Titre: Title:	Analyse des propriétés structurelles et computationnelles des clones logiciels
Auteur: Author:	Thierry M. Lavoie
Date:	2011
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lavoie, T. M. (2011). Analyse des propriétés structurelles et computationnelles des clones logiciels [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/660/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/660/
Directeurs de recherche: Advisors:	Ettore Merlo
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

ANALYSE DES PROPRIÉTÉS STRUCTURELLES ET COMPUTATIONNELLES DES CLONES LOGICIELS

THIERRY M. LAVOIE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU
DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DES PROPRIÉTÉS STRUCTURELLES ET COMPUTATIONNELLES DES CLONES LOGICIELS

présenté par : LAVOIE Thierry M..
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. GUÉHÉNEUC, Yann-Gaël, Doct., président
M. MERLO, Ettore, Ph.D., membre et directeur de recherche
M. ANTONIOL, Giuliano, Ph.D., membre

À tous les gens qui me donnent des idées,
sans qu'ils le sachent…

REMERCIEMENTS

Tout le travail présenté ici n'aurait pas pu être réalisé directement ou indirectement sans les diverses contributions des gens et des organismes suivants.

Tout d'abord, je tiens à remercier chaleureusement mon mentor, le Professeur Ettore Merlo, superviseur passé, présent et futur de mes travaux de recherche depuis le milieu de mon baccalauréat. Merci de toujours être aussi dévoué et impliqué dans mes travaux.

Je veux ensuite remercier les membres de mon jury qui se sont acquittés diligemment de leur tâche.

Je tiens également à remercier le CRSNG et le FQRNT de m'avoir accordé les bourses nécessaires à la continuité de mes études ainsi qu'aux différents voyages de conférence auxquels j'ai participé.

Je ne peux passer sous silence la contribution toujours aussi présente de mes parents, Murielle et Daniel. Merci d'être encore dans cette aventure après 6 ans.

Un merci particulier à sensei Richard Goulet et à tous les gens avec qui je pratique le kendo, sans qui mon travail cérébral aurait eu raison de ma santé physique depuis longtemps.

Et, finalement, un merci à mes collègues Dominic, François et Maxime et à mes amis Simon, David, Daniel, Catherine, Amélie et Patricia. Vous faites de la vie une expérience si agréable.

RÉSUMÉ

La détection des clones est une discipline dynamique du génie logiciel consacrée à la recherche et l'analyse de similitude à l'intérieur des logiciels. Ces similitudes peuvent être à l'intérieur de tous les artefacts d'un produit logiciel, mais cette discipline se consacre principalement à celles présentes dans le code source. Les travaux présentés dans ce mémoire sont consacrés à l'étude de certaines propriétés structurelles et computationnelles des clones logiciels syntaxiques et lexicaux ; les clones sémantiques ne sont pas formellement abordés dans ces travaux.

Dans un premier temps, une analyse des relations structurelles entre les clones a été réalisée. Les clones considérés sont produit par la méthode de détection des clones basée sur les métriques d'AST. Contrairement à la très grande majorité des autres travaux présents dans la littérature, qui s'intéressent exclusivement aux fonctions ou aux méthodes comme unité minimale de clonage, ces travaux s'intéressent à la fois aux relations impliquant des fonctions et des blocs, engendrant ainsi trois relations d'intérêt : les clones fonction à fonction, bloc à bloc et fonction à bloc. Les conclusions de l'étude indiquent clairement l'intérêt de s'intéresser à ces relations et l'insuffisance de la réduction de l'espace de recherche composé de tous les fragments de code d'un programme à celui restreint aux fonctions. Les résultats présentés dépendent seulement partiellement de la méthode ; l'intérêt des relations présentées est indépendant tandis que les résultats spécifiques présentés sont dépendants.

Dans un second temps, une étude a été réalisée sur l'applicabilité d'algorithmes écrits sur GPU pour le problème de postfiltrage des clones. Le filtrage est une pratique répandue dans les outils de détection de clonage pour améliorer la précision des résultats. La technique choisie ici est celle du filtrage avec le calcul de la plus longue sous-séquence commune (LCS). Cet algorithme se parallélise aisément et se porte sans heurt sur une architecture GPU ; il est donc un candidat idéal. La conversion de l'algorithme classique de programmation dynamique pour le calcul de la LCS est présenté avec plusieurs considérations techniques pour faciliter son implémentation. Les résultats de l'application de l'algorithme sur deux systèmes de grande taille, Tomcat et Eclipse, sont présentés. Les temps d'exécution de l'algorithme sur GPU sont inférieurs à ceux obtenus sur CPU, même si le facteur d'accélération total est relativement petit (de l'ordre de 1.2). Cette accélération est inférieure aux attentes et quelques pistes d'explication sont fournies.

Dans un dernier temps, la réalisation d'oracles automatisés basés sur la distance de Levenshtein est présentée. Pour la détection des clones, un oracle consiste en un ensemble de clones jugé exact pour fin de comparaison de différents outils. Ce résultat est une grande nouveauté puisqu'il prouve l'applicabilité de la distance de Levenshtein sur de grands systèmes en utilisant une quantité raisonnable de ressources. Pour arriver à calculer ces oracles, les fragments des systèmes sont

insérés dans un arbre de métriques qui se construit en répartissant les fragments en fonction de leur distance de Levenshtein. L'utilisation des arbres de métriques dans des expériences de détection des clones est une nouveauté et les oracles obtenus sont prometteurs. Différentes statistiques sur les oracles sont présentées. On remarque selon ces chiffres que le nombre de pairs de clones est très petit par rapport au nombre de paires total possible dans un système. Cela explique le gain de performance notable de la nouvelle méthode présentée par rapport à une recherche exhaustive.

ABSTRACT

Clone detection is a widely known research area of software engineering, which goal is to enhance programs understanding and detect potential bugs as well as re-factoring opportunities. The focus of this research area is on finding and analyzing self-similarities in source code. This thesis focuses on structural and computational properties of clones.

First, a structural classification scheme of clones is presented. Many studies have researched properties of function clones. These studies did not take in consideration the possible presence of clones in smaller or bigger computational unit. Moreover, they relied on the hypothesis that clone code preserves completely the syntactic structure between fragments. To challenge this hypothesis, classification scheme and algorithm are presented in this work and are applied to clone sets of different systems. Results show that clones can belong to at least three types of structural relationship: method-method, block-block, and method-block. Examples are provided and support the usefulness of such classification. As a result, it mitigates the choice of performing clone analysis only on functions and suggest to focus on other computational units.

Second, an attempt to port some clone-related computation to GPU is presented. To enhance results quality, it is often required to filter clone analysis results with precise but computationally expensive algorithm. The length of the longest common sub-sequence is a good candidate for such a filter and is used by many in the literature. However, its computational cost is prohibitive in many cases. To overcome the cost, a port on GPU of the algorithm specialized for clone detection is proposed. The observed execution times suggest however that little gain is obtained for clone filtering problems. Discussion of this unexpected result is provided along with ideas of other possible GPU applications to clone detections.

Finally, an original algorithm to compute clone oracles is presented. An oracle is a set of clones presumed to be of good quality and used to benchmark clone detection tools results quality. This result is of great importance to the clone community as it provides the first automatically-generated oracle based on an objective algebraic criterion. It also gives insight on computational difficulties of computing good clone results. The key point of the technique is to observe the computational bottleneck of exhaustive search is the space search size and not the distance computation, even for complex distance. Using this knowledge, a metric tree-based algorithm is presented to compute oracles. Details of the computed oracles are presented.

After these topics discussions, the thesis is concluded with future research possibilities.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AST	Arbre de Syntaxe Abstrait (*Abstract Syntax Tree*)
MM	Méthode-Méthode
BB	Bloc-Bloc
MB	Méthode-Bloc
FIB	Forêt d'inclusion des blocs
CPU	Central Processing Unit
GPU	Graphic Processing Unit
LCS	Plus longue sous-séquence commune (*Longest Common Subsequence*)
IDE	Environnement intégré de développement (*Integrated Development Environment*)
SISD	Une donnée, une instruction (*Single Instruction, Single Data*)
SIMD	Plusieurs données, une instruction (*Multiple Instructions, Single Data*)
MIMD	Plusieurs données, plusieurs instructions (*Multiple Instructions, Multiple Data*)
GLSL	OpenGL Shading Language
API	Application Program Interface
LOC	Lignes de code (*Line of Code*)

CHAPITRE 1

INTRODUCTION

La compréhension du logiciel, la découverte automatisée de bogues, la refactorisation et la détection du plagiat sont quelques-uns des problèmes les plus importants du génie logiciel. Depuis une vingtaine d'années, la détection de similitudes internes dans les systèmes, appelées clones, fournit différentes pistes de solution pour les problèmes sus-mentionnés. Malgré un progrès constant des outils de détection des clones, les résultats obtenus au cours des années n'ont pas démontré clairement leur fiabilité et leur utilité et la communauté du génie logiciel n'intègre pas encore les pratiques proposées par la communauté des clones. Cela est dû à deux problèmes : l'incapacité de ces outils à produire des résultats de grande qualité (taux élevés de faux-positifs et faux-négatifs) et le choix des relations des clones étudiées. Bien entendu, les outils disponibles aujourd'hui produisent des résultats intéressants d'un point de vue théorique, mais l'applicabilité industrielle fait toujours défaut, faute de résultats significatifs. Les recherches présentées ici visent à analyser certaines propriétés structurelles et computationnelles des clones afin d'améliorer la compréhension de la nature des clones. Les connaissances découvertes pourront servir à calibrer les outils de détection de clones existants et à orienter les efforts de construction de nouveaux outils. Les trois problèmes étudiés ici sont : la nature des relations structurelles syntaxiques des clones, l'applicabilité des GPU pour le postfiltrage et l'utilisation des arbres de métriques pour construire des oracles avec la métrique de Levenshtein.

1.1 Définitions et concepts de base

1.1.1 Les fragments

Les fragments sont des ensembles ordonnés de caractères provenant d'un même fichier d'un système. Les caractères contenus dans les fragments sont associés au numéro de la ligne et au numéro de la colonne où ils sont localisés dans le fichier d'origine. L'ensemble des 2-tuples $(ligne, colonne)$ associés aux caractères ordonné lexicographiquement donne l'ordre des caractères dans le fragment. Un fragment ne peut être discontinu ; s'il existe un caractère associé au 2-tuple (i, j) dans le fichier original et que :

$$(i_0, j_0) \leq (i, j) \leq (i_n, j_n) \tag{1.1}$$

où (i_0, j_0) est le plus petit 2-tuple associés aux caractères du fragment et (i_n, j_n) le plus grand,

alors le caractère associé à (i, j) doit appartenir au fragment. Ceci est une définition très générale d'un fragment de code. Il est commun de restreindre la notion de fragments au texte associé aux fonctions. Cependant, la notion de fonction varie selon la grammaire utilisée pour rédiger le programme analysé et il devient donc nécessaire au préalable d'analyser syntaxiquement le système, ce qui est une contrainte mineure. Dans un le cas ou un fragment est syntaxiquement une fonction, on dit que le fragment est associé à une fonction ou tout simplement on parle de la fonction. Il est plus rare de définir des fragments associés à d'autres unités logiques du programme. Par exemple, il peut être utile d'associer des fragments à des pairs d'accolades ouvrantes et fermantes. Syntaxiquement parlant, ces pairs forment des blocs logiques à l'intérieur du code. Les fonctions étant consitituées de blocs, on peut donc voir les fonctions comme une extension naturelle des blocs logiques. Cependant, comme les fonctions sont des blocs remarquables, une distinction sera toujours faite lorsque le fragment est associé à une fonction.

Le formalisme utilisé ici n'est nécessaire que pour fournir une définition algébrique des fragments. La définition intuitive que le lecteur a d'un fragment satisfait probablement trivialement la définition proposée ici. Cette définition n'est nécessaire que par souci de clarté mathématique.

1.1.2 Les clones, les pairs de clones et les types de clones

Un clone est un fragment de code source qui est similaire à un autre fragment de code présent dans le même système. Il pourrait exister des similitudes entre les fragments de différents systèmes informatiques, mais dans le cadre de cette recherche ce cas n'a pas été étudié. Par conséquent, pour le reste de cet exposé, lorsque qu'on parle de deux fragments clones l'un à l'autre, on assume que les deux fragments proviennent du même système. On définit une relation de clonage, ou pair de clones, comme un 2-tuple (f_0, f_1) dans lequel f_0 et f_1 sont des fragments de code source.

Les pairs de clones étant définis, il est maintenant facile de donner une définition précise d'un clone. Un clone est un fragment membre d'une relation de clonage (une paire de clones). Évidemment, cette définition n'est que terminologique, puisque rien ne définit encore la nature de la relation qui unit deux fragments membres d'une paire de clones.

La littérature et la communauté des clones s'entendent sur l'existence d'au moins quatre types de relation de clonage :

- type 1 : clone identique ;
- type 2 : clone paramétrique ;
- type 3 : clone partiel ;
- type 4 : clone sémantique.

Les clones de type 1 forment la plus simple des catégories. Il s'agit des clones dont le texte des fragments est strictement identique. De par la simplicité de leur définition, les clones de type 1 sont les plus simples à détecter.

Les clones de type 2 contiennent des paramètres en commun. Paramètre doit être ici pris au sens large ; il peut s'agir autant d'un paramètre formel différent dans les deux fragments, mais situé au même endroit dans les deux clones, que d'un identificateur dont le nom a changé d'un fragment à un autre. On peut voir un clone de type 2 comme des fragments dont on peut extraire et modifier un paramètre qui transformerait alors les fragments en clone de type 1.

La définition des clones de type 3 et 4 est beaucoup plus vague. Le type 3 regroupe les clones dont les fragments ne partagent pas toutes les mêmes unités syntaxiques. En ce sens, un clone de type 3 est une copie partielle d'un autre fragment qui a pu subir des transformations (opérations sur les caractères : substitution, ajout, suppression, permutation, etc.) tout en préservant clairement les structures syntaxiques présentes dans le fragment d'origine. Cette définition n'est pas algébrique et porte à interprétation. Malheureusement, aucun consensus n'est atteint parmi les chercheurs. Pour les besoins présents, la définition présentée est suffisante malgré ses lacunes. Une attention particulière aux clones de type 3 est apportée dans la section 4. Le type 4 est encore plus vague. Il porte sur les aspects sémantiques des clones. Il n'en existe aucune définition algébrique. L'idée générale est de regrouper dans cette catégorie tous les fragments qui partagent au moins partiellement la même sémantique d'exécution. Ce type n'est d'aucun intérêt pour les recherches présentées ici et sera ignoré dans le reste du texte.

1.2 Éléments de la problématique

Les recherches présentées sont divisées en trois problématiques distinctes. Tout d'abord, l'emphase est porté sur la nature syntaxique des relations de clonage. Il ne faut pas confondre ici la nature syntaxique des relations avec les catégories présentées. La nature syntaxique s'intéresse à la position des fragments dans l'AST du programme. Par exemple, il est courant de voir des fragments être des fonctions dans un système. La majorité des analyses de clonage se concentrent exclusivement sur les relations de nature fonction à fonction. Or, plusieurs fonctions forment des unités syntaxiques de grande taille qui peuvent en fait abriter des clones plus petits que l'unité entière. Si l'unité principale n'est pas un clone, mais qu'elle contient des fragments clonés, alors il peut y avoir un intérêt à les détecter. Cette recherche propose d'explorer cet aspect structurel des relations de clonage.

