregistre les portions de code similaires dans un fichier.
- Finalement, les paires de programmes comparés sont classées selon leurs distances de similarité et affichées dans une matrice. L'utilisateur a également la possibilité de visionner les portions de code similaires via l'interface graphique de *SID*.

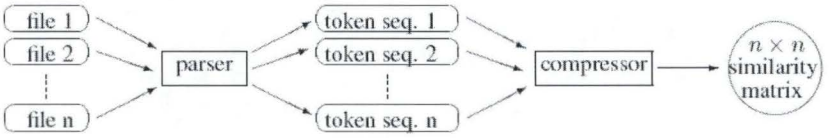

FIGURE 3.9 – Architecture de *SID*

Les auteurs de *SID* affirment que leur système a été testé sur plusieurs centaines de fichiers de code source provenant de travaux d'étudiants et donne d'excellents résultats. Ces derniers déclarent en outre que[47] :

- *SID* est insensible au renommage de variables et/ou de fonctions, au réordonnancement de fonctions/méthodes/variables, et à l'insertion et/ou la suppression de code inutile.
- Dans tous les tests, *SID* n'a donné aucun faux positif. A l'inverse, tous les programmes jugés comme similaires par *SID* ont été déclarés plagiés par un examinateur humain.
- Les résultats des tests montrent clairement que *SID* est plus fiable que les autres outils de détection de plagiats auxquels il a été comparés, en l'occurrence *MOSS* et *JPlag*. Ces tests sont disponibles sur le site internet de *SID*.

Complexité

Peu de précisions concernant le temps d'exécution du système *SID* sont apportées par la littérature. Les auteurs de *SID* déclarent que leur pro-

gramme peut s'exécuter en temps linéaire mais admettent néanmoins que la priorité donnée à leur algorithme est la meilleure compression possible plutôt que le temps d'exécution. Comme la complexité de l'algorithme de Lempel-Ziv se situe entre $O(n)$ et $O(n^2)$ suivant les implémentations choisies, il est à supposer que le temps d'exécution de leur système tend plus vers $O(n^2)$ que vers un temps d'exécution linéaire.

3.8 Méthodes d'évaluation des techniques de détection de plagiat

L'évaluation des différentes techniques de détection de plagiat reste un problème crucial. En général, les comparaisons des systèmes recensés dans la littérature sont effectuées par leurs concepteurs et sont habituellement des méthodes d'évaluation simplistes souffrant d'un manque d'objectivité, le but premier étant de mettre en avant les effets positifs des nouvelles techniques proposées.

Dès 1990, Whale [12] a tenté de mesurer la fiabilité des méthodes de détection de plagiat en adaptant des métriques habituellement utilisées dans le calcul de l'efficacité de recherche d'informations. La première mesure, appelée la précision, représente la proportion de détections positives retournées par le système parmi l'ensemble des programmes plagiés. Une haute précision permet de diminuer le nombre de faux-positifs. La deuxième mesure, le rappel, représente la proportion de détections positives retournées par le système parmi l'ensemble des programmes analysés. Un rappel important permet de s'assurer que des programmes plagiés ne seront pas marqués comme non-plagiés. Néanmoins les métriques utilisées par Whale sont difficilement calculables car le nombre de programmes plagiés contenus dans l'ensemble des programmes à traités doit être connu à l'avance, ce qui implique une intervention par un tiers humain. Cette investigation manuelle peut poser certains problèmes [49] :

– Des opinions divergentes peuvent être émises à propos d'un même pro-

gramme.

- L'analyse manuelle de collections contenant des dizaines ou centaines de programmes est irréalisable.

Plus récemment, Mozgovoy [50] compare plusieurs moteurs de détection de plagiat en appliquant un test dit de conformité. Ce test examine les résultats fournis par ces différents moteurs en utilisant les mêmes données en entrée, constituées de 220 programmes écrits en Java. Chaque moteur est alors confronté à un jury formé par les autres systèmes. Les résultats de cette étude sont montrés à figure 3.10. Néanmoins, le classement fourni par les recherches de Mozgovoy est uniquement basé sur des déductions et ne peut donc être considéré comme une preuve formelle de la supériorité de certains moteurs de détection par rapport à d'autres.

A l'heure actuelle, il n'est donc toujours pas possible d'affirmer qu'une technique est supérieure aux autres, aucune procédure simple et fiable n'étant en mesure d'évaluer la fiabilité des moteurs de détection de plagiat de code source.

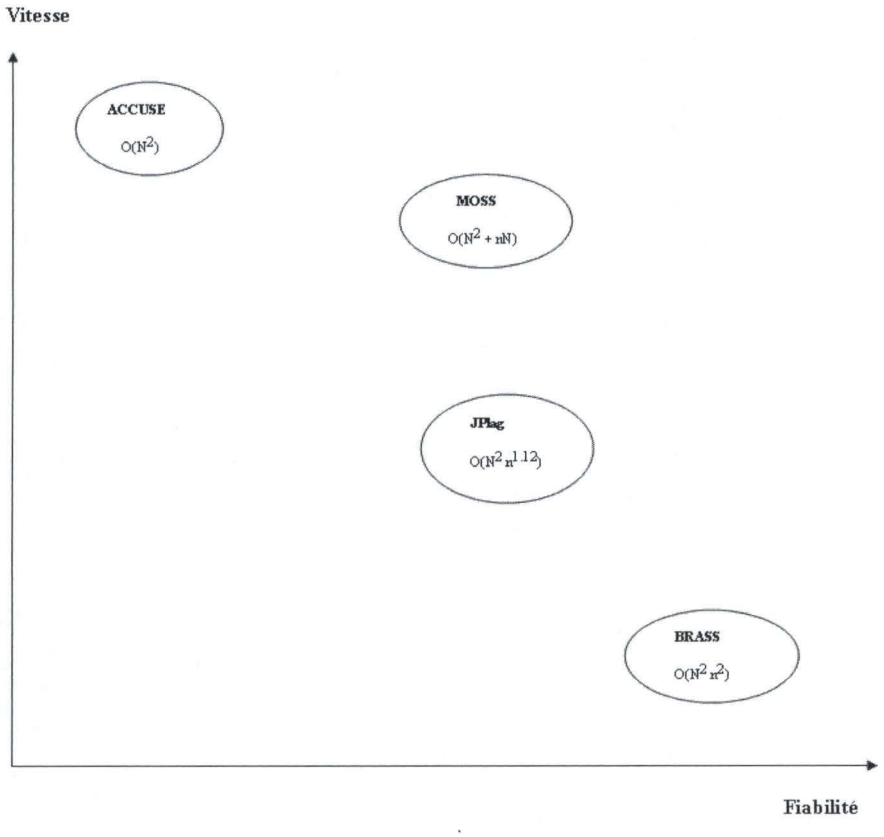

FIGURE 3.10 – Évaluation des différents systèmes par l'étude de Mozgovoy

3.9 Conclusion

Depuis une trentaine d'années, de nombreuses techniques de détection de plagiat ont été développées et les algorithmes utilisant ces techniques ont subi une amélioration constante. Ce chapitre a présenté tout d'abord les systèmes de détection selon les classifications habituellement recensées dans la littérature : les systèmes à comptage d'attributs et les systèmes à métriques de structure. Il a été démontré que ces classifications sont inappropriées pour classer les différentes techniques, néanmoins cette partie de chapitre a permis de comprendre l'évolution des premiers systèmes, les systèmes basés sur la

comparaison d'empreintes quantitatives (systèmes à comptage d'attributs), non fiables et dès à présent devenus obsolètes, vers des systèmes plus modernes mais également plus complexes. La figure 3.11 montre l'évolution de ces techniques d'un point de vue chronologique.