Ensuite, plusieurs outils connus de détection de clonage offrent des résultats de faible précision et utilisent un filtrage *a posteriori* des résultats pour en augmenter la qualité (voir [1] pour plusieurs exemples). Ce filtrage est souvent fait sur la base du calcul de la LCS entre les fragments (ou de la distance de Levenshtein). Ce calcul peut être très exigeant en temps. Il est donc nécessaire d'améliorer les performances de cette méthode ou d'en concevoir une nouvelle plus efficace. L'approche retenue ici est l'amélioration des performances de la méthode actuelle. En effet, le calcul de la LCS utilise un algorithme dont les caractéristiques offrent l'opportunité d'exploiter une approche parallèle. Le problème général avec les approches parallèles est la nécessité d'utiliser une grappe de calcul de plusieurs centaines de processeurs ; de telles grappes sont coûteuses à obtenir et à maintenir et sont donc peu accessibles. Il existe toutefois des composantes plus répandues qui offrent une puissance de calcul parallèle tout aussi intéressante : les GPU. C'est avec cette architecture que le problème de filtrage est traité ici.

Finalement, un problème chronique de la communauté est l'incapacité d'évaluer de manière fiable et à grande échelle les techniques connues de détection de clonage en mesurant leur taux de faux-positifs et de faux-négatifs. Actuellement, toutes les méthodes d'évaluation partagent les mêmes bases : l'injection artificielle de clones et la comparaison entre outils sans référence externe établie. La validation des techniques repose donc sur une logique circulaire : les techniques existantes doivent détecter ce qu'elles sont capables de détecter. Chaque outil ayant une définition algébriquement différente des clones, l'ensemble considéré valide devient alors un amalgame des clones détectés par chacun des outils en ajustant l'ensemble obtenu au gré de la volonté subjective des chercheurs. Il s'agit évidemment d'un grave problème méthodologique. Le problème est particulièrement flagrant lorsqu'il s'agit d'évaluer la qualité des outils qui détectent les clones de type 3, dont la définition est vague et varie d'un chercheur à l'autre. Contourner ce problème requiert une définition algébrique qui englobe des clones de type 3 raisonnablement reconnu comme tel. La dis-

tance de Levenshtein permet de formuler quelques définitions intéressantes de clones et sa fiabilité fait l'objet d'un certain consensus dans la communauté. Il est donc envisageable de construire une base de données de référence de toutes les paires de clones d'un système en se basant sur la distance de Levenshtein. Construire cette référence nécessite une recherche exhaustive des candidats dans l'espace des pairs de fragments d'un système, ce qui est impraticable. Une solution originale et exacte basée sur la réduction de l'espace de recherche par arbre de métriques est présentée ici. L'évaluation de la méthode est présentée en construisant l'oracles des clones de Levenshtein pour les systèmes Tomcat et Eclipse.

1.3 Objectifs de recherche

Les objectifs poursuivis dans ces recherches sont les suivants :

– Établir l'existence et l'utilité d'étudier les relations syntaxiques entre les clones outre la relation fonction à fonction ;
– déterminer l'applicabilité d'une approche de filtrage sur GPU des clones, particulièrement en terme de gain computationnel ;
– décrire une technique de construction automatisée d'une base de données de référence, un oracle, pour les clones de type 3.

Tableau 1.1 Liste des publications associées aux chapitres du mémoire

Chapitre	Publications
Chapitre 1	[2]
Chapitre 2	[3]
Chapitre 3	[4]

1.4 Plan du mémoire

Ce mémoire est divisé en trois grandes sections. La première est consacrée à l'étude de la nature syntaxique des relations de clonage, la seconde à l'applicabilité des GPU dans la détection de clonage et la troisième à la construction automatisée d'oracles pour les clones de type 3. Le contenu de chaque section est entièrement indépendant de celui des autres et suit son propre développement. À chaque section, un bref rappel introductif est présenté suivi d'un résumé des références littéraires pertinentes au thème présenté. Suit dans tous les cas d'une présentation des concepts importants et des algorithmes développés pour résoudre chacun des problèmes. Les expériences et les résultats sont ensuite présentés suivis d'une discussion et d'une conclusion. Une conclusion globale suit la dernière section et présente les pistes de recherches futures.

Les résultats présentés dans les chapitres de ce mémoire ont été publiés dans quelques conférences. Le tableau 1.1 indique l'association de chacun des chapitres à ses publications.

CHAPITRE 2

ANALYSE DES RELATIONS STRUCTURELLES SYNTAXIQUES DE CLONAGE

2.1 Introduction aux clones par métriques et aux relations de clonages

La détection des clones basée sur les métriques est essentiellement concentrée sur la détection de similitudes entre fonctions ou entre méthodes [5, 6]. Bien que les relations de clonage entre blocs syntaxiques arbitraires aient été étudiées entre autres par Bellon et al. [1], l'existence et la pertinence de ces relations restent sous analysée.

Une relation d'inclusion d'ensembles des blocs syntaxiques extraits du code source d'un système peut aisément être calculée à partir de l'AST représentant le code. La représentation de cette relation d'inclusion entre les blocs syntaxiques est une forêt d'arbres baptisée Forêt d'Inclusion des blocs, FIB. L'extraction des FIB à partir de l'AST nécessite un temps linéaire sur la taille de l'AST ; par conséquent, on peut déduire qu'il est indirectement linéaire sur la taille du système en lignes de code.

Deux types structurels de blocs sont définis à l'intérieur des FIB : les blocs à la racine des arbres d'inclusion sont appelés $methodes$ et les autres noeuds sont simplement appelés $blocs$. On utilise $methodes$ au lieu de $fonctions$, car le langage étudié est Java et qu'il n'y a que des méthodes en Java. Cette terminologie s'impose naturellement et s'adapte à la forte majorité des cas de classifications structurelles, mais il existe en Java des classes internes incompatibles avec cette nomenclature. Ce cas particulier sera analysé dans la section 2.5.

À partir de cette nomenclature des fragments de code, on définit trois relations de clonage fondées sur la combinaison de l'appellation des blocs : $methode - methode$ (MM), $bloc - bloc$ (BB) et $methode - bloc$ (MB).

Un des buts recherchés en proposant cette classification des clones basée sur les relations structurelles est d'en vérifier l'existence et l'ampleur dans des logiciels existants. Pour ce faire, une évaluation expérimentale de la classification a été effectuée sur deux systèmes Java de grande taille : Tomcat et Eclipse.

La méthode de détection de clonage utilisée pour appliquer la classification proposée est celle des clones par métriques. Cette méthode a fait l'objet de nombreuses publications ([5, 6, 7, 8, 9, 10, 11]). Il existe plusieurs autres méthodes de détection des clones telles que celles présentées dans [12, 13, 14, 15, 16, 17, 18, 19, 20]. Ces méthodes ont fait l'objet de plusieurs études empiriques et de plusieurs évaluations de la qualité dans [1, 21, 22, 23, 24, 25, 26].

D'autres références pertinentes incluent l'application à grande échelle des outils de détection

de clonage telle que présentée dans [27, 28], l'étude de l'évolution des clones à travers la vie et les versions des logiciels dans [29, 30] et l'étude d'applications industrielles dans le monde des affaires [31] et l'industrie automobile [32]. Finalement, deux analyses complètes de la littérature actuelle des clones peuvent être trouvées dans [33, 34].

Le reste de ce chapitre est divisé comme suit. La section 2.2 explique à l'aide d'un exemple la classification proposée ainsi que l'application de l'algorithme à cet exemple pour classifier les clones de l'ensemble présenté. La section 2.3 présente l'algorithme qui calcule la classification de chaque pair de clones détectée dans un système. La section 2.4 présente les expériences et les résultats obtenus avec la classification. La section 2.5 interprète les résultats et discute des problèmes liés à cette méthode et la section 2.6 résume le contenu du chapitre et propose de nouvelles pistes de recherche.

2.2 Exemple

Imaginons un système composé des méthodes $A,\ B,\ C,\ D$ et E dont la structure syntaxique est schématisée sur les figures 2.1 et 2.2. Dans ce système, la méthode A est similaire à B et la méthode C est similaire à D.

```
1  A(Z...) {          1  B(Z...) {
2     if (W...) {     2     if (W...) {
3        X...         3        X...
4     } else {        4     } else {
5        Y...         5        Y...
6     }               6     }
7  }                  7  }
```

Figure 2.1 Structure syntaxique des méthodes A et B

```
1  C(Z...) {          1  D(Z...) {          1  E(Z...) {
2     if (W...) {     2     if (W...) {     2     X...
3        X...         3     X...            3  }
4     }               4     }
5  }                  5  }
```

Figure 2.2 Structure des méthodes C, D et E

Tout au long de l'exemple, les blocs et les méthodes sont identifiés par leur nom en majuscule suffixée en indice par les identifiants numériques des lignes de début et de fin associée à une paire cohérente d'accolades ouvrante et fermante. Par exemple, $A_{2,4}$ identifie le bloc de la méthode A dont le début est situé à l'accolade ouvrante de la ligne 2 et dont la fin est située à l'accolade

fermante de la ligne 4. Par construction de l'exemple, aucune confusion ne peut exister puisqu'une ligne ne peut contenir qu'une seule accolade ouvrante ou fermante. La première paire cohérente d'accolades ouvrante et fermante dans une méthode est associée à la méthode et les autres aux blocs qui la composent.

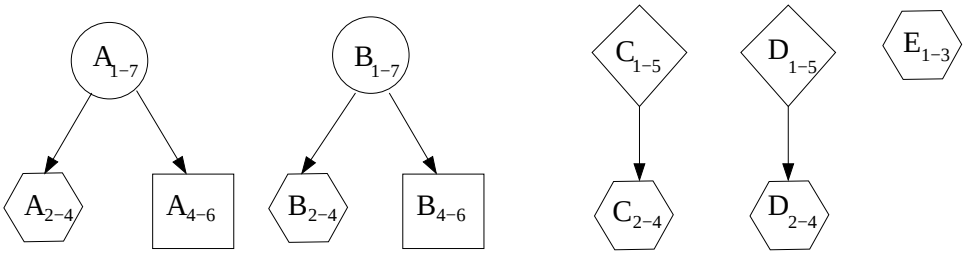


Figure 2.3 Imbrication de la structure de sous-arbre

La figure 2.3 montre les FIB correspondant aux méthodes $A,\ B,\ C,\ D$ et E. Les noeuds de la figure 2.3 sont représentés par des polygones afin de mettre en relief la relation de similitude entre les différents blocs. Deux blocs sont similaires si et seulement si la forme polygonale choisie pour représenter leur noeud est identique.

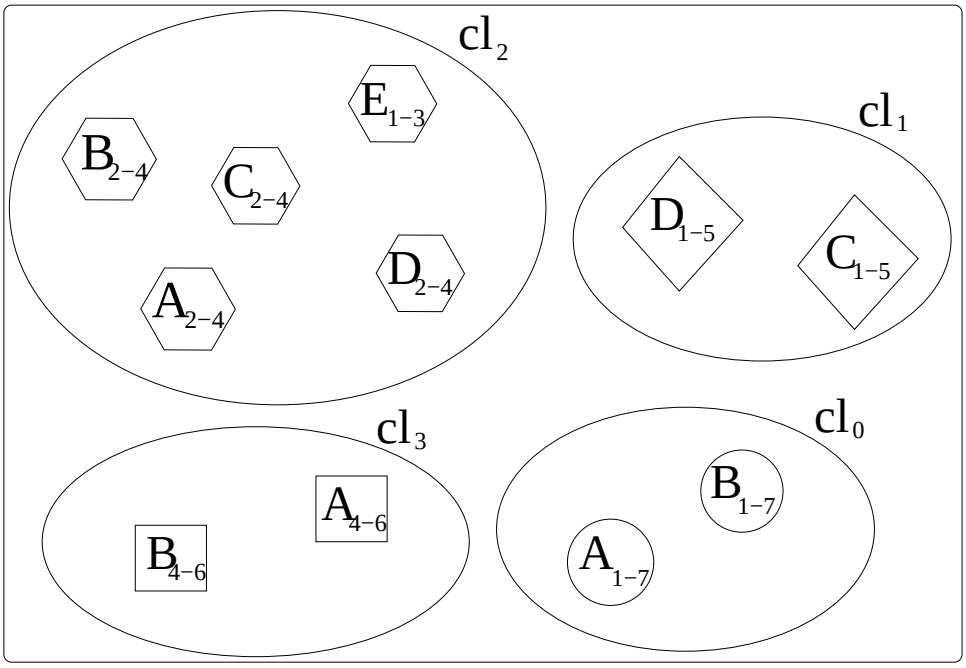


Figure 2.4 Partitions des clones

La figure 2.4 regroupe les fragments en grappe de similitude nommée $cl_0,\ cl_1,\ cl_2,\ cl_3$. Une grappe contient des fragments clones à tous les autres fragments de la grappe. On peut facilement

constater que les blocs A_{1-7} et B_{1-7} appartiennent à la grappe cl_0, les blocs C_{1-5} et D_{1-5} appartiennent à la grappe cl_1, les blocs A_{2-4}, B_{2-4}, C_{2-4}, D_{2-4} et E_{1-3} appartiennent à la grappe cl_2 et les blocs A_{4-6} et B_{4-6} appartiennent à la grappe cl_3.

La paire $(A_{1-7},\ B_{1-7})$ est un clone de type MM, car ses fragments sont deux méthodes qui n'ont aucun parent dans la FIB

La paire $(A_{2-4},\ C_{2-4})$ est une relation de type BB puisque ses éléments possèdent des ancêtres dans la FIB, mais que ces ancêtres ne sont pas clones entre eux. On remarque que le parent de A_{2-4}, A_{1-7}, est membre de cl_0 alors que le parent de C_{2-4}, C_{1-5}, est membre cl_1. La paire $(A_{2-4},\ C_{2-4})$ n'est donc pas redondante.

La paire $(A_{2-4},\ B_{2-4})$ appartenant à la grappe cl_2 est redondante, car les parents de ses éléments sont des clones. En effet, A_{1-7}, parent de A_{2-4} et B_{1-7} parent de B_{2-4} sont membres de la grappe cl_0 et forment une paire de clones de type MM. Par conséquent, il serait incorrect de classifier $(A_{2-4},\ B_{2-4})$ comme une paire de type BB, contrairement à la paire $(A_{2-4},\ C_{2-4})$ mentionnée plus haut. Il est important de souligner qu'il n'existe pas de contradiction à ne rejeter qu'une seule des deux paires même si A_{2-4} appartient aux deux paires mentionnées. La classification proposée se fait au niveau des pairs et non au niveau des grappes.

La paire $(B_{2-4},\ E_{1-3})$ illustre le dernier type de relation de clonage de la classification : le type BM. Le fragment B_{2-4} est un bloc imbriqué dans la méthode B alors que E_{1-3} est une méthode. On peut observer que, contrairement aux relations de type BB, il est impossible qu'une relation de type BM soit redondante, puisqu'un fragment de type méthode ne peut avoir, par définition, un parent dans la FIB. Trivialement, on peut faire la même observation pour les relations de type MM.

La liste exhaustive de toutes les paires de clones appartenant aux grappes de la figure 2.4 ainsi que leur classification est donnée dans le tableau 2.2. Les relations classifiées R sont des relations de type BB redondantes.

2.3 Description de la classification structurelle des clones

Soit la FIB d'un système dont la relation de similitude entre les différents fragments de code est une relation d'équivalence $clone(f_i, f_j)$ entre les fragments f_i et f_j. Il en résulte une partition de l'espace de tous les fragments en grappe cl_k mutuellement exclusive (les grappes sont des classes d'équivalence). Les propriétés nécessaires à la relation d'équivalence sont généralement satisfaites par la majorité des techniques de détection de clonage, dont la détection basée sur les métriques, sur la similitude des préfixes et des suffixes et sur la similitude des AST.

Les classes d'équivalence résultantes peuvent être de quatre types : BB, BM, MB et MM. Le type MB est équivalent au type BM, mais il s'applique aux pairs ordonnées pour lesquelles le

Tableau 2.1 Exemples de classification selon les différents types proposés

Grappe	Paire	Type
0	$A_{1-7},\ B_{1-7}$	MM
1	$C_{1-5},\ D_{1-5}$	MM
2	$A_{2-4},\ B_{2-4}$	R
2	$A_{2-4},\ C_{2-4}$	BB
2	$A_{2-4},\ D_{2-4}$	BB
2	$A_{2-4},\ E_{1-3}$	BM
2	$B_{2-4},\ C_{2-4}$	BB
2	$B_{2-4},\ D_{2-4}$	BB
2	$B_{2-4},\ E_{1-3}$	BM
2	$C_{2-4},\ D_{2-4}$	R
2	$C_{2-4},\ E_{1-3}$	BM
2	$D_{2-4},\ E_{1-3}$	BM
3	$A_{4-6},\ B_{4-6}$	R

```
1  (fSet, bSet, cSet) ← classifyClones(
                clusters, clId, BIF)

2     forall cl_k ∈ clusters
3        forall v_i, v_j ∈ cl_k | i < j
4           clType = computeCloneType(BIF, clId, v_i, v_j)
5           switch(clType)
6              case MM
7                 fSet = fSet ∪ (v_i, v_j)
8                 break
9              case BB
10                bSet = bSet ∪ (v_i, v_j)
11                break
12             case BM
13                cSet = cSet ∪ (v_i, v_j)
14                break
15             case MB
16                cSet = cSet ∪ (v_j, v_i)
17                break

18    return fSet, bSet, cSet
```

Figure 2.5 Algorithme de regroupement des clones par catégorie relationnelle

```
1  cloneType ← computeCloneType(
       BIF, clId, v_i, v_j)

2      p_i = BIF.parent(v_i)
3      p_j = BIF.parent(v_j)
4      if p_i ≠ UNDEF
5          if p_j ≠ UNDEF
6              if clId[p_i] ≠ clId[p_j]
7                  return BB
8              else
9                  return R
10         else
11             return BM
12     else
13         if v_j ≠ UNDEF
14             return MB
15         else
16             return MM
```

Figure 2.6 Algorithme de classification des clones

premier fragment est une méthode au lieu d'être un bloc. L'algorithme original conçu pour calculer les différents types est présenté à la figure 2.6. Tout d'abord, les parents p_i et p_j des fragments candidats v_i et v_j sont calculés dans la FIB (ligne 2 et 3). Ensuite, la déduction du type structurelle est effectuée (ligne 4 et 5) à partir de la nature des parents. Si un bloc ne possède aucun parent, on conclut que ce fragment est une méthode. Dans le cas particulier ou les deux fragments sont des méthodes (ligne 14), on retourne le type MM. Si un seul des fragments ne possède aucun parent, alors on retourne soit un type BM (ligne 9), soit un type MB (ligne 12) en fonction de l'ordre des fragments dans la paire. Lorsque les deux fragments sont des blocs, c'est-à-dire qu'ils ne sont pas la racine d'un arbre dans la FIB, une condition supplémentaire (ligne 6) vérifie si les parents sont eux-mêmes des clones. Le type BB est retourné seulement si les parents ne sont pas des clones étant donné que, dans le cas contraire, on identifierait une paire structurellement redondante. La notion de redondance n'est pas abordée dans ce mémoire, mais une discussion en profondeur de ce problème est disponible dans [35].

L'algorithme $SSCT$ de la figure 2.6 contient une brève séquence d'énoncés conditionnels et d'énoncés simples ; il ne contient aucune boucle. Sa complexité est clairement $\mathcal{O}(1)$. L'algorithme SCT à figure 2.5 exécute l'instruction $switch$ des lignes 5 à 17 en complexité $\mathcal{O}(1)$. Ce branchement conditionnel est exécuté pour toutes les grappes $cl_k \in clusters$ (ligne 2) et pour toutes les pairs $(v_i,\ v_j) \in\ cl_k$ (ligne 3). La complexité au pire cas de l'algorithme se produit lorsque tous les

fragments sont similaires entre eux et appartiennent à la même grappe. Dans ce cas, l'algorithme SCT est en $\mathcal{O}(n^2)$ où n est le nombre de fragments analysés dans le système. Ce cas est hautement improbable, car il implique que tous les fragments d'un système soient clones à tous les fragments, une absurdité en soi. Cependant, l'algorithme contient dans tous les cas une composante quadratique sur la taille des grappes. Il est nécessaire de calculer le type structurel des relations de clonage pour chaque pair puisqu'il a été démontré à la section 2.2 que deux pairs appartenant à la même grappe peuvent être d'un type structurel différent (par exemple, un bloc peut être à la fois dans une relation de type BB et une autre de type BM).

2.4 Expériences et résultats

Des expériences ont été menées sur deux systèmes Java en code source ouvert : Tomcat et Eclipse. Elles ont toutes été exécutées sur un processeur Intel Core 2 Duo 3.0 GHz avec 3 GB de mémoire vive opérant sur le système d'exploitation Linux Fedora 8. Le logiciel de calcul des types structurels a été compilé avec g++ 4.1.2. Tomcat ([36]) est une implémentation du Servlet Java et de la technologie Java Server Pages largement répandue pour mettre en service différents systèmes web. Eclipse ([37]) est un vaste IDE pour le développement d'applications dans différents langages. La taille, le nombre de méthodes, le nombre de blocs et le niveau moyen d'imbrication des blocs sont présentés à la figure 4.4 pour les deux systèmes. Toutes les données de cette table ont été produites en considérant les blocs et les méthodes de 7 LOC et plus. Les méthodes ont un niveau d'imbrication de 0. Le niveau d'imbrication des blocs commence à 1 et augmente de 1 à chaque nouveau niveau d'imbrication.

Tableau 2.2 Caractéristiques des systèmes

Systemème	Tomcat	Eclipse
Version	5.5	3.3
LOC	130K	1.3M
Nombre de méthodes	5047	60326
Nombre de blocs	5538	32113
Imbrication moyenne des FIB	0.89	0.62

L'analyse syntaxique des systèmes analysés a été faite avec Eclipse. Elle consiste en l'extraction des FIB et des métriques des blocs. Comme mentionné précédemment, seuls les blocs de 7 LOC et plus ont été sélectionnés. Ce choix provient de la littérature des clones et de l'expérience de l'auteur et est généralement admis comme raisonnable. Il est, toutefois, possible d'utilise un autre seuil si désiré.

La similitude entre les fragments a été calculée avec le logiciel $CLAN$. On peut trouver un résumé de la méthode dans [1]. Les métriques suivantes ont été utilisées pour tous les blocs dans les systèmes choisis : le nombre d'énoncés, le nombre de branches (IF,CASE,etc.), le nombre de boucles (FOR,WHILE,DO,etc.), le nombre d'appels à fonction ou à méthode, le nombre de paramètres (0 pour les blocs imbriqués, possiblement non nul pour les méthodes). Les expériences présentées ici ont été réalisées pour différentes valeurs d'un paramètre $i \in \mathbb{N}$ associé à un tuple de seuils $[i,\ i,\ ...\ ,\ i,\ i]$ avec une valeur initiale $i = 0$. Les différentes étapes de l'expérimentation correspondent toutes à une valeur différente du paramètre i. La première étape est le calcul de toutes les grappes de fragments pour lesquels les métriques sont identiques. Les étapes subséquentes calculent les grappes de fragments pour un tuple uniforme de seuils associé au paramètre non nul i dont la valeur correspond à $n - 1$, où n est le numéro de l'étape courante. Par exemple, à la 3e étape, toutes les composantes du tuple auront une valeur de 2.

Tableau 2.3 Nombre de fragments appartenant à chaque type de relation

System	MM	BB	BM
Tomcat	3,859	10,413	1,991
Eclipse	324,656	190,464	175,607

Le tableau 2.4 donne la cardinalité des ensembles de clones de chaque système pour chacun des types structurels. Les chiffres sont relativement élevés, mais il faut considérer que le nombre de pairs est influencé de manière quadratique par le nombre de fragments dans chacun des systèmes. Néanmoins, les chiffres pour les trois types de relation sont intéressants et peuvent indiquer des opportunités de factorisation du code (voir [7, 38]).

Tableau 2.4 Statistiques sur la taille des blocs

	Tomcat (LOC)		Eclipse (LOC)	
	Moyenne	Maximum	Moyenne	Maximum
MM	9.82	208	9.27	723
BB	7.97	56	8.08	600
BM	7.80	32	7.67	36

Le tableau 2.4 contient différentes statistiques sur les clones déclarés dans les deux systèmes. La taille est mesurée en lignes de code (LOC) telles que rapportées par la commande Linux wc.

Les figures 2.7 et 2.8 montrent la distribution de la taille des blocs appartenant à une relation de type BM pour les deux systèmes. Seules les distributions pour un seuil de 0 sont incluses, car le

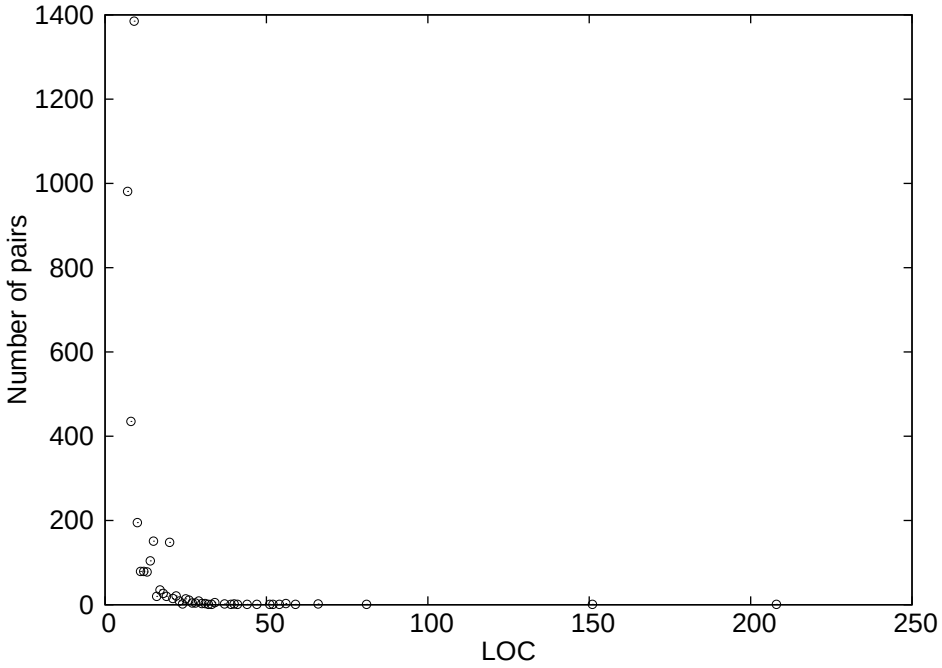


Figure 2.7 Distribution de la taille des fragments de Tomcat appartenant à une relation BF

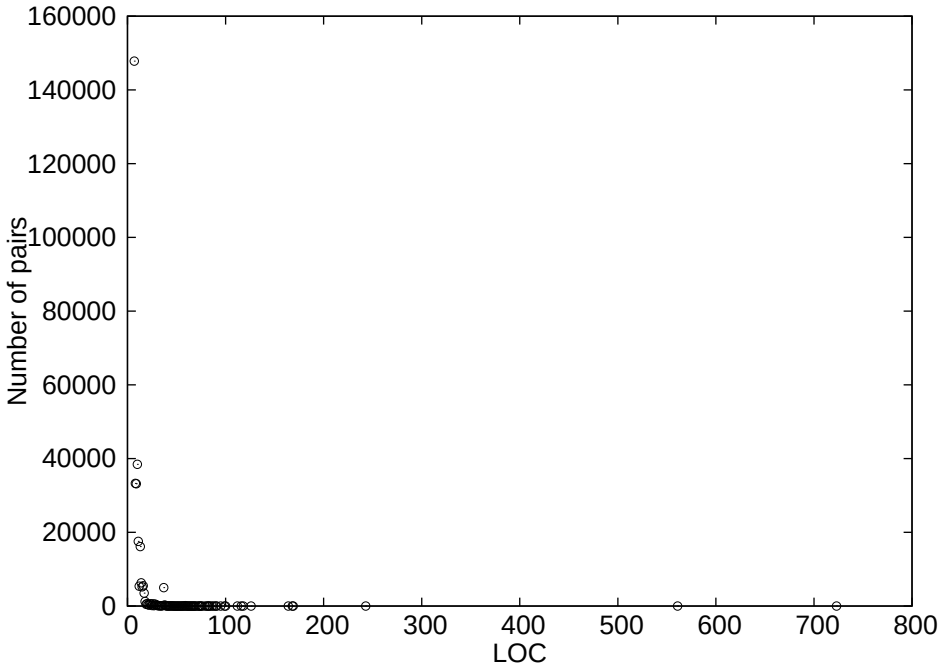


Figure 2.8 Distribution de la taille des fragments d'Eclipse appartenant à une relation BF

volume de données pour chaque seuil est trop élevé. De plus, la significativité des données diminue rapidement à chaque seuil d'un point de vue analytique et de génie logiciel.

Même si les diagrammes dans les figures présentées montrent des données très proches de l'axe des x pour des tailles élevées, les courbes ne semblent pas complètement lisses et des discontinuités semblent être présentes. La raison de la présence de pointes de fréquences à certaines tailles n'a pas été étudiée dans le cadre de cette étude, mais une telle étude pourrait s'avérer intéressante.

```
320 void release(){
322   if(this.filter!=null)
323   {
324      if(Globals.IS_SECURITY_ENABLED){
325        try{
326          SecurityUtil.doAsPrivilege("destroy",filter);
327        }catch(java.lang.Exception ex){
328          context.getLogger().error("ApplicationFilterConfig.doAsPrivilege",ex);
329        }
330        SecurityUtil.remove(filter);
331      }else{
332        filter.destroy();
333      }
334      if (!context.getIgnoreAnnotations()){
335        try{
336          ((StandardContext)context).getAnnotationProcessor().preDestroy(this.filter);
337        }catch(Exception e){
338          context.getLogger().error("ApplicationFilterConfig.preDestroy",e);
339        }
340      }
341   }
342   this . filter = null ;
344 }
```

Figure 2.9 Premier fragment d'une paire de clones Tomcat appartenant à une relation de type BF

Différents exemples de clones trouvés et classés par la méthode sont présentés aux figures 2.9, 2.10 et 2.12.

Dans la figure 2.9, le corps de la méthode $release$ aux ligne 320 à 344 dans le fichier $ApplicationFilterConfig$ dans le répertoire $catalina/core$ de Tomcat a été identifié et est un clone de type-1 avec un bloc des lignes 370 à 394 dans la méthode de la figure 2.10 $setFilterDef$ provenant du même fichier.

Si nécessaire, la méthode $setFilterDef$ pourrait être modifiée tel que proposé à la figure 2.11 pour éviter une duplication du code. Il s'agit d'un exemple de re-factorisation du code découvert par le schéma de classification proposé.

Table 2.4 reports execution times for Tomcat and Eclipse.