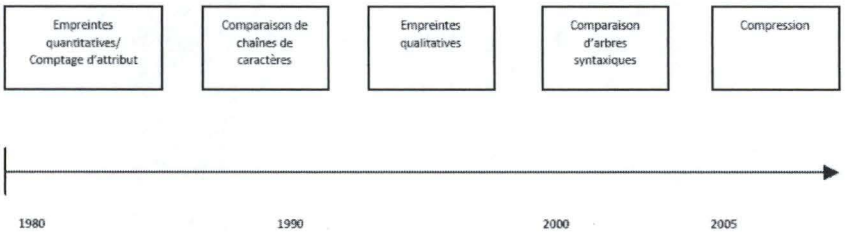

FIGURE 3.11 – Chronologie des différentes méthodes de détection de plagiat de code source

Les différentes techniques de détection de plagiat ont ensuite été discutées en détail selon la classification évoquée par Mozgovoy. Une nouvelle catégorie de technique y a été ajoutée : les techniques basées sur la compression. De plus, la catégorie des techniques basées sur la génération d'empreintes a été sous-classée par l'apparition des techniques à comparaison d'empreintes qualitatives (traditionnellement appelées techniques à comptage d'attributs) et des techniques à comparaison d'empreintes quantitatives. Les techniques suivantes ont donc été identifiées :

- Les techniques à comparaison d'empreintes
  - empreintes qualitatives
  - empreintes quantitatives
- Les techniques à comparaison de chaînes de caractères
- Les techniques à comparaison d'arbres syntaxiques
- Les techniques basées sur la compression de fichiers

La figure 3.12 schématise la classification de cette différentes techniques, les algorithmes utilisés ainsi que les moteurs de détection de plagiat qui implémentent ces algorithmes. Les différents algorithmes implémentant ces techniques ont ensuite été expliqués en détail et une complexité moyenne des systèmes a été calculée.

Finalement, les méthodes permettant d'évaluer le degré de fiabilité des ces algorithmes ont été exposées. Deux procédés d'évaluation des systèmes de détection de plagiat ont été examinés : la précision et le rappel ainsi que le test de conformité. Actuellement, ces procédés ne permettent pas de déterminer si une technique de détection de plagiat est supérieure aux autres.

Les algorithmes détaillés dans ce chapitre sont implémentés par des moteurs de détection de plagiat de code source. Ces derniers sont présentés dans le chapitre suivant.

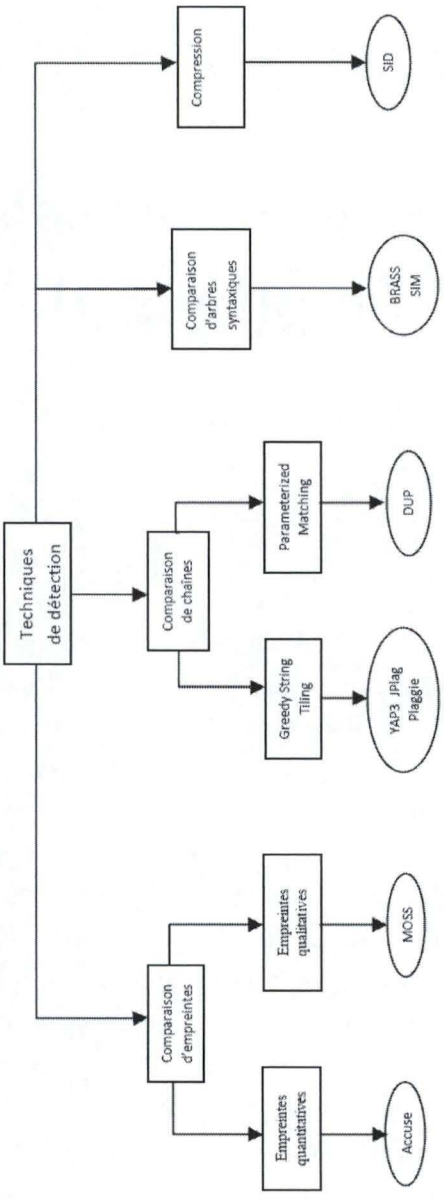

FIGURE 3.12 – Techniques de détection de plagiat de code source

Chapitre 4

Revue des outils actuels de détection de plagiat de code source

4.1 Introduction

Le processus de détection de plagiat de code source implique la présentation détaillée à l'utilisateur des résultats obtenus. Plus spécifiquement, les moteurs de détection de plagiat devraient être capables[15] :

- D'afficher la liste des paires de programmes considérés comme copiés ainsi leur degré de similarité.
- D'afficher le code identique pour une paire de programmes sélectionnée.

Ce chapitre présente une revue des moteurs de détection de plagiat de code source modernes recensés dans la littérature concernée : chaque système y est exposé d'un point de vue utilisateur et son interface graphique y est détaillée de manière à ce que le lecteur puisse opérer un choix potentiel parmi le panel de moteurs proposés. Toutefois, certains moteurs précédemment cités dans ce travail ne sont pas exposés dans ce chapitre :

- Le système *BRASS[1]*, développé à l'université de Tulane et utilisant une technique de comparaison d'arbres, est un système privé, à savoir qu'il n'est pas disponible librement.

- *SID[47, 46],* développé à l'université de Santa Barbara et implémentant une technique de comparaison basée sur la compression, est indisponible à l'heure où sont écrites ces lignes.
- *SIM[35],* implémenté à l'université d'Amsterdam et utilisant une technique de comparaison d'arbres, n'offre aucune interface graphique, il n'est donc pas envisageable de détailler ce système.

A la fin du chapitre, plusieurs tableaux comparatifs (tableaux 4.2 et 4.4) confrontent les systèmes de détection de plagiat suivant une dizaine de caractéristiques et suivant la présentation des résultats (tableau 4.6).

4.2 Moteurs de détection modernes

4.2.1 *JPlag*

Développé en Java par Guido Malpohl à l'université de Karlsruhe en 1996, *JPlag*[51] est certainement l'un des moteurs de détection de plagiat de code source les plus populaires parmi la communauté académique. Les langages supportés par *JPlag* sont Java, C#, C, C++ et Scheme. L'algorithme de comparaison de code source utilisé par *JPlag* est l'algorithme *Running-Karp-Rabin-Greedy-String-Tiling*, élaboré à la base par Michael Wise pour l'alignement de séquences ADN[32]. Cet algorithme est appelé par le système après une phase de lexèmisation appliquée sur le code source à examiner.

L'utilisation du système est gratuite mais nécessite la création d'un compte. *JPlag* est disponible à travers un service web, la partie client de l'application, implémentée avec la technologie Java Web Start[52], étant téléchargée et exécutée sans installation sur l'ordinateur de l'utilisateur. L'utilisation du programme nécessite cependant une machine java virtuelle (Java 1.5 ou plus).

Une fois l'application démarrée, l'utilisateur fournit au système l'emplacement du répertoire contenant les soumissions devant être comparées (figure 4.1). Ce répertoire peut contenir des sous-répertoires représentant les travaux de chaque étudiant à analyser. Les soumissions sont ensuite téléchargées vers le serveur *JPlag* pour y être examinées (figure 4.2). Les résultats sont alors

stockés sur la machine de l'utilisateur sous forme d'un ensemble de pages HTML (figure 4.3) et présentés sous la forme d'un histogramme affichant les scores de similarité obtenus, ce qui permet de distinguer aisément les cas de plagiat des autres. Si un doute subsiste, l'utilisateur peut examiner plus particulièrement les soumissions suspectes (figure 4.4) en effectuant une comparaison côte à côte de la paire de programmes problématiques : les lignes de code source correspondantes seront surlignées en rouge.