Tableau 2.5 Temps d'exécution de la détection et de la classification

Système	Temps (s)
Tomcat	0.303
Eclipse	2.257

```
364 void setFilterDef(FilterDef filterDef)
365   throws ClassCastException, ClassNotFoundException,
366     IllegalAccessException, InstantiationException,
367     ServletException, InvocationTargetException, NamingException {
368
369   this.filterDef = filterDef;
370   if (filterDef == null) {
371
372     // Release any previously allocated filter instance
373     if (this.filter != null){
374       if( Globals.IS_SECURITY_ENABLED) {
375         try{
376           SecurityUtil.doAsPrivilege("destroy", filter);
377         }catch(java.lang.Exception ex){
378           context.getLogger().error("ApplicationFilterConfig.doAsPr
378             ivilege", ex);
379         }
380         SecurityUtil.remove(filter);
381       } else {
382         filter.destroy();
383       }
384       if (!context.getIgnoreAnnotations()) {
385         try {
386           ((StandardContext)context).getAnnotationProcessor().preDestroy(this.filter);
387         }catch (Exception e) {
388           context.getLogger().error("ApplicationFilterConfig.preDestroy", e);
389         }
390       }
391     }
392     this.filter = null;
393
394   }else {
395
396   // Allocate a new filter instance
397   Filter filter = getFilter();
398
399   }
400
401 }
```

Figure 2.10 Second fragment d'une paire de clones Tomcat appartenant à une relation de type BF

```
void setFilterDef(FilterDef filterDef)
   throws ClassCastException, ClassNotFoundException,
     IllegalAccessException, InstantiationException,
     ServletException, InvocationTargetException, NamingException {

   this.filterDef = filterDef;
   if (filterDef == null) {
     this.release();
   }

   } else {
     // Allocate a new filter instance
     Filter filter = getFilter();

   }

}
```

Figure 2.11 Proposition de factorisation du code pour la paire présentée de type BF

```
295 public void nextRequest(){
298   request.recycle();
301   if(lastValid-pos>0){
302     int npos = 0;
303     int opos = pos;
304     while(lastValid-opos>opos-npos){
305       System.arraycopy(buf,opos,buf,npos,opos-npos);
306       npos+=pos;
307       opos+=pos;
308     }
309     System.arraycopy(buf,opos,buf,npos,lastValid-opos);
310   }
313   for(int i=0;i<=lastActiveFilter;i++) {
314     activeFilters[i].recycle();
315   }
318   lastValid=lastValid-pos;
319   pos=0;
320   lastActiveFilter=-1;
321   parsingHeader = true;
322   swallowInput = true;
324 }
```

Figure 2.12 Fragment de Tomcat appartenant à une relation de type MM

2.5 Discussions

Les résultats suggèrent que la classification des relations structurelles de clonage peut être évaluée avec l'algorithme SCT sur un ordinateur de bureau ordinaire.

L'approche proposée a identifié 3.859 MM, 10.413 BB, et 1.991 BM pairs de clones dans Tomcat et 324.656 MM, 190.464 BB, et 175.607 BM dans Eclipse. La manipulation de résultats de cette taille nécessite une forme de traitement automatisée, comme dans [7, 38] ; il est inconcevable de devoir valider l'ensemble de ces résultats manuellement. Il faut souligner que la grande quantité de résultats obtenus ici confirme l'influence de la composante quadratique du calcul tel que prédit à la section 2.3. Néanmoins, il est clair qu'il existe des relations de clonage autre que la relation BB. On ne peut donc accepter la fonction comme unité syntaxique de base sans au préalable rejeter la pertinence des clones formés au niveau des blocs.

Les résultats présentés sont également influencés par plusieurs facteurs expérimentaux.

Tous les systèmes analysés sont en Java. De plus, seulement deux systèmes ont été analysés avec des configurations d'imbrication et de clonage qui leur sont propres. Il est donc difficile de généraliser les résultats. Les résultats pourraient différer sur des systèmes écrits dans d'autres langages orientés-objets ou d'autres paradigmes. D'autres systèmes en Java pourraient également être analysés avec des caractéristiques d'imbrication différentes. Néanmoins, la nature très différente des systèmes et leur taille considérable confèrent une certaine qualité aux résultats.

2.6 Conclusions

Dans cette section, il a été question de l'existence de différents types structurels de relations de clonage et de leur classification en fonction de leur position dans une FIB. Trois types de relations ont été considérées : (MM), (BB), (BM) et la relation symétrique (MB).

Les relations présentées sont intéressantes parce qu'elles peuvent offrir des opportunités de factoriser le code : les relations BB peuvent être des candidats à l'encapsulation dans une méthode, les relations BM ou MB peuvent indiquer un remplacement direct par un appel à méthode et les relations MM peuvent suggérer une fusion de l'interface de deux classes.

Le schéma de classification proposé a été évalué sur deux systèmes Java : Tomcat et Eclipse. Les résultats suggèrent que les quatre types de relations sont présentes dans les systèmes et qu'on ne peut pas se restreindre à la seule analyse des méthodes pour identifier tous les clones d'un système.

L'élargissement de la base expérimentale et la diversification des langages analysés pourraient contribuer à améliorer la qualité des résultats. Le déploiement de méthodes automatisées ou semi-automatisées de factorisation du code pourrait vérifier l'utilité pratique des résultats.

CHAPITRE 3

FILTRAGE DE CLONES SUR GPU

3.1 Introduction au calcul des clones sur GPU

L'utilisation de larges grappes de CPU est traditionnellement réservée au calcul parallèle intensif. À la fois difficile à construire et à entretenir, les grappes de calcul sont utilisées par des groupes spécialisés avec des besoins précis. L'arrivée sur le marché de nouvelles générations de GPU, architecture parallèle spécialisée dans le rendu graphique, dans les ordinateurs de bureau a introduit une nouvelle façon d'exécuter des calculs massivement parallèle à faible coût. Cependant, en comparaison des CPU, les GPU ont une architecture restrictive et un modèle de flot de données différent. Par conséquent, la résolution d'un problème sur GPU requiert une approche différente. Cette section est consacrée à l'implémentation de la variante Smith-Watterman de l'algorithme de Wagner-Fisher [39, 40] pour le filtrage des clones en fonction de leur distance de Levenshtein. Smith-Watterman et Wagner-Fisher sont deux algorithmes de programmation dynamique ; ils se parallèlisent donc facilement. Une brève discussion des autres applications possibles des GPUs pour la résolution d'autres problèmes de détection des clones est également présentée.

Cette section est divisée comme suit : la section 3.2 présente quelques travaux pertinents sur les GPU, la section 3.3 compare les architectures des CPU et ds GPU, la section section 3.4 décrit des applications des GPU aux problèmes de clonage, la section 3.5 décrit l'algorithme de filtrage des clones sur CPU et GPU, les sections 3.6 décrivent l'environnement expérimental et les résultats de l'application de l'algorithme de filtrage, la section 3.7 discute des résultats et des limitations et la section 3.8 conclut le chapitre.

3.2 Revue de la littérature

Il a été démontré dans [41] que l'architecture GPU peut résoudre des problèmes dont la nature n'est pas graphique si on peut établir la correspondance entre le domaine du problème et le domaine graphique. L'algorithme de Floyd-Warhsall a été implémenté avec succès dans [42] pour la prédiction des structures protéiniques. Des algorithmes de tris avec de meilleures performances que ceux sur CPU ont été implémentés dans [43]. De ces exemples, on conclut que certains algorithmes peuvent avoir une efficacité accrue avec une implémentation sur GPU.

En ce qui a trait aux problèmes de comparaison de sous-chaînes, une solution utilisant les arbres de suffixes sur GPU a été implémentée dans [44]. Récemment, la comparaison approximative de

chaînes pour la détection de la signature de virus a été implémentée dans [45] en utilisant une méthode basée sur le fenêtrage. Cependant, les méthodes de fenêtrage sont susceptibles de rapporter un nombre élevé de faux-positifs et, si une précision élevée est requise, une méthode comme l'identification de la plus longue sous-séquence commune (LCS) est recommandée. L'algorithme classique pour calculer la LCS est l'algorithme de Wagner-Fisher, un modèle de programmation dynamique utilisée dans plusieurs livres pédagogiques [40]. L'implémentation de l'algorithme de [40] est parfois nommé *DP-matching*. En bio-informatique, elle est appelée Smith-Watterman. Manavski a implémenté Smith-Watterman sur GPU dans [39]. Une variation appelée Needleman-Wunsch a également été implémentée sur GPU dans [46].

Ce chapitre introduit une nouvelle façon d'organiser les données sur le GPU afin de calculer la LCS de plusieurs chaînes de caractères en même temps en appliquant plusieurs idées de [39]. L'approche proposée est cependant supérieure, car elle permet de traiter plusieurs chaînes au lieu d'une seule. L'utilité du calcul de la LCS dans la détection du clonage est expliquée dans 3.4. La section suivante compare les architectures CPU et GPU.

3.3 Architecture

Les architectures CPU usuelles sont associées aux modèles de traitement SISD (*Single Instruction, Single Data*, une instruction, une donnée). Une seule unité de calcul (par exemple ALU, FPU, SSE, etc.) peut être active à un moment donné et elle exécute une seule instruction à la fois (la technologie d'Intel Hyper Threading introduit de légères variations dans ce modèle, mais elles ne sont pas significatives dans le cadre de cette discussion. Le lecteur peut se référer à [47] pour plus d'information sur le Hyper Threading). Les circuits de calcul sont connectées à une cache unique connectée à la mémoire principale du système tel qu'illustré sur la figure 3.1. Cependant, avec l'arrivée des CPU à plusieurs coeurs, il est maintenant possible d'avoir plusieurs fils d'exécution sur un seul CPU, un sur chaque coeur. Ces fils peuvent exécuter des calculs indépendants les uns des autres en partageant à la fois la mémoire cache et la mémoire principale. La figure 3.3a illustre une architecture à deux coeurs typique avec une cache et une mémoire principale partagée. Les CPU à plusieurs coeurs peuvent exécuter différentes instructions sur différentes données en même temps ; par conséquent, ils appartiennent aux architectures de type MIMD (*Multiple Instruction, Multiple Data*, plusieurs données, plusieurs instructions).

Le modèle architectural des GPU partage quelques caractéristiques avec les CPU à plusieurs coeurs : les deux possèdent plusieurs coeurs de calcul et ils partagent la mémoire cache et la mémoire principale entre leurs processeurs. Le nombre de processeurs sur les GPU est cependant beaucoup plus élevé que sur les CPU : plusieurs centaines contre moins d'une dizaine. De plus, contrairement aux CPU à plusieurs coeurs, les coeurs d'un GPU sont limités à exécuter la

même instruction lors d'un cycle, ce qui ajoute une contrainte. Par conséquent, le modèle architectural des GPU correspond au type SIMD (*Single Instruction, Multiple Data*, une instruction, plusieurs données). Les architectures récentes des GPU ont éliminé cette contrainte en permettant aux programmeurs de définir plusieurs programmes à exécuter simultanément sur le GPU. La figure 3.3b montre un GPU moderne avec deux blocs logiques d'ALU. Chaque bloc effectue des calculs différents. Le regroupement logique des ALU est effectué par le pilote du GPU si son architecture le permet. On peut donc voir les GPU moderne comme un regroupement d'unité SISD formant une plus grosse architecture MIMD.

Malgré cette amélioration récente de l'architecture GPU, il existe toujours un certain nombre de contraintes qui doivent être prises en considération :

- les langages de programmation orientés vers les GPU n'autorisent pas les fonctions récursives et les pointeurs de fonctions ;
- les instructions de branchement peuvent diminuer les performances, car elles interrompent le pipeline d'exécution ;
- la bande passante du bus et la latence entre le GPU et le CPU forment un goulot d'étranglement ;
- la majorité des plateformes de développement se conforment chacune à une seule famille de GPU et donc ne produisent pas du code portable.

Les plateformes CUDA de Nvidia [48], CTM de AMD [49] et le standard ouvert OpenCL [50] sont les plateformes de développement principales pour GPU. Sans ces outils, le programmeur doit contourner le pipeline de rendu graphique imposé par les différents API comme OpenGL et DirectX. Néanmoins, avec l'utilisation d'un langage de nuanceurs de haut niveau et une manipulation adéquate de la mémoire de textures pour les entrées et les sorties, il est tout à fait possible d'utiliser un GPU pour exécuter des calculs arbitraires.

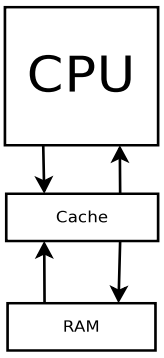
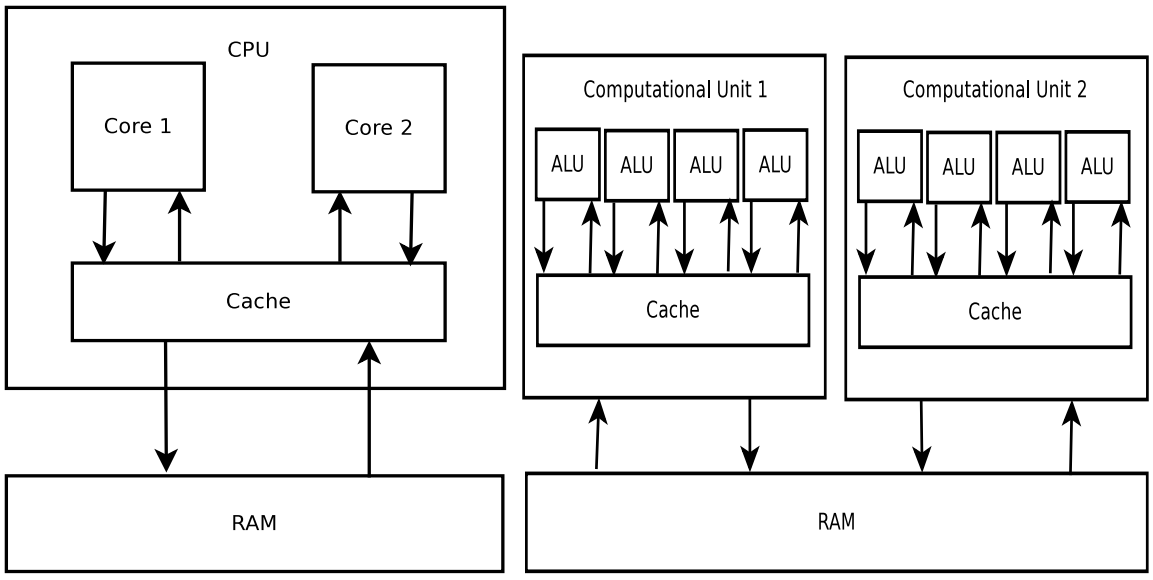


Figure 3.1 Schéma de l'architecture standard d'un CPU



(a) Multicore CPU Architecture (b) Modern GPU Architecture

Figure 3.2 Schéma de différents types d'architectures multicoeurs

3.4 Détection des clones

Dans cette section, l'idée développée est la suivante : choisir une méthode de détection de clonage et identifier une de ces étapes pouvant bénificier d'un calcul sur GPU. La méthode de détection de clonage d'intérêt ici est la technique basée sur les vecteurs de métriques. Cette technique a été utilisée plusieurs fois dans la littérature, entre autres dans [51, 52], pour analyser le clonage des méthodes. L'analyse des unités syntaxiques tels que les fonctions et les méthodes n'est pas l'apanage de la méthode par vecteurs de métriques et le lecteur pourra en trouver d'autres exemples dans [53]. En suivant l'approche proposée par les auteurs de [51], les blocs syntaxiques sont d'abord extraits de l'AST et sont ensuite comparés en utilisant différentes métriques du logiciel représentatives de la taille, du flot de contrôle, du couplage et la communication entre les unités syntaxiques.

Dans le cadre de cette expérience, seuls les blocs d'énoncés et les méthodes sont considérés comme des fragments valides. À partir de l'AST, les limites structurelles des fragments choisis sont facilement identifiables. À titre comparatif, les méthodes basées sur les jetons doivent effectuer un post-traitement sur les résultats bruts afin d'obtenir des résultats associés aux unités syntaxiques. Par conséquent, le pré-traitement associatif de la méthode basée sur les vecteurs de métriques n'engendre pas de coût computationnel significatif par rapport à d'autres méthodes textuelles.

L'approche basée sur les métriques d'AST requiert les étapes suivantes afin de produire les grappes de similitude :

1. Identification des fragments.

2. Extraction des métriques.
3. Extraction des jetons associés aux fragments.
4. Construction et identification des grappes.
5. Filtrage (LCS ou autre)

Avant l'application de la première étape, l'analyse syntaxique du système analysé est réputée complète et les AST de ses fichiers disponibles. L'identification des fragments est l'étape à laquelle chaque fragment est associé à un unique nombre naturel, son identifiant. Cette association est faite de manière incrémentielle en parcourant les AST. Chaque nouveau fragment détecté lors du parcours des AST engendre la création d'une entrée dans un conteneur associatif. Cette entrée associe les informations positionnelles du nouveau fragment, i.e., le nom du fichier et les lignes de début et fin du fragment, et son identifiant unique. Pour chacun des fragments identifiés à cette étape, un parcours de l'AST est effectué pour collecter les métriques requises. Dans les expériences présentées ici, les métriques retenues sont le nombre de lignes de code, le nombre d'énoncés, la complexité cyclomatique, le nombre de branches, le nombre de boucles, le nombre de paramètres reçus, le nombre de variables locales et le nombre d'appels à méthodes. Même si elles sont identifiées comme deux étapes séparées, l'identification des fragments et l'extraction des métriques peuvent être réalisées lors du même parcours des AST.

La construction des grappes se fait ensuite sur la base des métriques extraites. La technique exacte utilisée est la création de grappes basée sur la quantification de l'espace telle que présentée dans [51]. L'égalité entre les métriques est requise pour que deux fragments appartiennent à la même grappe. Les grappes ainsi créées contiennent les clones candidats. On dit candidats, car certains clones seront rejetés à l'étape du filtrage.

Les grappes formées à partir des vecteurs de métriques contiennent de faux positifs, car deux fragments ayant des métriques identiques ne sont pas nécessairement syntaxiquement similaires. Dans les faits, deux fragments radicalement différents peuvent produire des vecteurs de métriques identiques. Par conséquent, il est nécessaire de filtrer les résultats après la construction des grappes. L'idée retenue pour cette étape est l'alignement lexical des séquences de jetons des candidats en utilisant le calcul de la plus longue sous-séquence commune (LCS). Si cet alignement était fait sur tout un système, le temps requis serait trop élevé étant donné qu'il est à la fois quadratique sur la taille et le nombre de fragments. En utilisant d'abord la mise en grappe, le temps requis demeure quadratique, mais seulement sur la cardinalité des grappes et la taille des fragments. De plus, étant donné que les fragments d'une grappe ont tendance à se ressembler, il est proposé ici de faire l'alignement sur un seul fragment de référence, choisi de manière arbitraire comme le premier de la grappe, dans une grappe. Cela conduit à une complexité linéaire sur la cardinalité des grappes au prix de la précision des résultats. L'alignement désiré est calculé sur la représentation lexicale

abstraire du code, les jetons, et non le texte original. Pour cette raison, il est nécessaire d'extraire les jetons à l'étape précédente.

Cette dernière étape, l'alignement basé sur la LCS, est intrinsèquement parallélisable. Il s'agit donc d'un excellent candidat au calcul sur GPU. La prochaine section explique comment exploiter cette opportunité pour améliorer les performances du filtrage des grappes.

3.5 Algorithme

Les éléments discutés précédemment justifient l'utilisation du GPU pour filtrer les grappes de clones. Cette section présente l'implémentation sur GPU de l'algorithme calculant la LCS, mais aux fins de rappel, la version classique est d'abord présentée.

Soit s_1 et s_2 deux chaînes de caractères. Leur longueur est respectivement notée $len(s_1)$ et $len(s_2)$. Soit D une matrice de taille $(len(s_1)+1)$ x $(len(s_2)+1)$ indexée par des entiers naturels i et j. Enfin, soit $d_{i,j}$ une cellule de la matrice D indexée par i et j. L'algorithme de la figure 3.3 calcule la plus longue sous-séquence commune entre les chaînes de caractères s_1 et s_2. Il s'agit d'un exemple classique d'un algorithme de programmation dynamique (Dynamic Programming ou DP en anglais) ; tous les détails sur cet algorithme se trouve dans [40].

```
1  subseqLength ← DPMatching(s1, s2)
2      i = 0
3      j = 0
4      while i <= len(s1)
5          D[i][0] = i
6          if i <= len(s2)
7              D[0][i] = i
8          i = i + 1
9      i = j = 1
10     while i <= len(s1)
11         while j <= len(s2)
12             if s1[i] ≠ s2[j]
13                 δ = 0
14             else
15                 δ = 1
16             D[i][j] = max(D[i − 1][j − 1] + δ, D[i − 1][j],
                   D[i][j − 1])
17             j = j + 1
18         i = i + 1
19     return D[len(s1)][len(s2)]
```

Figure 3.3 Algorithme du calcul de la LCS sur CPU

Connaissant les contraintes des GPU (voir la section 3.3 sur l'architecture), il est d'abord

nécessaire d'identifier les cellules de la matrice qui n'ont aucune dépendance computationnelle. La valeur de ces cellules peut alors être calculée simultanément sur différents coeurs. Une cellule est libre de dépendances computationnelles si à une étape donnée du calcul sa valeur ne dépend que de valeurs déjà calculées à une étape précédente. Une vérification rapide permet de conclure que les valeurs sur les antidiagonales de la matrice sont libres de dépendances au même moment puisque leur valeur ne dépend que des valeurs des deux antidiagonales précédentes. Aux fins de clareté, définissons l'étape k comme le calcul de toutes les valeurs de la k^e antidiagonale.

Il est clair que les tampons de computation essentiels sont les antidiagonales. Pour simplifier le calcul, les antidiagonales sont projetées dans un tampon linéaire, c'est-à-dire un tableau. Ces tampons linéaires sont requis pour optimiser l'utilisation de l'espace mémoire et le nombre de transferts entre le CPU et le GPU ; la consommation de mémoire est maintenant linéaire au lieu d'être quadratique. Toutefois, il est nécessaire d'établir une correspondance significative entre les index originaux de la matrice et les index du tableau. Soit V_k le tampon linéaire de l'étape k et f_k la fonction de correspondance entre les indexes de la matrices et les indexes du tableau, alors :

$$\begin{aligned} f_k : \mathbb{N} &\rightarrow \mathbb{N} \times \mathbb{N} \\ f_k(u) = (u, k-u) &\quad if\ k <= len(s_1) \\ f_k(u) = (u+k-len(s_1), len(s_2)-u) &\quad if\ k > len(s_1) \end{aligned} \tag{3.1}$$

où i et j sont les indexes de la cellule originale de la matrice D, u est l'index de la cellule correspondante dans V_k et k l'étape courante. Le tampon V_k est la représentation désirée de l'antidiagonale correspondante de la matrice originale.

La correspondance étant maintenant établie entre la matrice conceptuelle et le tampon implémenté en pratique, on peut maintenant définir la valeur de chaque cellule. Soit $d_{i,j}$ une cellule arbitraire de la matrice D. Alors, sa valeur est définie par :

$$\begin{aligned} d_{i,j} &= 0 \quad if\ i = 0\ or\ j = 0 \\ d_{i,j} &= max(d_{i-1,j-1} + \delta, d_{i-1,j}, d_{i,j-1}) \end{aligned} \tag{3.2}$$

où δ est défini par :

$$\begin{aligned} \delta &= 1 \quad if\ s_1[i] = s_2[j] \\ \delta &= 0 \quad if\ s_1[i] <> s_2[j] \end{aligned} \tag{3.3}$$

Suivant la transformation proposée sur la matrice, soit $V_{k,u}$ une cellule d'un V_k. Alors, la valeur

de $V_{k,u}$ est $d_{f_k(u)}$ et δ est maintenant défini par :

$$\begin{aligned} if\ k < len(s_1) : \\ \delta = 1\ if\ s_1[u] = s_2[k-u] \\ \delta = 0\ if\ s_1[u] <> s_2[k-u] \end{aligned} \tag{3.4}$$

$$\begin{aligned} if\ k\ >= len(s_1) : \\ \delta = 1\ if\ s_1[u+k-len(s_1)] = s_2[len(s_2)-u-1] \\ \delta = 0\ if\ s_1[u+k-len(s_1)] <> s_2[len(s_2)-u-1] \end{aligned}$$

Il devrait maintenant être clair qu'à l'étape k, toutes les valeurs des cellules de V_k peuvent être calculées sans aucune dépendance entre elles. L'algorithme du calcul de la LCS sur GPU peut donc être séparé en deux étapes : la première sur CPU est une boucle de commande qui indique au GPU l'itération et les tampons courants et la deuxième le calcul de la valeur de chaque cellule sur les coeurs du GPU.

Le pseudo-code du pilote CPU est détaillé à la figure 3.4. Ce code appelle la procédure du GPU à chaque itération. Le pilote reçoit deux paramètres : les chaînes s_1 et s_2. À la ligne 3, on voit le nombre d'itérations borné par la longueur des deux chaînes alignées. Ce nombre est extrait de la structure de la matrice originale D. À la ligne 4, un appel à la procédure sur GPU est exécuté et implicitement calcule la valeur de toutes les cellules du tampon courant. La fonction est appelée avec quatre paramètres même si la procédure GPU en prend cinq ; ce cinquième paramètre est passé à la procédure par l'environnement d'exécution du code et l'API (GLSL est le langage utilisé ici). La longueur de la plus longue sous-séquence se trouve en temps normal dans la cellule en bas à droite de la matrice de calcul. Cette cellule correspond à la dernière antidiagonale de la matrice et cette antidiagonale possède trivialement une seule cellule. La valeur recherchée est donc l'unique valeur du dernier tampon linéaire calculé et est récupérée puis retournée aux lignes 6 et 7.

Tel que mentionné à la section 3.3, une unité de calcul physique, un coeur, est associé et dédié au calcul de la valeur de chaque cellule. La figure 3.5 montre le pseudo-code du calcul de chaque coeur. Les paramètres s_1 et s_2 sont les chaînes à aligner, b est le tableau contenant les trois tampons des antidiagonales nécessaires au calcul, $bIndex$ l'index du tampon dans b contenant l'antidiagonale courante, k l'index du calcul courant et u est l'index de la cellule dont la valeur est calculée. C'est ce dernier paramètre qui est automatiquement calculé par l'environnement. Le paramètre $bIndex$ est la clef derrière l'utilisation des tampons linéaires au lieu de la forme matricielle. Cette valeur permet de cycler à travers les tampons et de réutiliser les tampons dont les valeurs ne sont plus requises aux calculs future. Il a été montré plus haut que la valeur d'une cellule n'est dépendante que des valeurs

```
1  subseqLength ← DPMatching(s1, s2)
2      k = 0
3      while k ≠ len(s1) + len(s2)
4          computeDPIteration(s1, s2, buf, k)
5          k = k + 1
6      subseqLength = buf[k%3][0]
7      return subseqLength
```

Figure 3.4 Pilote CPU de l'algorithme de LCS sur GPU

de cellules contenues sur les deux antidiagonales précédentes. Comme ces antidiagonales ont été linéarisées, les valeurs deviennent dépendantes de valeurs des deux tampons linéaires précédents. L'utilisation de l'arithmétique modulaire permet de calculer l'index des deux tampons précédents à partir de l'index du calcul courant.

Les lignes 8 et 9 et les lignes 19 et 20 récupèrent les caractères des chaînes nécessaires au calcul du δ (voir équation 3.4). Les groupes de lignes 11 à 13, 15 à 17 et 21 à 23 récupèrent les trois valeurs précédentes requises au calcul de $V_{k,u}$. Les lignes 25 à 28 calculent la valeur δ et la ligne 30 calcule la valeur de $V_{k,u}$.

Le temps requis en pire cas par l'algorithme est quadratique sur la taille des chaînes alignées. Dans le pire cas général, l'implémentation sur GPU exhibe un temps quadratique en pire cas. Toutefois, le grand nombre de coeurs performant le calcul de la valeur des cellules peut diviser la constante de quadraticité par un très grand facteur. En théorie, ce facteur et égal au nombre de coeurs sur le GPU (environ 500 sur les cartes disponibles en 2010). Mentionnons également que si le nombre de coeurs sur le GPU est supérieur à la taille de la plus grande des deux chaînes, l'algorithme s'exécute en un temps linéaire sur la taille des chaînes.

Sur de très petites chaînes de caractères, le CPU est plus rapide que le GPU, en grande partie à cause du faible taux de transfert entre les mémoires. Cependant, la performance dans ce cas peut être améliorée si la chaîne doit être comparée à un ensemble de chaînes. Dans ce cas, il est possible d'organiser les données en mémoire de manière à obtenir le résultat de plusieurs alignements dans le temps normalement requis pour en avoir un seul. Cette technique est décrite ci-après.

Soit C un ensemble de chaînes de caractères, c_0 un élément de C et C' l'ensemble défini par :

$$C' = C - \{c_0\} \tag{3.5}$$

Les alignements désirés correspondent à l'alignement de c_0 avec chacune des c_i de C'. Ces alignements peuvent être calculés dans une matrice qui regroupe les matrices des alignements individuels. Voir figure 3.7 pour l'illustration de ces regroupements de matrices.

```
1  computeDPIteration(s1, s2, b,
       bIndex, k, u)
2      a0 = UNDEF
3      a1 = UNDEF
4      l0 = 0
5      l1 = 0
6      l2 = 0
7      if k <= len(s1) + 3
8          a0 = s1[u]
9          a1 = s2[k − u − 1]
10         if k < len(s1) + 3
11             l0 = b[(bIndex − 1)%3][u − 1]
12             l1 = b[(bIndex − 1)%3][u]
13             l2 = b[(bIndex − 2)%3][u − 1]
14         else
15             l0 = b[(bIndex − 1)%3][u − 1]
16             l1 = b[(bIndex − 1)%3][u]
17             l2 = b[(bIndex − 2)%3][u]
18     else
19         a0 = s1[u + k − len(s1)]
20         a1 = s2[len(s2) − u − 1]
21         l0 = b[(bIndex − 1)%3][u]
22         l1 = b[(bIndex − 1)%3][u + 1]
23         l2 = b[(bIndex − 2)%3][u + 1]
24
25     if a0 ≠ a1
26         δ = 0
27     else
28         δ = 1
29
30         b[bIndex][u] = max(l2 + δ, l0, l1)
```

Figure 3.5 Algorithme du calcul de la LCS sur GPU

	-	B	B	A
-	0	0	0	0
A	0			
B	0			
B	0			

(a) Initial step

	-	B	B	A
-	0	0	0	0
A	0	**0**		
B	0			
B	0			

(b) Step 1

	-	B	B	A
-	0	0	0	0
A	0	0	**0**	
B	0	**1**		
B	0			

(c) Step 2

	-	B	B	A
-	0	0	0	0
A	0	0	0	**1**
B	0	1	**1**	
B	0	**1**		

(d) Step 3

	-	B	B	A
-	0	0	0	0
A	0	0	0	1
B	0	1	1	**1**
B	0	1	**2**	

(e) Step 4

	-	B	B	A
-	0	0	0	0
A	0	0	0	1
B	0	1	1	1
B	0	1	2	**2**

(f) Final step

Figure 3.6 Matrices de la programmation dynamique à chaque étape k du calcul

Bien entendu, l'implémentation doit contourner les contraintes architecturales du GPU. En pratique, cela requiert l'insertion d'octets de remplissage entre les chaînes à aligner. Cela normalise le nombre d'étapes k nécessaires au calcul de toutes les sous-matrices conceptuelles (en pratique, les matrices sont linéarisées selon le même patron présenté plus haut). Les sous-matrices sont clairement identifiées aux figures 3.7 et 3.8 ; elles sont encadrées par une ligne doublée. On remarque la présence de caractères ϵ aux fins de remplissage. La longueur p du remplissage concaténé à chaque c_i est défini par :

$$p = len(c_i) - max(len(c_j)|c_j \in C) \tag{3.6}$$

Les c_i de C' une fois concaténé avec leur remplissage sont concaténés un à la suite de l'autre puis séparé par un caractère vide spécial pour éviter des effets de transbordement du calcul. L'ordre des concaténations n'a aucune importance. Cette chaîne est équivalente à s_1 dans le problème original. Ensuite, c_0 précédé par le caractère vide est concaténé $(|C'| - 1)$ fois à elle-même ; le résultat est équivalent à s_2 dans le problème original. Un exemple des chaînes résultantes est présenté à la figure 3.9. Le remplissage assure que chaque sous-matrice soit carrée ; la matrice complète est donc carré. Le résultat de toutes ces opérations est visible dans les figures 3.7 et 3.8. Ces figures contiennent également l'état des matrices au début et à fin du calcul.