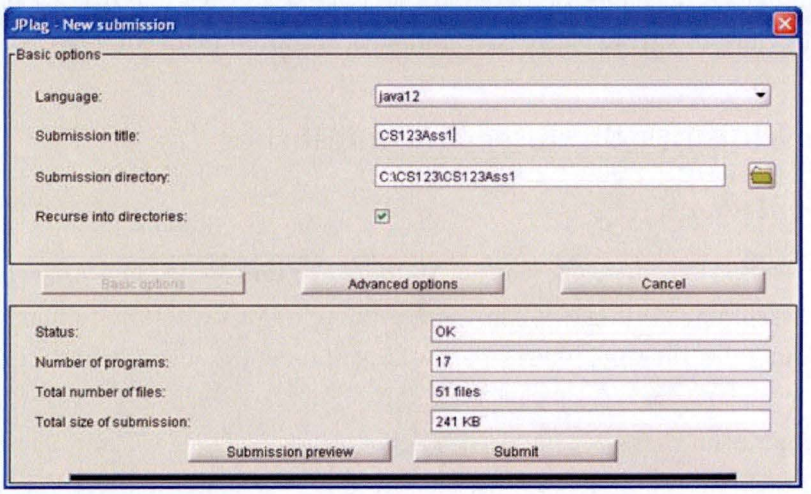

FIGURE 4.1 – Sélection des soumissions dans *JPlag*

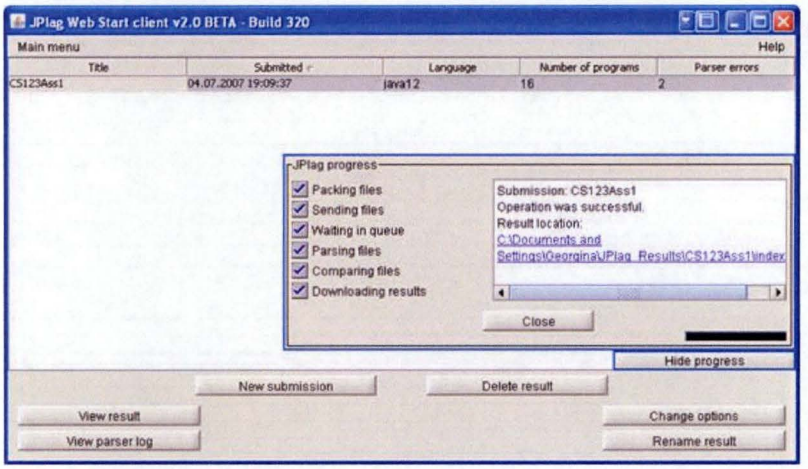

FIGURE 4.2 – Processus de détection dans *JPlag*

Search Results	
Title:	CS123
Directory:	C:\CS123\CS123Ass1
Programs:	011 - 062 - 126 - 128 - 165 - 273 - 318 - 373 - 426 - 481 - 802 - 812 - 886 - 912 - 955 - 995
Language:	Java1.2 Parser
Submissions:	16 **(1 has not been parsed successfully)**
Invalid submissions (see log file) :	527
Matches displayed:	20 (Treshold: 50.0%) (average similarity) 20 (Treshold: 52.0%) (maximum similarity)
Date:	2007-07-04
Minimum Match Length (sensitivity):	7
Suffixes:	.java, .jav, .JAVA, .JAV

Distribution:

90% - 100%	1	##
80% - 90%	0	.
70% - 80%	5	##########
60% - 70%	6	############
50% - 60%	8	################
40% - 50%	14	############################
30% - 40%	18	###################################
20% - 30%	34	##
10% - 20%	22	##
0% - 10%	12	#########################

Matches sorted by average similarity (What is this?):

812	->	165 (100.0%)	912 (79.4%)	426 (74.5%)	273 (67.2%)	011 (58.4%)	126 (51.4%)
165	->	912 (79.4%)	426 (74.5%)	273 (67.2%)	011 (58.4%)	126 (51.4%)	
011	->	955 (76.4%)	273 (64.2%)	426 (59.8%)	912 (56.7%)		
426	->	912 (69.7%)	273 (69.1%)				
273	->	912 (63.1%)	955 (50.7%)				
128	->	126 (50.0%)					

FIGURE 4.3 – Résultats affichés par *JPlag*

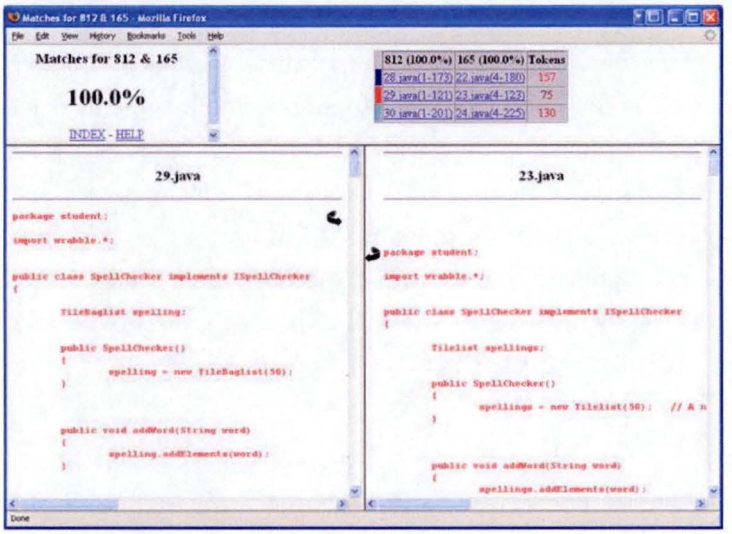

FIGURE 4.4 – Comparaison côte à côte d'une paire de programmes dans *JPlag*

La popularité de *JPlag* peut sans aucun doute s'expliquer par sa facilité d'utilisation, par la précision de son interface graphique et par la robustesse de son algorithme de comparaison dont l'efficacité n'est plus à démontrer. De plus, la documentation fournie est claire et explique non seulement le fonctionnement interne du système mais aussi la méthode utilisée pour évaluer ce dernier par ses concepteurs. Enfin, *JPlag* permet à l'utilisateur d'indiquer que du code source ne doit pas faire l'objet d'une comparaison, option utile lorsque le professeur fournit aux étudiant une base commune de code source.

Néanmoins, deux points négatifs sont à relever. Premièrement, *JPlag* ne peut pas examiner les programmes générant des erreurs de compilation alors que d'autres moteurs de détection de plagiat peuvent gérer de tels programmes. Deuxièmement, les travaux des étudiants doivent être téléchargés sur le serveur de *JPlag* pour y être examinés, ce qui pourrait poser des problèmes de confidentialité si des informations sensibles, par exemple le nom des étudiants ou de l'université se trouvant en commentaire dans le code source, venaient à être envoyées par erreur vers le serveur de *JPlag*.

Un autre moteur de détection de plagiat de code source, similaire en

de nombreux points à *JPlag*, a été implémenté à l'université de Helsinki en 2006. Ce moteur, appelé *Plaggie*[33], utilise également l'algorithme de comparaison de Wise et son interface graphique est pratiquement identique à celle de *JPlag*. De plus, le code source de *Plaggie* est librement distribuable. En revanche, *Plaggie* n'est pas un service web, l'examen des soumissions s'effectuant directement sur l'ordinateur de l'utilisateur, ce qui peut résoudre le problème de confidentialité de *JPlag* précédemment soulevé. Néanmoins, vu l'absence de littérature à ce sujet, il apparaît impossible d'affirmer que *Plaggie* est aussi fiable que *JPlag*.