À chaque étape k, toutes les cellules des antidiagonales de toutes les sous-matrices sont évaluées. Tel que mentionné, les antidiagonales sont également linéarisées dans cette version du problème suivant les formules présentées plus haut. La figure 3.10 montre les tampons linéaires correspondants des matrices de la figure 3.7 à l'étape k=3. Les tampons contenant les chaînes sont illustrés à la figure 3.9.

Une observation importante est que la première antidiagonale évaluée n'est pas située à la première cellule de la matrice ; la valeur initiale de k n'est donc pas 0. Cette valeur est en fait $|C' - 1| * (max(len(c_i)|c_i \in C) + 1)$ et le nombre d'étapes k et réduit à $2(max(len(c_i)|c_i \in C) - 1$ étant donné que seules les valeurs des sous-matrices sont requises.

Les valeurs désirées sont situées dans les cellules du bas à droite de chaque sous-matrice. À partir des tampons linéaires introduits précédemment, les indexes γ correspondant aux cellules d'intérêt peuvent être déterminés par l'équation :

$$\gamma = x(max(len(c_i)|c_i \in C) + 1) \; \forall \; x \in \{0..|C'| - 1\} \tag{3.7}$$

Le choix de la plus longue sous-séquence commune dans cette expérience au lieu de la distance de Levenshtein est arbitraire. Toutefois, convertir la technique présentée pour calculer la distance de Levenshtein ne requiert que des modifications au sous-programme qui calcule les valeurs des cellules ; cette conversion se fait aisément. La formule suivante met en lumière une des relations

	ε	B	B	ε		ε	A	ε	ε		ε	B	B	A
ε											0	0	0	0
A											0			
B											0			
A											0			
ε						0	0	0	0					
A						0								
B						0								
A						0								
ε	0	0	0	0										
A	0													
B	0													
A	0													

Figure 3.7 Étape initiale du calcul de la LCS d'une à plusieurs chaînes de caractères. Les différentes valeurs sont : c_0 = "ABA" et C'= {"BB", "A", "BBA"}. Les sous-matrices sont à l'intérieur des lignes doubles.

	ε	B	B	ε		ε	A	ε	ε		ε	B	B	A
ε											0	0	0	0
A											0	0	0	1
B											0	1	1	1
A											0	1	1	**2**
ε						0	0	0	0					
A						0	1	1	1					
B						0	1	1	1					
A						0	1	1	**1**					
ε	0	0	0	0										
A	0	0	0	0										
B	0	1	1	1										
A	0	1	1	**1**										

Figure 3.8 Calcul de la LCS d'un à plusieurs à la dernière étape

s_1	-	B	B	ε		-	A	ε	ε		-	B	B	A
s_2	-	A	B	A		-	A	B	A		-	A	B	A

Figure 3.9 Tampons des chaînes du calcul de la LCS d'un à plusieurs après la concaténation de tous les c_i dans C' (s_1) et de toutes les instances de c_0 à elle-même (s_2)

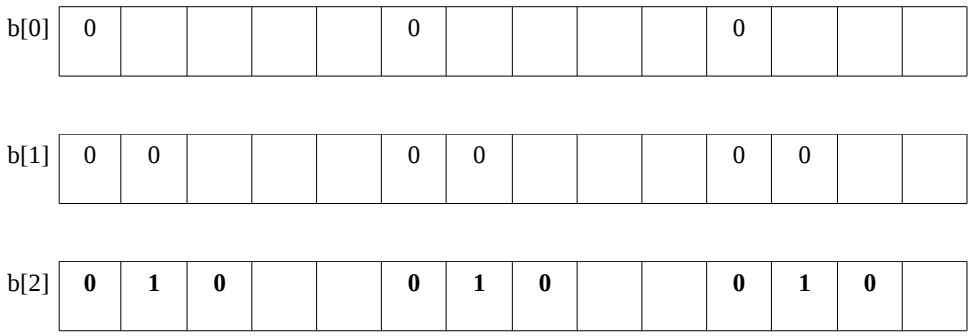


Figure 3.10 Calcul de la LCS d'un à plusieurs dans des tampons linéaires à l'étape k=2

entre la LCS et la distance de Levenshtein :

$$\mu = max(s_1, s_2) - \sigma \tag{3.8}$$

où μ est une borne supérieure de la distance de Levenshtein entre s_1 et s_2 et σ la LCS entre s_1 et s_2.

À la section 3.4, la nature des données de clonage a été expliquée. Le pont entre ces données et les structures proposées dans cette section est maintenant présenté. L'algorithme de conversion est décrit à la figure 3.11. La fonction $clusterDPmapping$ prend deux paramètres : la grappe, $cluster$ qui doit être organisée dans les matrices et le tableau associatif, $tokens$, de chaque fragment et sa représentation sous forme de chaînes de jetons. Étant donné que les jetons sont identifiés par des nombres et non leur image, ces chaînes forment en pratique des chaînes de caractères qu'il ne faut pas confondre avec les chaînes de caractères originaux provenant du code. Les chaînes de jetons sont produites par l'analyse lexicale des chaînes de caractères provenant de la source. Les fragments peuvent avoir des chaînes de jetons de longueur variable et nécessitent donc le remplissage proposé.

Aux lignes 4 et 6 de la figure 3.11, on calcule la valeur p de l'équation 3.6. En accord avec cette équation, la variable $tokens[f_i]$ de l'algorithme de la figure 3.11 est équivalente au c_i de l'équation ; les deux sont des chaînes de caractères. L'union de tous les $tokens[f_i]$ est équivalent à l'ensemble C et $tokens[f_0]$ est équivalent à c_0 ; l'équivalence avec C' s'ensuit. Le remplissage et la construction des tampons des chaînes de caractères sont effectués aux lignes 8 à 20. La valeur de retour $buffers$ est un tableau contenant les deux tampons de chaînes de caractères. En supposant que la grappe et que les jetons ont été extraits, la procédure complète pour calculer la LCS entre le premier fragment de la grappe et tous les autres fragments est décrite à la figure 3.12. Le lecteur la comprendra sans peine.

Une dernière remarque s'impose sur l'implémentation des algorithmes proposés. Étant donné que la méthode retenue est l'implémentation en GLSL, tous les tampons sont vus par le GPU

```
1  buffers ← clusterDPmapping(cluster, tokens)
2      buffers = [ ]
3      maxlen = 0
4      forall f_i ∈ cluster
5          if len(tokens[f_i]) > maxlen
6              maxlen = len(tokens[f_i])
7
8      forall f_i ∈ cluster
9          buffers[0]+ = tokens[f_i]
10         i = 0
11         while i <= len(tokens[f_i]) − maxlen
12             buffers[0]+ =′ 0′
13             i+ = 1
14     i = 0
15     while i <= len(cluster)
16         buffers[1]+ = tokens[f_0]
17         j = 0
18         while j <= len(tokens[f_0]) − maxlen
19             buffers[0]+ =′ 0′
20             j+ = 1
21
22     return buffers
```

Figure 3.11 Algorithme d'association entre les grappes et les tampons de la programmation dynamique

```
1  subseqLength ← computeClusterDP(cluster, tokens)
2      buffers = clusterDPmapping(cluster, tokens)
3      return DPMatching(buffers[0], buffers[1])
```

Figure 3.12 Calcul de la LCS d'une grappe

comme des textures (1D ou 2D au besoin) dont la largeur maximale est en pratique souvent bornée à 8192. Cette contrainte impose de travailler avec des séquences d'au plus 8192 caractères. En pratique, étant donné la longueur moyenne des clones ce n'est pas un problème (voir la section 3.6).

3.6 Expériences et résultats

Les résultats présentés ici sont différentes mesures de l'étape du filtrage ainsi que d'autres mesures intéressantes afin d'interpréter les résultats d'intérêts.

Le tableau 3.1 montre différentes caractéristiques du nombre de jetons dans les systèmes analysés. En moyenne, on voit que le nombre de jetons contenus dans les fragments d'Eclipse est un peu plus élevé que dans Tomcat, mais dans les deux cas, les données indiquent une tendance dans les systèmes à contenir des fragments de petite taille. Un petit écart-type soutient cette hypothèse en indiquant une forte concentration de la taille des fragments autour de la moyenne.

Le tableau 3.2 montre le nombre de lot de tâches requis pour effectuer le calcul de la LCS sur toutes les grappes de clones identifiées dans un système. La nécessité des lots vient de la limitation en taille des tampons. Le nombre de lots requis est directement influencé par le nombre de fragments clonés dans un système ; on observe une proportion similaire entre le nombre de lots et le nombre de fragments dans les deux systèmes.

Le temps d'exécution mesuré sur les architectures CPU et GPU de l'algorithme calculant la LCS sont dans le tableau 3.6. Pour les deux systèmes, il y a une légère augmentation des performances lorsque l'algorithme est exécuté sur le GPU. Cette augmentation des performances est significativement plus élevée pour Eclipse. Cela est probablement causé par la plus grande cardinalité des grappes. Le temps requis pour extraire les jetons est également rapporté. On peut clairement voir que cette étape domine le calcul de la LCS.

Tableau 3.1 Statistiques sur la distribution des fragments dans les systèmes analysés

Systèmes	Eclipse	Tomcat
Nombre de fragments	92.439	10.585
Longueur moyenne en jetons	75,63	61,15
Écart-type de la distribution des jetons	6,54	5,69

3.7 Discussions

Les résultats confirment l'hypothèse de travail : le GPU peut améliorer les performances du filtrage des clones basé sur le calcul de la LCS.

Tableau 3.2 Nombre de lots de calculs sur GPU pour les différents systèmes

Système	Nombre de lots de calcul GPU
Eclipse	714
Tomcat	86

Tableau 3.3 Temps de calcul de la LCS sur CPU et sur GPU

System	Eclipse	Tomcat
Temps d'extraction des jetons	1.581,39	116,00
Temps de calcul de la LCS sur CPU	6,02	0,38
Temps de calcul de la LCS sur GPU	4,90	0,36

L'algorithme proposé montre un facteur d'accélération qui peut atteindre 1,2. Comme mentionnée à la section 3.5, la performance de l'algorithme est probablement meilleure avec des ensembles de chaînes plus longues, d'où la meilleure performance observée sur Eclipse. Toutefois, il faudrait une base de données expérimentale plus vaste pour pouvoir conclure à plus forte validité de l'hypothèse. Comme la distribution de la taille des fragments semble concentrée autour de la moyenne, le remplissage de chaque fragment est petit. Il est donc peu probable que l'on puisse améliorer le remplissage pour améliorer les performances.

L'observation des temps d'exécution met en relief la dominance du temps d'extraction des jetons par rapport au calcul de la LCS. Cela peut s'expliquer par l'organisation des données des jetons qui requièrent plusieurs accès au disque dur, une opération coûteuse unitairement. De plus, l'expérience a un biais dans la construction des grappes puisque l'égalité entre les métriques des fragments est exigée. Conséquemment, les fragments identiques ont plus forte probabilité d'être identifiée. Or, des fragments semblables, mais non identiques, pourraient être d'excellents clones candidats, mais ont très peu de chance d'être identifiés en amont par la méthode de détection. Par conséquent, les conclusions de l'expérience sont limitées à un échantillon de l'ensemble de filtrage total. Il est possible que sur l'ensemble au complet le gain en performance soit plus grand.

La littérature fait état d'un facteur d'accélération pouvant aller jusqu'à 3 sur des problèmes similaires en bio-informatique [39]. Cependant, puisqu'il s'agit de la première expérience en ce sens pour les clones logiciels, il est difficile de statuer sur la provenance exacte de la différence de performance : est-ce que la cause est une différence d'implémentation et d'architecture ou une faible similitude entre les jeux de données ? En bio-informatique, les chaînes de caractères utilisées ont des longueurs de l'ordre des milliers contrairement à des longueurs inférieures à la centaine pour les clones (voir dans le tableau 3.1). Même si cela suggère une incompatibilité dans les jeux de données, il ne faut pas négliger l'environnement expérimental et davantage d'expériences seraient

nécessaires.

Étant donné la possibilité d'améliorer les performances des méthodes de détection de clonage en utilisant un GPU, il est intéressant d'énumérer des problèmes différents du calcul de la LCS qui pourraient bénéficier de cette approche. D'abord, les techniques de détection de clonage basées sur les arbres de suffixes peuvent être transposées directement sur GPU suivant l'algorithme proposé par [44]. De plus, comme la technique de [44] se base sur les parcours d'arbre, il serait possible de collecter les métriques des AST avec une modification de la méthode proposée. Une tout autre technique de recherche de patrons dans les chaînes de caractères sur GPU a été proposée par [45] et pourrait devenir en soi une nouvelle technique de détection de clonage.

3.8 Conclusions

Cette section a présenté l'implémentation de l'algorithme du calcul de la LCS entre deux chaînes de caractères dans le but de filtrer des résultats produits par des analyses de clonage. Après expérimentation, un léger gain de performance a été observé. Cette section a également décrit les principales limitations des plateformes GPU et quelques solutions pour les contourner ont été proposées.

Bien qu'il ait été démontré la possibilité d'exploiter le GPU pour obtenir de meilleurs résultats de clonage, une autre technique pour améliorer les performances de la notion duale de la LCS, la distance de Levenshtein, offre des avantages supérieurs à la programmation sur GPU d'un point de vue de la performance et de simplicité de programmation. La prochaine section lui est consacrée.

CHAPITRE 4

CONSTRUCTION AUTOMATISÉE D'ORACLES DE CLONE PAR ARBRES DE MÉTRIQUES AVEC LA DISTANCE DE LEVENSHTEIN

4.1 Introduction

Le développement de nouvelles techniques de détection de clonage requiert une évaluation diligente de la qualité des résultats tout autant qu'une manière de la comparer avec les outils existants. L'évaluation de qualité se base généralement sur deux critères : le premier est l'efficacité en terme d'utilisation de ressources (temps d'exécution, consommation de mémoire, etc.) et le second la qualité des résultats en terme de faux positifs et faux négatifs. L'utilisation des ressources est simple à mesurer et à comparer, mais obtenir des résultats fiables en terme d'évaluation de qualité de résultats et plus difficile. Une manière de le faire est de comparer les résultats avec un oracle, qui, pour les méthodes de détection des clones, est une base de données de tous les clones acceptés dans un système. L'acceptation d'un ensemble de clones peut faire l'objet de débats et il n'existe pas d'ensemble universel unique à ce jour.