4.2.2 *MOSS*

Même si *MOSS* [28] *(Measure Of Software Similarity)* apparaît comme l'un des meilleurs moteurs de détection de plagiat, la littérature à son sujet reste discrète. Cette réserve peut s'expliquer par la volonté de son concepteur, Alex Aiken de l'université de Berkeley, à ne pas dévoiler le fonctionnement interne de son système afin que des plagiaires ne puissent pas modifier le code soumis dans le but de tromper le système.

Développé en 1994, *MOSS* utilise une technique de comparaison d'empreintes qualitatives (algorithme *Winnowing*) pour comparer des paires de programmes. La comparaison s'effectue après transformation du code source en lexèmes.

Par le passé *MOSS* était uniquement reservé aux professeurs d'informatique ; ce n'est plus le cas à présent et n'importe qui peut obtenir gratuitement un compte pour utiliser le système. *MOSS* supporte également une grande variété de langages de programmation : C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, ...

```
C:\Users\phil\Downloads>perl moss -l java Stack.java Stackbis.java
Checking files . . .
OK
Uploading Stack.java ...done.
Uploading Stackbis.java ...done.
Query submitted.  Waiting for the server's response.
http://moss.stanford.edu/results/538560809
```

FIGURE 4.5 – Soumission des programmes vers le serveur *MOSS*

Le système est implémenté sous la forme d'un service web et l'utilisation de *MOSS* s'effectue en ligne de commande (figure 4.5) via un script de soumission nécessitant l'installation de Perl et devant être préalablement téléchargé sur le site internet de *MOSS*[30]. Les soumissions sont alors envoyées vers le serveur de *MOSS* pour y être comparées. Les résultats de l'analyse sont disponibles pour une durée de deux semaines sur le site web de *MOSS*. A l'instar de *JPlag*, ces résultats sont affichés via un ensemble de pages HTML sous la forme d'une liste de paires de programmes classés selon leur degré de similarité. L'utilisateur peut alors sélectionner une paire particulière pour procéder à un examen plus détaillé : les deux programmes suspects sont présentés côte à côte et les portions de code identiques sont surlignées dans la même couleur (figure 4.6). Comme dans *JPlag*, un professeur peut aussi fournir une base commune de code source qui sera exclue de la comparaison.

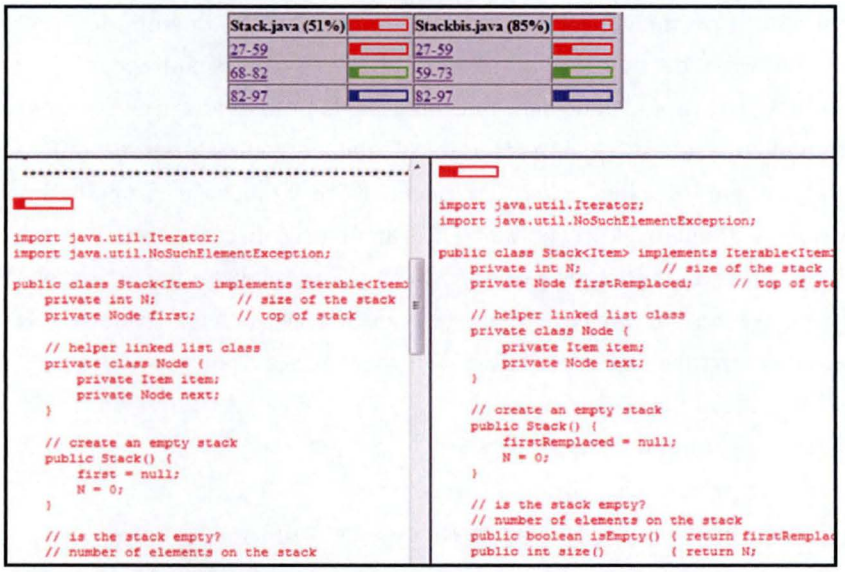

FIGURE 4.6 – Comparaison côte à côte d'une paire de programmes dans *MOSS*

En 2001, Culwin, MacLeod et Lancaster[24] publient une étude dans

laquelle ils comparent *JPlag* et *MOSS* en fournissant à chacun des deux moteurs de détection la même collection de soumissions. Ceux-ci sont incapables de déterminer si l'un des deux systèmes est meilleur. Le principal avantage de *MOSS* est qu'il est en mesure d'examiner des programmes générant des erreurs de compilation, alors que *JPlag* ne l'est pas. Par contre, *JPlag* fournit une interface graphique de qualité permettant de soumettre aisément les programmes à analyser au serveur tandis qu'avec *MOSS* la soumission se fait par ligne de commande.

Finalement, comme *JPlag*, *MOSS* présente également un problème de protection des données puisque les programmes soumis sont exportés vers un serveur se trouvant hors de l'institution concernée.

4.2.3 *Sherlock*

Développé au département informatique de l'université de Warwick, le système de soumission en ligne *BOSS*[53] est un outil de gestion de travaux d'étudiants. Il permet à ces derniers de soumettre leurs devoirs de programmation en ligne de manière sécurisée pour être par la suite côtés par les professeurs. *BOSS* contient de nombreux outils pour réaliser ces tâches dont notamment un moteur de détection de plagiat de code source, nommé *Sherlock*. Implémenté en Java, *Sherlock* nécessite une machine virtuelle Java et s'exécute localement. Le système est gratuit et son code source est libre. *Sherlock* utilise l'algorithme *Greedy-String-Tiling* élaboré par Michael Wise pour détecter des similarités au sein de code source mais à l'inverse de *JPlag*, le système compare 5 fois les paires de programmes à analyser :

- Dans leurs formes originales.
- En supprimant les espaces.
- En supprimant les commentaires.
- En supprimant les espaces et les commentaires.
- En appliquant une phase de lexèmisation sur les fichiers à comparer.

D'après ses concepteurs, les résultats obtenus par le système *Sherlock* sont encourageants : sur une période de trois années d'utilisation, le pourcentage des cas de plagiat de code source recensés dans le département informatique

de l'université de Warwick est passé de 6 % à environ 1%. *Sherlock* a été également comparé avec le système *Plague* de Whale[12], un ancien moteur de détection de plagiat de code source : les résultats de cette confrontation sont également positifs puisque *Sherlock* détecte tous les cas de plagiat présents dans une suite de 154 programmes. Il est à noter que la comparaison incrémentale utilisée par *Sherlock* risque fort de ralentir son temps d'exécution par rapport à des outils comme *JPlag* ou *MOSS* sur des collections importantes de programmes à analyser.

Sherlock se distingue cependant en proposant une interface graphique impressionante. L'utilisateur sélectionne tout d'abord le répertoire contenant les programmes à analyser (figure 4.7) et peut par la suite modifier les paramètres de la comparaison (figure 4.8).

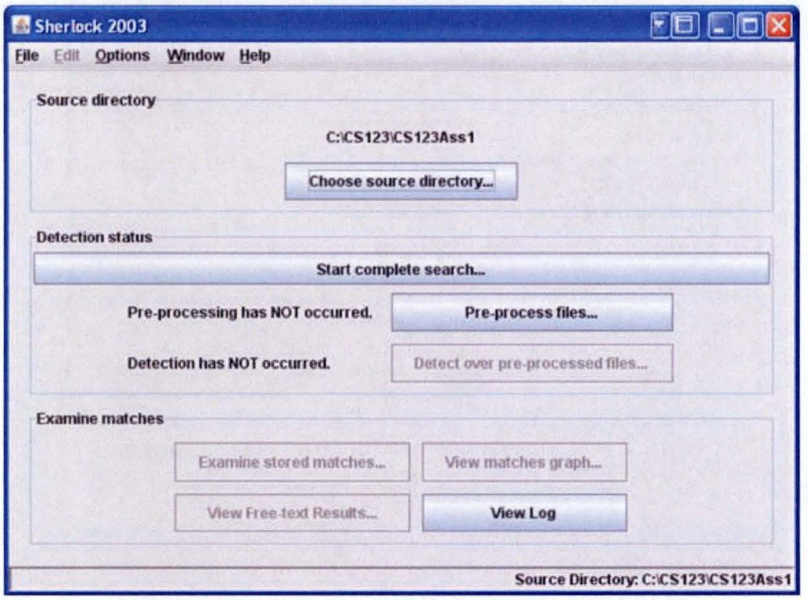

FIGURE 4.7 – *Sherlock* : écran initial

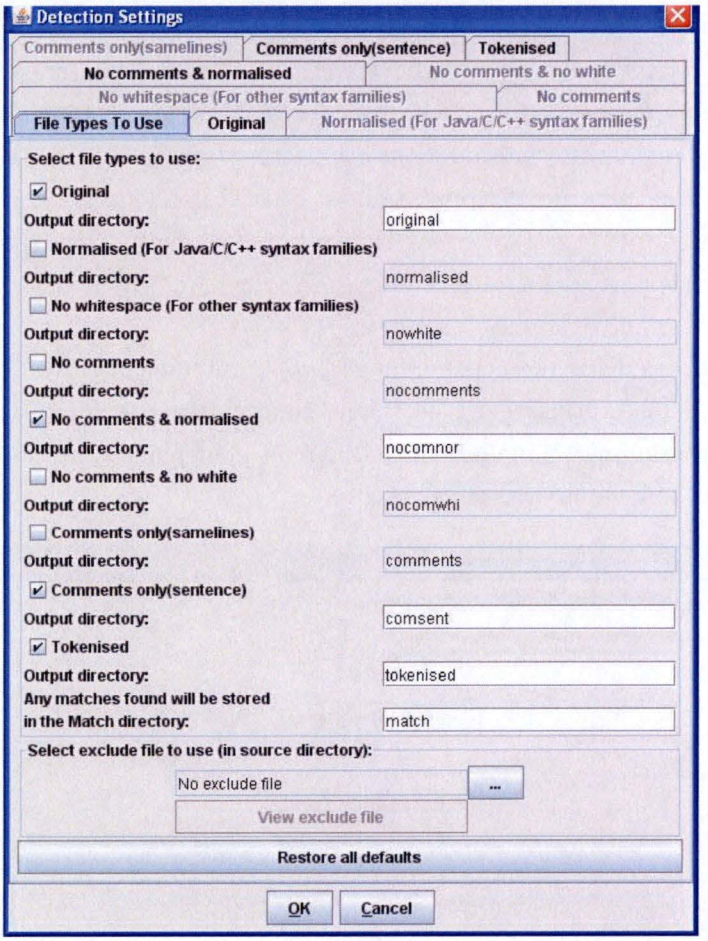

FIGURE 4.8 – *Sherlock* : modification des paramètres

Une fois la comparaison effectuée, le système affiche les paires de programmes analysés sous la forme d'un arbre. Par exemple, à la figure 4.9, trois portions de code suspect ont été détectées pour les programmes 39 et 27.

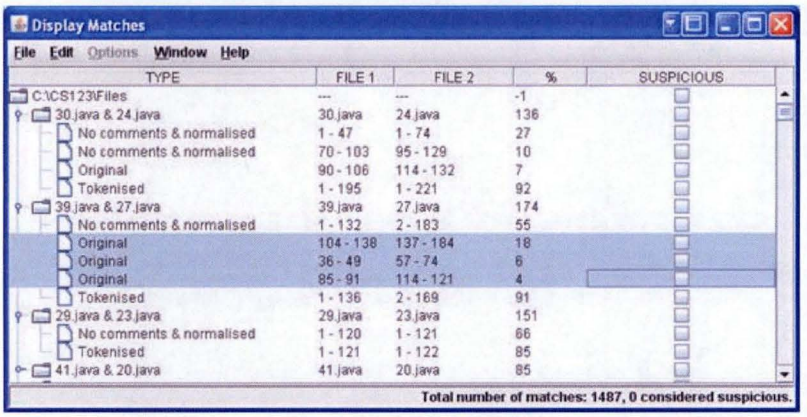

FIGURE 4.9 – *Sherlock* : arbre de paires de programmes

L'utilisateur peut par la suite ouvrir un écran permettant d'effectuer une comparaison côte à côte de la paire de programmes sélectionnés (figure 4.10). Le code source est affiché en haut de l'écran sous sa forme originale et sous la forme de lexèmes en bas de l'écran. Les parties identiques de code sont mises en corrélation sur la gauche de l'écran de comparaison. L'utilisateur a également la possibilité de marquer les paires douteuses pour une éventuelle impression (figure 4.9).

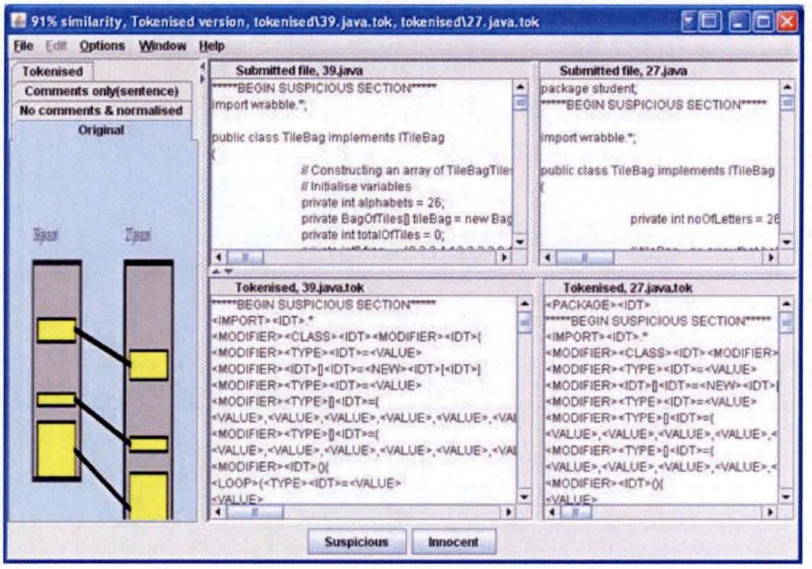

FIGURE 4.10 – *Sherlock* : écran de comparaison

Enfin, le système *Sherlock* permet à l'utilisateur d'explorer les résultats grâce à un graphe (figure 4.11) : chaque noeud représente une soumission et chaque arête symbolise au moins une correspondance entre deux soumissions.