Plusieurs outils sont performants, mais ils offrent des résultats approximatifs étant donné qu'ils rapportent de faux positifs et des faux négatifs. Mesurer leur taux de faux positif requiert seulement de valider les candidats identifiés par l'outil pour trier les vrais et les faux positifs. Mesurer le taux de faux négatif nécessite un ensemble prédéterminé de clones considérés corrects duquel on déduit les faux négatifs.

La construction d'un bon oracle, c'est-à-dire un oracle permettant une mesure fiable des taux de faux positifs et de faux négatifs, doit contourner plusieurs problèmes encore ouverts. D'abord, une définition définitive de ce qu'est un clone doit être établie. Étant donné que plusieurs définitions sont acceptées, ce problème peut être résolu partiellement en acceptant une définition raisonnable et stricte au coût de limiter l'applicabilité de l'oracle. Ensuite, un oracle doit pouvoir être produit sur des systèmes de grande taille, il est suggéré ici d'utiliser au moins un million de lignes de code, pour être d'un intérêt pratique et permettre de produire de résultats significatifs en terme de mesure de faux positifs et de faux négatifs. Cet oracle devient donc dépendant d'une recherche exhaustive des clones au lieu d'une approximation. Cependant, les systèmes de cet ordre de grandeur imposent un coût prohibitif aux recherches exhaustives les rendant impraticables. Finalement, un oracle sur un grand système devrait pouvoir être produit automatiquement, même si la validation *a posteriori* reste utile. Les oracles utilisés aujourd'hui sont produits soient par validation manuelle, de manière artificielle à l'extérieur d'un système existant ou par échantillonnage statistique. L'objectif de ces

travaux est de réduire le temps de calcul nécessaire pour produire un oracle de qualité à un niveau raisonable et ce pour des systèmes de grande taille.

Dans cette section, une méthode est proposée pour construire un oracle pour les clones de types-1,2 et 3 ; la méthode est appliquée pour produire l'oracle de deux systèmes à code source ouvert, Tomcat et Eclipse. L'oracle est basé sur la distance de Levenshtein pour mesurer la similitude des fragments. Il contient toutes les paires de clones dans un système pour lesquelles la distance de Levenshtein est inférieure à un seuil sélectionné au moment de la construction de l'oracle. Les résultats obtenus peuvent servir de base de données de référence pour comparer différents outils de détection des clones en terme de taux de faux positifs et de faux négatifs étant donné qu'il contient toutes les paires satisfaisant le critère choisi. L'introduction d'un seuil d'acceptation des candidats visent le partitionnement de l'espace de recherche en différents secteurs afin d'éliminer des sous-ensembles de pairs sans avoir à les comparer explicitement. Si le seuil est suffisamment petit, les paires non analysées seront garanties très différentes, sans toutefois connaître exactement le degré de leur dissimilitude, et ne seront pas incluses dans l'oracle. L'oracle peut évidemment être construit pour tous les seuils possibles, mais comme il sera expliqué, le coût en temps augmente avec le seuil. En accord avec les trois critères d'un bon oracle mentionné ci-haut, la technique proposée les satisfait et semble intéressante.

Cette section est subdivisée comme suit : la section 4.2 survole les références littéraires pertinentes, la section 4.3 explique en détail les algorithmes et la méthodologie nécessaire à construire l'oracle, la section 4.4 présente les résultats de la construction d'oracles en utilisant la méthode proposée et la section 4.5 discute des utilisations possibles et des conséquences de cet oracle.

4.2 Revue de littérature

Il existe différentes techniques reconnues de détection de clonage. Pour la détection des clones de type-1 ou 2, la détection basée sur les AST a été introduite dans [13]. D'autres méthodes pour les types-1 ou 2 incluent la détection basée sur les vecteurs de métriques [5], les arbres de suffixes [15] et la détection par reconnaissance de chaînes [14]. D'autres approches peuvent également être trouvées dans les références suivantes [6, 7, 12, 17, 18, 20, 31, 54]. Si le lecteur est intéressé à lire en profondeur sur les techniques de détection de clonage publiées à ce jour, il est invité à lire le rapport exhaustif de Cordy et Roy [34].

Dans la suite de cette section, la construction d'oracles et les autres techniques d'évaluation sont abordées. Cette discussion est divisée en deux sections : les techniques pour les clones de type-1 et 2 et les techniques pour les clones de type-3.

4.2.1 Clones de type-1 et type-2

Construire un oracle pour les clones de type-1 est un problème simple à résoudre. À titre de rappel, un clone de type-1 et un clone textuellement identiques. L'utilisation d'une table de dispersement associative en utilisant la représentation textuelle des fragments comme clef et leur identifiant comme valeur permet de produire un oracle de manière automatique et pour de grands systèmes. Comme il est simple de formuler une définition claire des clones de type-1, il est simple de construire un oracle qui satisfait à une définition algébrique formelle. Les évaluations produites dans [1] et [55] incluent des oracles pour les clones de type-1.

Construire un oracle pour les clones de type-2 est plus complexe étant donné qu'il est nécessaire d'inclure les clones paramétriques. Même si [1] et [55] incluent également des oracles pour les clones de type-2, les oracles utilisés ne sont pas construits pour des systèmes complets. Bellon *et al.* [1] ont utilisé un oracle basé sur des statistiques des systèmes analysés, tandis que Cordy [55] utilise un oracle artificiel constitué de 16 cas typiques de clones. Les mêmes observations peuvent être faites pour leur oracle des clones de type-3.

Étant donné que les clones de type-3 forment un superensemble des clones de type-1 et type-2, il n'est pas nécessaire d'étendre cette discussion plus loin, car toutes les observations faites sur les clones de type-3 dans la prochaine section s'appliquent aux types inférieurs.

4.2.2 Clones de type-3

En ce qui concerne la détection de clones de type-3, Tiarks *et al.* ont publié une étude résumant l'ensemble des techniques actuelles [56]. Ce rapport classifie les clones de type-3 en deux sous catégorie :

– **clones à substitution structuelle,** fragments copiés auxquels on a substitué certaines des structures du système
– **clones modifiés**, fragments copiés auxquels on a ajouté ou supprimé du code.

Un clone de type-3 peut appartenir aux deux sous-catégories, leur définition n'étant pas mutuellement exclusive. La définition d'un clone de type-3 peut différer d'un auteur à l'autre et se confondre avec les clones de type-4 (clones sémantiques). Toutefois, la définition utilisée par Tiarks *et al.* convient parfaitement aux fins de définition d'un oracle automatisé.

Un point important dans l'étude de Tiarks est son utilisation de la distance de Levenshtein pour calculer un oracle. L'étude contient des résultats sur la distribution des clones détectés avec cette distance. Cependant, les résultats obtenus proviennent d'un petit échantillon de 751 pairs de clones dont 90% sont composés de fragments de moins de 65 LOC. De toute évidence, la faible portée d'application de la distance de Levenshtein dans cette étude provient du coût computationnel prohibitif qui lui est associé. Il faut donc retenir de cette étude la volonté d'utiliser cette distance, mais la nécessité d'améliorer ses performances à l'utilisation. La nouvelle méthode proposée dans ces travaux augmente le volume traiter par la distance de Levenshtein de plusieurs ordres de magnitude. Pour cette raison, les résultats obtenus sont différents et meilleurs que ceux de Tiarks.

Une autre étude intéressante de l'évaluation qualitative des outils de détection de clonage se retrouve dans [55]. Cette étude, tel que mentionné dans la discussion sur les clones de type-1 et 2, définit 16 patrons de clones typiques et détermine la performance de chaque outil pour chacun des scénarios. Neuf des cas présentés sont des cas de type-3 (selon notre définition). Selon cette étude, la méthode de détection de clonage basée sur les graphes a le meilleur potentiel de détection des clones de type-3. Les méthodes basées sur métriques, jetons et la représentation textuelle a aussi un bon potentiel de détection des cas présentés. Même si elles peuvent détecter les clones de type-3, les méthodes actuelles basées sur l'AST semblent être les moins efficaces. Néanmoins, cette étude conclut qu'aucune des méthodes actuelles ne peut identifier de manière fiable les 9 cas imaginés par l'auteur. Cette étude démontre également le bienfait des oracles construits à la main. Roy *et al.* ont également publié [57] un outil automatisé d'évaluation de qualité basé sur l'injection artificielle de clones dans des systèmes. Les clones injectés doivent respecter des patrons spécifiés *a priori* par l'évaluateur. Même s'il est possible avec cet outil de dépasser les 16 cas de l'étude précédente, il n'en reste néanmoins que les clones injectés sont soumis à l'imagination de l'évaluateur.

Enfin, une dernière étude d'intérêt pour les clones de type-3 est celle de Bellon [1]. Cette étude ne fournit pas de définition claire des clones de type-3 : elle définit les clones de type-3 comme tous les clones acceptés par un évaluateur expert qui ne sont pas de type-1 ou 2. De plus, l'étude a été réalisée sur un échantillon aléatoire des clones rapportés par les outils évalués. L'espace des clones était donc biaisé par le choix des outils évalués. Cela ne discrédite pas l'étude, mais cela démontre la difficulté de construire un oracle indépendant des méthodes évaluées tout en gardant

une définition des clones de type-3 claire et raisonnable. Un autre point d'intérêt de cette étude est le temps requis pour construire l'oracle. Bellon a travaillé 77 heures (3,2 jours ou 10,3 jours de travail régulier) pour classifier un échantillon de 6 518 candidats sur un total de 325 935, soit 2% de tous les candidats. Un travail identique sur tous les candidats aurait demandé 3 850 heures (160 jours) au même rythme. Un tel oracle, même exhaustif, souffre toujours du biais dû aux outils. Il est tout de même important de souligner que cette étude est une des premières à avoir évaluer avec autant de profondeur les outils de détection de détection de clones et est une référence dans le domaine.

La technique de construction d'oracle proposée ici a plusieurs caractéristiques propres qui la distinguent de celles présentées plus haut. Premièrement, elle est complètement automatisée. Deuxièmement, la méthode s'applique à grande échelle, sur des systèmes de l'ordre des millions de LOC sans l'utilisation de propriétés statistiques. Finalement, comme l'oracle construit est exhaustif en fonction d'un critère indépendant, il n'est pas fonction des outils choisis pour être évalués. Ces caractéristiques le rendent plus robuste que ceux connus et aussi le plus grand oracle à ce jour basé exclusivement sur un critère algébrique. Le choix de la distance de Levenshtein pour l'indépendance ne vient pas contredire notre définition acceptée des clones de type-3. En effet, puisque la distance de Levenshtein mesure la différence entre les fragments en nombre d'insertion, de suppression et de remplacement de jetons, les clones identifiés appartiennent à au moins une des deux sous catégories identifiées. Outre Tiarks, l'utilisation de la distance de Levenshtein pour des résultats de qualité est mise de l'avant par Merlo [5], Cordy [58] et Mende et Koschke [59] et [60]. Dans toutes ces publications, la distance de Levenshtein permet de mesurer et diminuer le taux de faux positifs produits par les outils. Par contraste, la nouvelle technique proposée permet de mesurer les taux de faux positifs et de faux négatifs.

En assumant le bien-fondé de la distance de Levenshtein (voir section 4.5, il est simplement nécessaire pour la technique de définir une organisation des données qui permet d'optimiser son calcul à travers toutes les paires de candidats. La prochaine section est consacrée à cette discussion.

4.3 Description de la méthode

Le but est de construire un oracle défini comme l'ensemble de toutes les paires de fragments d'un système pour lesquelles la distance de Levenshtein entre les deux fragments constituants est inférieure à un certain seuil. L'approche naïve est le calcul de la distance entre toutes les paires possibles et l'impression de toutes celles satisfaisant le critère. Toutefois, le nombre de pairs étant quadratique sur le nombre de fragments, cela est impraticable. Une approche du problème serait de trouver une ou plusieurs propriétés de la distance choisie afin d'organiser l'espace de recherche de toutes les paires afin de trouver plus facilement les sous-espaces des paires dont la distance est

inférieure au seuil choisi. L'arbre de métrique, tel que présenté par Ciaccia dans [61], est une structure de données qui organisent ses éléments en fonction de la distance qui les sépare si cette distance satisfait certains critères. Cette structure de données spécialisée résout exactement le problème posé par la construction d'oracle avec la distance de Levenshtein. La description de l'arbre de métrique et son application au problème à résoudre sont expliqués ici.

Le point clé de la technique réside dans l'utilisation d'une distance qui satisfait les axiomes d'une métrique dans un espace topologique. Soit X un espace topologique et δ une fonction $\delta : X \times X \rightarrow \mathbb{R}$ appelée une distance. Alors, la distance δ est une métrique si et seulement si les propriétés suivantes sont vraies pour tout $x, y, z \in X$:

$$\delta(x, y) \geq 0 \ (non\ negativite) \tag{4.1}$$

$$\delta(x, y) = \delta(y, x) \ (symmetrie) \tag{4.2}$$

$$\delta(x, y) = 0 \Leftrightarrow x = y \tag{4.3}$$

$$\delta(x, y) + \delta(y, z) \geq \delta(x, z) \ (inegalitedutriangle) \tag{4.4}$$

Si la métrique δ existe pour l'espace topologique X, alors X est un espace de métrique. Dans la majorité des cas, δ satisfait aux trois premiers axiomes de la métrique, mais pas au quatrième, l'inégalité du triangle. En fait, c'est la quatrième propriété qui permet de séparer l'espace pour optimiser les recherches. Plusieurs distances usuelles ne sont en fait pas des métriques. Par exemple, le coefficient de chevauchement et le coefficient de Dice sont des distances utiles, mais ne respectent pas l'inégalité du triangle. Cependant d'autres distances tout aussi communes comme le coefficient de Jaccard et la distance de Levenshtein respectent les quatre axiomes et sont des métriques. Pour cette raison, pour le reste de cette section, la distance de Levenshtein sera appelée la métrique de Levenshtein. Soit N l'espace de tous les fragments de code d'un logiciel avec la métrique de Levenshtein, alors N est l'espace de métrique désirée pour construire l'oracle.

En utilisant l'espace N, on peut maintenant construire la structure proposée par Ciaccia dans [61] : l'arbre de métrique. Cette structure possède deux primitives importantes : $insert(f)$ et $rangeQuery(f, \epsilon)$. La primitive $insert(f)$ prend comme argument un fragment et l'insert dans l'arbre. La primitive $rangeQuery(f, \epsilon)$ prend deux arguments : un fragment f et un nombre réel ϵ et retourne l'ensemble des fragments f' contenus dans l'arbre pour lesquels $\delta(f, f') \leq \epsilon$. Sous forme d'équation, on a :

$$rangeQuery(f, \epsilon) = \{f^{'} \mid \delta(f, f^{'}) \leq \epsilon\} \tag{4.5}$$

Les fragments f' sont au plus à distance ϵ du fragment f et sont ici considérés des clones de type-3 du fragment f. Une remarque importante à faire à ce point est que l'arbre de métrique ne teste que les fragments les plus susceptibles de satisfaire au critère et élimine les branches de l'arbre

qui ne peuvent plus satisfaire au critère grâce à l'inégalité du triangle. Par conséquent, en utilisant cette structure, le but de réduire la taille de la fouille exhaustive est atteint. La description des deux primitives est maintenant présentée en détail.