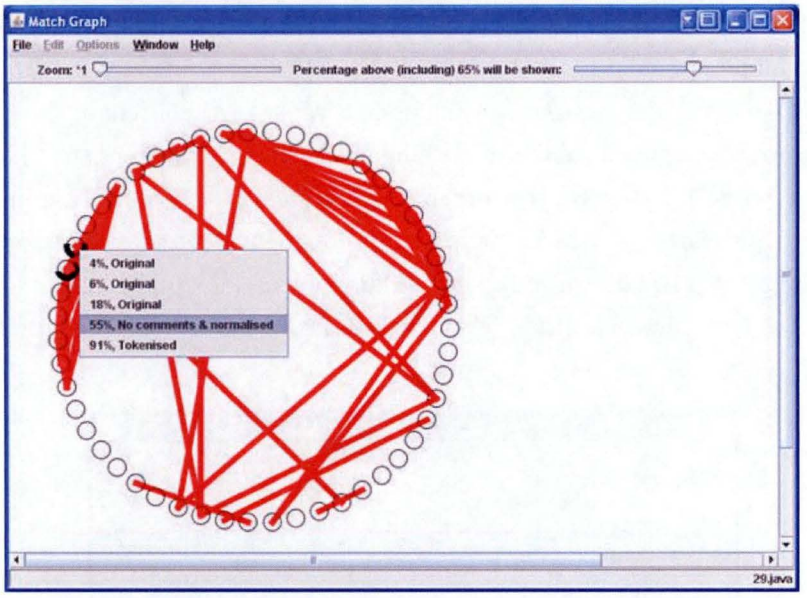

FIGURE 4.11 – *Sherlock* : graphe des résultats

4.2.4 *Copy/Paste Detector (CPD)*

Le système *Copy/Paste Detector (CPD)* est un autre outil de détection de plagiat de code source, produit par le projet *PMD*[54]. *PMD* est un outil d'audit de code Java et permet de contrôler le respect des bonnes pratiques requises par le langage Java, de détecter d'éventuels bugs ainsi que de trouver et d'éliminer du code redondant grâce à l'outil *CPD*. A l'inverse de *JPlag* ou *MOSS*, l'objectif prioritaire de *CPD* est de trouver du code dupliqué à l'intérieur de projets de programmation, néanmoins ses concepteurs affirment que leur système est capable de détecter également des cas de plagiat dans des travaux de programmation. *CPD* supporte les langages Java, JSP, C, C++, Fortran et PHP, et, similairement à *JPlag*, il implémente l'algorithme *Running-Karp-Rabin-Greedy-String-Tiling*. La documentation fournie par ses concepteurs est importante et explique notamment comment il est possible d'ajouter facilement de nouveaux langages supportés par *CDP*. L'utilisation de *CPD* s'effectue localement et nécessite une machine virtuelle Java. L'outil est capable de détecter aussi bien du code dupliqué à l'intérieur

d'un même fichier de code source mais aussi de détecter des similarités sur un ensemble de fichiers.

Initialement, l'utilisateur sélectionne le répertoire contenant la collection de programmes à analyser, le langage cible et la longueur de la plus grande chaîne de caractères à détecter (figure 4.12). *CDP* affiche ensuite, pour chaque paire de fichiers comparés, la longueur du plus grand segment de code trouvé ainsi que le code source identique au deux programmes (figure 4.13). Les résultats peuvent être sauvegardés sous la forme de fichiers texte ou xml.

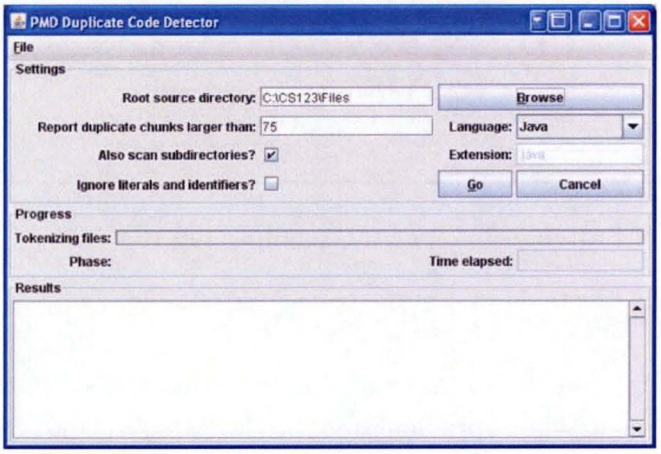

FIGURE 4.12 – *CDP* : écran initial

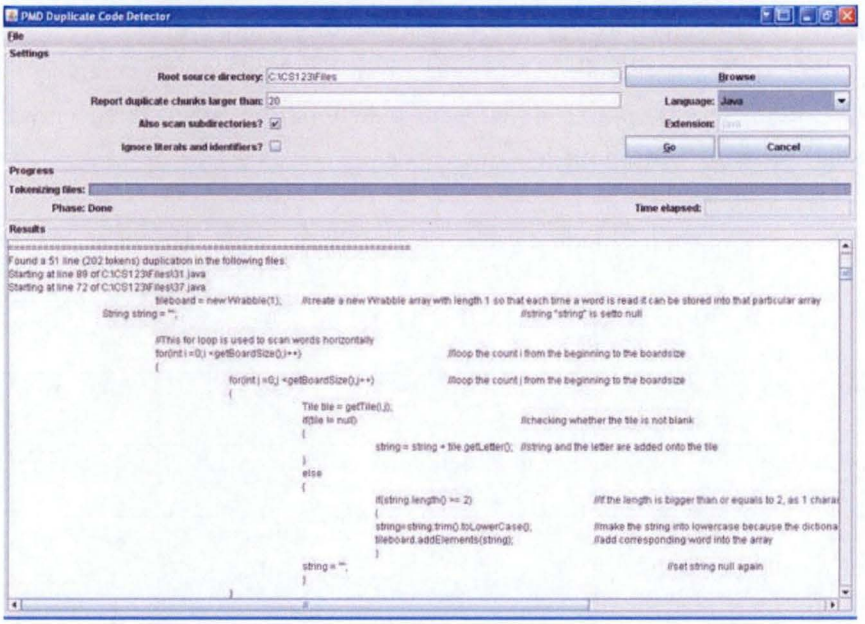

FIGURE 4.13 – *CDP* : affichage des résultats

Bien que *CDP* soit un outil simple, pratique et rapide à utiliser, celui-ci reste assez basique dans la mesure où son interface graphique ne permet pas d'identifier rapidement et facilement les cas de plagiat comme dans *MOSS* ou *JPlag*. Il n'est pas possible non plus à l'utilisateur de fournir du code commun et aucune étude à propos des résultats obtenus par *CDP* dans les cas de plagiat de code source n'est disponible dans la littérature. *CDP* est avant tout un outil de détection de code redondant (et pas un moteur de détection de plagiat de code source) et est estimé comme étant nettement inférieur à des systèmes comme *MOSS* ou *JPlag*.

4.3 Conclusion

D'importants progrès techniques dans le domaine du plagiat de code ont été réalisés durant ces dernières années : de nombreux détecteurs de plagiat de code source ont été et seront encore développés. Ce chapitre a présenté et détaillé quatre des moteurs modernes de détection de plagiat

les plus couramment cités parmi la communauté académique . Il apparaît cependant qu'il est difficile d'affirmer qu'un de ces systèmes surclasse les autres, aucune étude comparative formelle de fiabilité n'ayant été découverte dans la littérature abordant le sujet.

TABLE 4.1 – Comparaison de fiabilité des systèmes *JPlag*, *MOSS* et *Sherlock*

	JPlag	*MOSS*	*Sherlock*
Modification des identifiants	OUI	OUI	OUI
Modification de l'ordre des opérandes	OUI	OUI	OUI mais pas stable
Modification des types	Seulement pour Java	NON	NON
Remplacement des expressions par leurs équivalents	OUI	OUI mais pas stable	OUI mais pas stable
Ajout d'instructions et/ou de variables redondantes	Seulement pour Java	NON	OUI
Modification de l'ordre des instructions	OUI si quelques lignes de code source	Partiellement	OUI si quelques lignes de code source
Modification de la structure d'instructions de répétition	NON	NON	NON
Modification de la structure d'instructions de sélection	NON	NON	NON
Ajout d'instructions non-ordonnées	OUI si quelques lignes de code source	Partiellement	OUI si quelques lignes de code source
Combinaison de fragments du programme plagié et du programme original	OUI	OUI	OUI

Le tableau 4.1 résume les résultats d'une expérience réalisée en 2007 par *The Subject Centre for Information and Computer Sciences*[55]. Cette étude confronte les systèmes *JPlag*, *MOSS* et *Sherlock* avec les techniques de plagiat identifiées par Whale[12]. Ces résultats sont néanmoins à interpréter avec prudence dans la mesure où les auteurs de l'enquête n'expliquent pas la façon dont les tests ont été effectués (nombre de soumissions, ...).

Même si il est difficile de recommander un des systèmes détaillés dans ce chapitre, *MOSS* et *JPlag* sont généralement les outils les plus utilisés au sein des institutions académiques, et ce, malgré le problème de confidentialité précédemment expliqué. Il semble d'ailleurs que *MOSS* soit souvent préféré à *JPlag* pour des raisons de robustesse.

Finalement, les tableaux comparatifs (tableaux 4.2 et 4.4) confrontent les systèmes de détection de plagiat suivant une dizaine de caractéristiques (langages supportés, coût, sécurité, ...) et suivant la présentation des résultats (tableau 4.6).

TABLE 4.2 – Principales caractéristiques des différents moteurs modernes de détection de plagiat de code source

	JPlag	*MOSS*	*Sherlock*	*CPD*	*SID*
Date de création	1997	1994	1994	2003	2003
Langages supportés	Java, C#, C, C++, Scheme	C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, HCL2.	Java, C, C++	Java, JSP, C, C++, Fortan, PHP	Java, C++
Coût	Gratuit mais création d'un compte nécessaire	Gratuit mais création d'un compte nécessaire	Gratuit et open source	Gratuit et open source	Gratuit mais création d'un compte nécessaire
Service	internet	internet	local	local	internet
Interface	GUI	Ligne de commande et interface Web	GUI	GUI	Web

TABLE 4.4 – Principales caractéristiques des différents moteurs modernes de détection de plagiat de code source

	JPlag	*MOSS*	*Sherlock*	*CPD*	*SID*
Besoins	JVM	Perl	JVM	JVM	-
Sécurité	Identifiant utilisateur et e-mail nécessaires	Identifiant utilisateur et e-mail nécessaires	-	-	Identifiant utilisateur et e-mail nécessaires
Soumissions	Fichiers ou répertoire	Fichiers ou répertoire	Répertoire contenant un ou plusieurs fichiers compressés	Fichiers ou répertoire	Fichiers compressés
Possibilité d'exclure certaines sources	Oui	Oui	Oui	Non	Oui
Technique utilisée	Greedy String Tiling	Winnowing	Comparaison de chaînes de caractères	Greedy String Tiling	Compression

TABLE 4.6 – Méthodes d'affichage des résultats des différents moteurs modernes de détection de plagiat de code source

	JPlag	*MOSS*	*Sherlock*	*CPD*	*SID*
Format des résultats	Ensemble de pages HTML	Ensemble de pages HTML	Boîtes de dialogues interactives	Texte ou XML	Ensemble de pages HTML
Stockage des résultats	Localement	A distance	Localement	Localement	A distance
Méthode d'affichage général	Histogramme	Liste ordonnée	Arbre de paires correspondantes	Liste ordonnée	-
Méthode d'affichage détaillé	Comparaison côte à côte des fichiers suspects	Comparaison côte à côte des fichiers suspects		Code source identique affiché	-

Chapitre 5

Conclusion

Depuis déjà une trentaine d'années, des solutions informatiques sont constamment développées dans le but d'aider à la détection de plagiat de code source. Ce mémoire a proposé une synthèse des recherches qui ont été effectuées dans ce contexte. Pour apprécier l'utilité des ces solutions informatiques, appelées moteurs de détection de plagiat, une définition structurée du plagiat de code source a été présentée et les différentes techniques de modifications du code source utilisées par les plagiaires ont été mises en évidence. Plus précisément, l'identification des techniques de déguisement de code source a permis de préciser les fonctionnalités des moteurs de détection de plagiat.

Pour contrecarrer ces transformations de code, des techniques de détection de plagiat ont été inventées. Traditionnellement, dans la littérature, ces techniques sont répertoriées selon deux catégories : les systèmes à comptage d'attributs et les systèmes à métriques de structure. Ces deux systèmes ont été présentés, permettant, d'une part, de comprendre l'évolution des moteurs de détection de plagiat et, d'autre part, de démontrer que cette classification est inadéquate. Pour discuter des algorithmes de détection de plagiat de code source, une nouvelle classification a été établie, et quatre familles de techniques ont été dénombrées :

- Les techniques à comparaison d'empreintes

  - empreintes qualitatives
  - empreintes quantitatives
- Les techniques à comparaison de chaînes de caractères
- Les techniques à comparaison d'arbres syntaxiques
- Les techniques basées sur la compression de fichiers

Subséquemment, chacune de ces méthodes a été expliquée en détail ainsi que les algorithmes qui les implémentent.

Clairement, les techniques à comparaison d'empreintes quantitatives, anciennement nommées techniques à comptage d'attributs sont apparues moins fiables et donc obsolètes. Cependant, vu les lacunes des méthodes d'évaluation des techniques de détection de plagiat, il apparaît impossible d'affirmer qu'une technique est supérieure aux autres. Certains systèmes ont déjà probablement atteint leurs limites, par exemple il est difficile de croire qu'une technique à comparaison de chaînes de caractères meilleure que celle proposée par l'algorithme *Greedy String Tiling* pourrait être développée. D'un autre côté, les recherches concernant les techniques à comparaison d'arbres ou les techniques basées sur la compression ont émergées depuis peu et des améliorations pourraient probablement y être apportées.

Enfin, les moteurs actuels de détection de plagiat de code source ont été inventoriés dans la dernière partie de cette synthèse. Chaque système y a été présenté en détail de manière à ce qu'un utilisateur potentiel puisse se forger sa propre opinion sur un moteur en particulier recensé dans la littérature appropriée :

- *JPlag*, *Sherlock* et *Copy/Paste Detector* basés sur la technique à comparaison de chaînes de caractères
- *MOSS* utilisant la technique à comparaison d'empreintes qualitatives

Ces moteurs ont été ensuite comparés selon leurs caractéristiques principales :

- Langages supportés
- Coût

- Interface graphique
- Fiabilité

Même si le système *Copy/Paste Detector* apparaît certainement moins évolué que les autres moteurs et même si les institutions académiques préconisent majoritairement l'utilisation de *JPlag* ou de *MOSS*, affirmer qu'un moteur en particulier est supérieur à un autre s'avère impossible ; aucune étude comparative formelle de fiabilité n'ayant été découverte dans la littérature traitant le sujet.

Finalement, bien que de nombreux progrès aient été effectués dans la détection automatique de plagiat, une intervention humaine est toujours nécessaire pour prouver la présence de plagiat et pour émettre le jugement final.

Bibliographie

[1] B. Belkhouche, A. Nix, and J. Hassell, "Plagiarism detection in software designs," in *ACM-SE 42 : Proceedings of the 42nd annual Southeast regional conference.* New York, NY, USA : ACM, 2004, pp. 207–211.

[2] Perreault, "Le plagiat et autres types de triche scolaire à l'aide des technologies une réalité, des solutions," *Profweb*, 2007. [Online]. Available : http ://site.profweb.qc.ca/index.php ?id=87

[3] Urkund, "Les étudiants et l'intégrité à l'université," 2006. [Online]. Available : http ://www.urkund.com

[4] Giezendanner, "Le plagiat dans les systèmes éducatifs," *Département de l'instruction publique du Canton de Genève SEM*, 2007.

[5] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh, "Cheating and plagiarism : perceptions and practices of first year it students," *ACM SIGCSE Bulletin*, vol. 34, no. 3, pp. 183–187, 2002.

[6] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[7] Faidhi and Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Comput. Educ.*, 1987.

[8] M. Joy and M. Luck, "Plagiarism in programming assignments," *Education, IEEE Transactions on*, vol. 42, no. 2, pp. 129–133, 1999.

[9] A. Parker and J. Hamblen, "Computer algorithms for plagiarism detection," *IEEE Transactions on Education*, vol. 32 :2, pp. 94–99, 1989.

[10] U. Manber and U. Manber, "Finding similar files in a large file system," pp. 1–10, 1994.

[11] G. Cosma and M. Joy, "Source-code plagiarism : a uk academic perspective," *Research Report RR-422, Department of Computer Science, University of Warwick, Coventry, UK*, 2006.

[12] G. Whale, "Identification of program similarity in large populations," *The Computer Journal*, vol. v.33 n.2, pp. 140–146, 1990.

[13] P. Clough and D. O. I. Studies, "Old and new challenges in automatic plagiarism detection," pp. 391–407, 2003.

[14] P. Clough, "Plagiarism in natural and programming languages : an overview of current tools and technologies," *Research Memoranda : CS-00-05, Department of Computer Science, University of Sheffield*, 2000.

[15] M. Mozgovoy, "Desktop tools for offline plagiarism detection in computer programs," vol. 5, no. 1. Vilnius, Lithuania : Institute of Mathematics and Informatics, 2006, pp. 97–112.

[16] T. Lancaster and F. Culwin, "A comparison of source code plagiarism detection engines," *Computer Science Education*, vol. 14, no. 2, pp. 101–112, 2004.

[17] M. H. Halstead, "Elements of software science, elseveir." 1977.

[18] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[19] W. Harrison and K. Magel, "A complexity measure based on nesting level," *ACM SIGPLAN Notices*, vol. 16, no. 3, pp. 63–74, 1981.

[20] Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism." *SiGCSE Bulletin 8*, vol. 4, pp. 30–41, 1977.

[21] S. Grier, "A tool that detects plagiarism in pascal programs," in *SIGCSE '81 : Proceedings of the twelfth SIGCSE technical symposium on Computer science education.* New York, NY, USA : ACM, 1981, pp. 15–20.

[22] M. J. Verco, K. L. & Wise, "Software for detecting suspected plagiarism : Comparing structure and attribute-counting systems," *Proceedings of*

First Australian Conference on Computer Science Education, Sydney, Australia., vol. July 3-5 1996, 1996.

[23] E. L. Jones, "Metrics based plagarism monitoring," *J. Comput. Small Coll.*, vol. 16, no. 4, pp. 253–261, 2001.

[24] A. . L. T. Culwin, F. MacLeod, "Source code plagiarism in uk he computing schools, issues, attitudes and tools, south bank university technical report sbu-cism-01-02." 2001.

[25] S. Saxon, "Comparison of plagiarism detection techniques applied to student code, part ii computer science project, trinity college, cambridge, uk." 2000.

[26] M. J. Wise, "Yap3 : improved detection of similarities in computer program and other texts," *SIGCSE Bull.*, vol. 28, no. 1, pp. 130–134, 1996.

[27] M. Sallis, Aakjaer, "Software forensics : Old methods for a new science," in *Software Engineering : Education and Practice*, 1996, pp. 367–371.

[28] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing : local algorithms for document fingerprinting," in *SIGMOD '03 : Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* New York, NY, USA : ACM, 2003, pp. 76–85.

[29] R. Karp and M. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[30] Aiken, "A system for detecting software plagiarism." [Online]. Available : http ://theory.stanford.edu/ aiken/moss/

[31] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring similarity of large software systems based on source code correspondence," in *Proceedings of the 6th International Conference on Product Focused Software Process Improvement.* Springer, 2005.

[32] Wise, "Neweyes a system for comparing biological sequences using the running karp rabin greedy string tiling algorithm," pp. 393–401, 1995.

[33] R. Ahtiainen, Surakka, "Plaggie : Gnu-licensed source code plagiarism detection engine for java exercises," in *Proceedings of the 6th Baltic Sea*

conference on Computing education research : Koli Calling 2006. ACM New York, NY, USA, 2006, pp. 141–142.

[34] Wise, "Running rabin-karp matching and greedy string tiling," *Basser Department of Computer Science Technical Report*, 1994.

[35] D. Gitchell and N. Tran, "Sim : a utility for detecting similarity in computer programs," *SIGCSE Bull.*, vol. 31, no. 1, pp. 266–270, 1999.

[36] M. Frohlich and M. Werner, "Demonstration of the interactive graph-visualization system da vinci," *Lecture Notes in Computer Science*, vol. 894, pp. 266–269, 1995.

[37] C. Bennett, P. Vitanyi, M. Zurek, W. Res, and Y. Center, "Information distance," *Information Theory, IEEE Transactions on*, vol. 44, no. 4, pp. 1407–1423, 1998.

[38] J. Varre, "Transformation distances : a family of dissimilarity measures based on movements of segments," *Bioinformatics*, vol. 15, no. 3, pp. 194–202, 1999.

[39] R. Cilibrasi and P. Vitanyi, "Clustering by compression," *Information Theory, IEEE Transactions on*, vol. 51, no. 4, pp. 1523–1545, 2005.

[40] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 1997.

[41] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, "The similarity metric," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2003, pp. 863–872.

[42] Rooij, "Findfraud." [Online]. Available : http ://homepages.cwi.nl/ rooij/programming/findfraud/

[43] Wassner, "Baldr, l'outil anti-fraude et anti-plagiat." [Online]. Available : http ://professeurs.esiea.fr/wassner/ ?2007/06/15/75-baldr-l-outil-anti-fraude-anti-plagiat

[44] A. Kolmogorov and S. Fomin, "Elements of the theory of functions and functional analysis, vol. 1, metric and normed spaces," *Bull. Amer.*

Math. Soc. 64 (1958), 198-202. DOI : 10.1090/S0002-9904-1958-10210-4 PII : S, vol. 2, no. 9904, pp. 10 210–4, 1958.

[45] J. Delahaye, "Classer musiques, langues, images, textes et genomes," *Pour La Science*, vol. 317, no. 98-103, pp. 7–1, 2004.

[46] Chen, "Sid plagiarism detection website." [Online]. Available : http ://genome.math.uwaterloo.ca/SID/

[47] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," *Information Theory, IEEE Transactions on*, vol. 50, no. 7, pp. 1545–1551, 2004.

[48] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[49] M. Mozgovoy, K. Predriksson, D. White, M. Joy, and E. Sutinen, "Fast plagiarism detection system," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 3772, p. 267, 2005.

[50] M. Mozgovoy, "Enhancing computer-aided plagiarism detection," 2007.

[51] Malpohl, "Jplag - detecting software plagiarism." [Online]. Available : https ://www.ipd.uni-karlsruhe.de/jplag/

[52] Sun, "Java se desktop technologies." [Online]. Available : http ://-java.sun.com/javase/technologies/desktop/javawebstart/index.jsp

[53] U. of Warwick, "Boss online submission system." [Online]. Available : http ://www.dcs.warwick.ac.uk/boss/

[54] Dixon-Peugh, "Pmd." [Online]. Available : http ://pmd.sourceforge.net/

[55] T. S. C. for Information and C. Sciences, "Performance comparison of source code tools." [Online]. Available : http ://www.ics.heacademy.ac.uk