Le pseudo-code de la procédure $insert(f)$ est présenté à la figure 4.1. À la ligne 2, l'algorithme débute par l'initialisation de la variable $target$ avec comme valeur le noeud n_0, la racine de l'arbre. La variable $target$ représente le noeud dans lequel le nouveau fragment sera inséré. La boucle principale qui commence à la ligne 4 va assigner à $target$ différents noeuds durant l'exécution de l'algorithme, chaque nouvelle assignation étant un noeud d'une profondeur supérieur au précédent. Aux lignes 5 et 6 on calcule la distance de f avec les deux fragments, appelés pivots, contenu dans $target$. De la ligne 7 à 17, la distance de 7 avec les deux pivots est comparée à la distance entre les deux pivots. Une région dans l'arbre est sélectionnée en fonction des prédicats calculés avec ces distances. La boucle s'arrête lorsqu'un noeud dont au moins un des pivots n'est pas défini est découvert (si au moins un des pivots n'est pas défini, alors la distance entre les pivots n'est pas définie). Les lignes 17 à 22 vérifient quel pivot n'est pas définit et insert le fragment f à l'endroit approprié dans le noeud trouvé.

```
1  insert(f)
2      target = n_0
3      region = 0
4      while target.d! = UNDEFINED
5          d_1 = δ(target.x, f)
6          d_2 = δ(target.y, f)
7          if d_1 < target.d
8              if d_2 < target.d
9                  region = 0
10             else
11                 region = 1
12         else
13             if d_2 < target.d
14                 region = 2
15             else
16                 region = 3
17         target = target.c[region]
18     if target.x! = UNDEFINED
19         target.x = f
20     else
21         target.y = f
22         target.d = δ(target.x, target.y)
23     return
```

Figure 4.1 Algorithme d'insertion dans les arbres de métriques

La primitive $rangeQuery$ fonctionne d'une manière analogue dans la sélection des régions d'intérêt, mais avec des critères différents résumés dans le tableau 4.3. Dans ce tableau, y et x représente les pivots du noeud courant, f est le fragment de la requête, δ est la métrique, ϵ est le rayon de la requête et d est la distance entre les pivots, i.e. $d = \delta(x, y)$.

La primitive $rangeQuery$ s'exécute de manière récursive dans chaque région sélectionnée, en commençant par la racine. Cette procédure est décrite à figure 4.2. À la ligne 2, l'ensemble contenant le résultat de la requête est initialisé à l'ensemble vide. Les lignes 3 à 6 testent la distance entre la requête et les pivots avec le rayon désiré ϵ. S'il est plus petit ou égal, le pivot correspondant est ajouté à l'ensemble des résultats. Les lignes 7 à 14 test les prédicats de chaque région. S'ils sont vrais, $rangeQuery$ est appelé sur la région correspondante. Les quatre régions peuvent être visitées à chaque noeud, même si ce comportement est hautement improbable et il est attendu qu'au moins une région soit éliminée de la recherche à chaque étape. Sans en donner la démonstration, il est tout de même intuitif que cette procédure effectue moins de comparaison que la recherche exhaustive, surtout si le rayon est petit.

Tableau 4.1 Critère de sélection d'une région dans la primitive rangeQuery

Region 1	$\delta(x, f) < \epsilon + d \wedge \delta(y, f) < \epsilon + d$
Region 2	$\delta(x, f) + \epsilon \geq d \wedge \delta(y, f) < \epsilon + d$
Region 3	$\delta(x, f) < \epsilon + d \wedge \delta(y, f) + \epsilon \geq d$
Region 4	$\delta(x, f) + \epsilon \geq d \wedge \delta(y, f) + \epsilon \geq d$

```
1  rangeQuery(node, f, ε)
2      result = ∅
3      if δ(node.x, f) ≤ ε
4          result = result ∪ node.x
5      if δ(node.y, f) ≤ ε
6          result = result ∪ node.y
7      if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) < ε + node.d
8          result = result ∪ rangeQuery(node.region1, f, ε)
9      if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) < ε + node.d
10         result = result ∪ rangeQuery(node.region2, f, ε)
11     if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) + ε ≥ node.d
12         result = result ∪ rangeQuery(node.region3, f, ε)
13     if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) + ε ≥ node.d
14         result = result ∪ rangeQuery(node.region4, f, ε)
15     return result
```

Figure 4.2 Algorithme de recherche d'ensemble dans l'arbre de métrique

Pour plus de détails sur ces algorithmes, le lecteur se référera à [61]. La procédure détaillée construisant l'oracle des clones pour un système donné est présentée à la figure 4.3. Dans ce code,

$tree$ est un arbre de métrique tel que décrit ici avec les deux primitives $insert$ et $rangeQuery$. Deux étapes peuvent être distinguées dans cet algorithme. La première, des lignes 3 à 4, construit l'arbre en insérant tous les fragments de l'ensemble N. La seconde, des lignes 6 à 8, appelle $rangeQuery$ sur tous les fragments de N et mémorise les résultats dans la variable $clones$. La ligne 7 calcule le rayon de chaque requête.

Constuire l'oracle demande de trouver toutes les pairs de fragments de code pour lesquelles la distance de Levenshtein entre ses membres est inférieure ou égale au seuil, au rayon, spécifiée à $rangeQuery$. Le seuil choisi est fonction de la taille des fragments. Supposons que les fragments de code sont les chaînes de jetons produites par l'analyse lexicale des fragments initiaux. L'idée d'utiliser les jetons au lieu de la représentation textuelle provient de [18]. Soit a, b deux chaînes et $len(a), len(b)$ la longueur de ces chaînes. Alors, le seuil absolu entre ces deux chaînes est défini par :

$$\epsilon = distance * max(len(a), len(b))) \tag{4.6}$$

où $distance \in (0, 1)$ est le coefficient désiré de distance maximal. Une autre interprétation de $distance$ est exprimée par le degré de similitude : la similitude désirée est $(1.0 - distance)$. Le critère d'acceptation des clones peut alors être reformulé comme suit : la pair (a, b) est dans une relation de clonage *ssi* la distance de Levenshtein entre a et b est plus petite ou égale au seuil ϵ, où ϵ est la longueur maximale de a et b multipliée par le coefficient de distance relative (ou multipliée par un moins le coefficient de similitude). Cette phrase n'est qu'une explication de l'équation 4.3. Ainsi, le rayon ϵ de la requête est proportionnel à la longueur des fragments et le paramètre variable est le coefficient de distance et non la valeur de la métrique de Levenshtein. Ce choix d'utiliser des seuils proportionnels à la longueur n'est pas dû à une contrainte de l'arbre de métrique, mais plutôt un artifice de pseudonormalisation nécessaire pour éviter de détecter de mauvais clones. Voici un exemple qui illustre cette nécessité.

Il est facile de construire des fragments pour lesquels un seuil trop petit produira des faux négatifs. Par exemple, soit un fragment f_0 de taille 10 000 et un fragment f_1 de taille 13 000. Supposons que f_0 et f_1 aient un préfixe en commun d'une longueur de 10 000 jetons et que f_1 à un suffixe de 3 000 jetons supplémentaires. Si le seuil ϵ était en deçà de 3 000, l'algorithme ne trouverait pas ce clone évident. En pratique, un tel cas pourrait être deux classes clones dont une à 2 nouvelles méthodes. Cependant, le choix d'un seuil aussi grand produira invariablement des faux négatifs, considérant que la longueur moyenne des fragments est relativement petite. Par exemple, prenons f_2 et f_3 deux fragments de taille 200 pour lesquels $\delta_l(f_2, f_3) = 200$. Clairement, si $\epsilon = 3000$, f et f' seront considérés clones même s'ils ne partagent aucune similitude. Par conséquent, le seuil doit être sensible à la taille des fragments.

D'autres spécifications des seuils produisent des oracles différents. Néanmoins, la définition choisie semble raisonnable et sensée pour produire un oracle d'une qualité raisonnable.

Même si les arbres de métriques réduisent la taille de l'espace de recherche et le nombre de distances à calculer requises pour trouver les meilleurs candidats, leur utilisation ne peut à elle seule être suffisante pour calculer les oracles de systèmes dont la taille dépasse quelques centaines de KLOC. Cette affirmation provient de l'échec dans ces travaux de la construction de l'oracle pour Eclipse avec la technique décrite (tel que présenté dans la prochaine section 4.4). Le calcul s'est étendu sur 2 semaines sur un seul processeur, mais l'oracle n'était que partiellement construit. À partir des résultats partiels, le temps requis pour l'achèvement était estimé à 4 à 6 semaines.

Pour cette raison, il a été décidé de migrer le logiciel de calcul de l'oracle vers une architecture parallèle pour effectuer les calculs de $rangeQuery$. L'arbre étant déjà construit et écrit dans un fichier .xml, il est facile de distribuer le calcul des différentes requêtes avec un effort minimal. Il est nécessaire et suffisant pour calculer une partie de l'oracle de posséder l'arbre et une liste de requêtes. Si plusieurs coeurs possèdent l'arbre et des listes indépendantes de requêtes pour lesquelles leur union est l'ensemble total des requêtes, alors l'oracle peut être construit en parallèle avec un gain de temps proportionnel à un facteur constant. Étant donné que les processeurs à 2 ou 4 coeurs sont les plus répandus aujourd'hui, il est facile d'avoir 8,16 ou même 32 coeurs à sa disposition. Le calcul de grands oracles devient alors possible. La parallélisation sur GPU peut également être envisagée, comme celle proposée dans [3].

```
1  oracleClones(N, distance)
2      tree = new MetricTree
3      forall f ∈ N :
4          tree.insert(f)
5      clones = ∅
6      forall f ∈ N
7          radius = distance * len(f)
8          clones[f] = tree.rangeQuery(tree.root, f, radius)
9      return clones
```

Figure 4.3 Algorithme de construction d'oracles

4.4 Caractéristiques des oracles construits

) La construction d'oracle a été faite pour deux systèmes à code source ouvert en Java : Tomcat 5.5 et Eclipse 3.3 . Tomcat [36] est une implémentation des technologies Java Servlet et de Java Server Pages et est largement répandue pour soutenir différents types de systèmes orientés Web. Eclipse [37] est un IDE utilisé en milieu industriel pour plusieurs langages communs de program-

mation. La construction de l'oracle de Tomcat a été faite avec un processeur Intel Core 2 Duo, 2.16 GhZ couplé à 4 GB de mémoire vive sous Linux Fedora 13. L'oracle d'Eclipse a été calculé sur une grappe de 32 Opteron, 2.00 GHz couplés à 5 GB RAM sou Adelie Linux. Dans les deux cas, le logiciel de construction de l'oracle a été compilé avec g++ 4.4.4.

La taille, le nombre de fragments et plusieurs statistiques intéressantes sur les fragments sont présentés dans le tableau 4.4. Le nombre de LOC rapporté dans cette figure exclut tous les commentaires des fichiers. Étant donné que le calcul de la métrique de Levenshtein est quadratique sur la taille des fragments, les statistiques sur les fragments peuvent servir à estimer la difficulté à calculer les distances tout au long des requêtes.

Les fragments correspondent aux blocs, aux méthodes et aux classes. Les blocs peuvent être imbriqués, mais ne peuvent avoir d'autres formes d'intersection ou de chevauchement. Les séquences d'énoncés à l'intérieur d'un même bloc ne sont pas séparées davantage en sous-séquence pour l'instant. Si requis, il serait possible de construire des sous-séquences à partir de l'intérieur des blocs pour utiliser une granularité plus fine.

Dans le tableau 4.4 et dans tous les résultats qui suivent, les fragments font références à des blocs d'une longueur supérieure à 70 jetons, qui un seuil de significativité minimum pour un bloc. La littérature suggère d'utiliser une limite inférieure variant de 7 à 10 LOC. Une équivalence grossière peut être et l'expérience personnelle de l'auteur permet d'établir à 70 cette limite en terme de jetons.

Tableau 4.2 Caractéristiques des systèmes

Système	Tomcat	Eclipse
Version	5.5	3.3
LOC	130K	1.3M
Fragments	5084	129 258
Longueur moyenne des fragments en jetons	341.29	403.64
Longueur maximale des fragments en jetons	19999	128168

Les caractéristiques des oracles sont présentées dans le tableau 4.4. Les oracles ont été construits selon la procédure décrite à la section précédente pour un seuil ϵ déterminé pour chaque fragment avec un coefficient de distance de 0.3 (se référer à l'équation 4.3.

La valeur de 0.3 pour le coefficient de distance semble être une borne supérieure raisonnable, mais on ne peut le prouver formellement. Toutefois, les oracles présentés ici peuvent aisément être rapetissés à un oracle valide pour toute distance inférieure à 0.3. Chaque entrée dans l'oracle est un tuple $< f_1, f_2, \delta >$ où f_1 et f_2 sont des fragments dans une relation de clonage et δ la distance entre eux. Pour construire un oracle avec un coefficient de distance inférieur à 0.3 il est suffisant de lire les entrées une par une et de choisir seulement celle dont le seuil est inférieur à la nouvelle distance

désirée. Il est donc simple d'avoir des oracles plus astringents que ceux présentés.

Tableau 4.3 Nombre de paires de clones (a,b) dont la distance de Levenshtein est inférieure à 0.3 × max(len(a),len(b)) et le temps nécessaire pour les calculer

Systèmes	Tomcat	Eclipse
Nombre de paires détectées	2 933	316 728
Temps (h.)	2.67	148.80
Longueur maximal des clones en jeton	6534	115 141
Longueur moyenne des clones en jeton	181.99	222.31

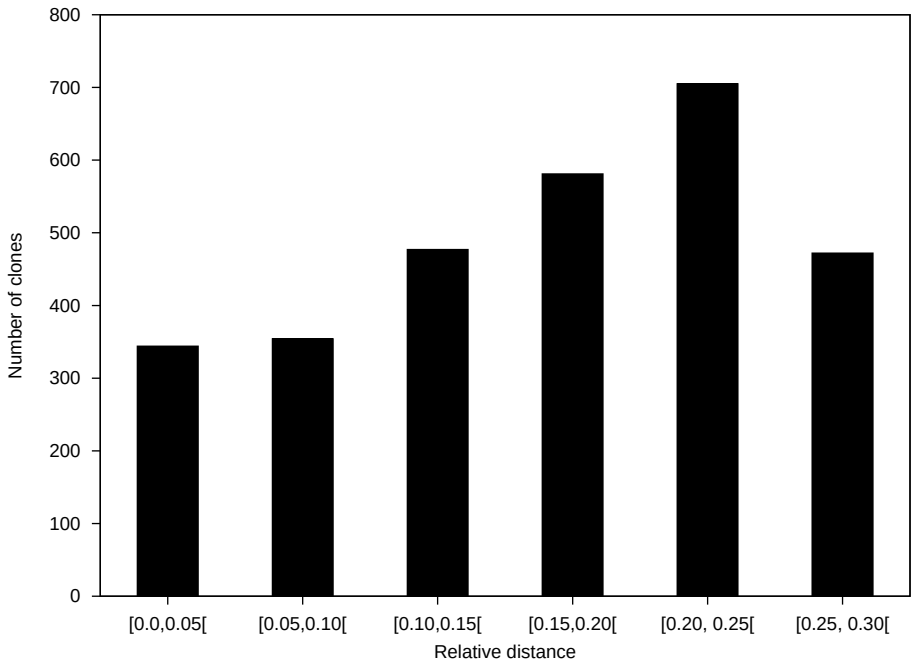


Figure 4.4 Distribution des pairs de clones en fonction de leur distance dans Tomcat

En se souvenant que l'oracle de Tomcat a été construit avec un seul coeur et que celui d'Eclipse a été construit avec une grappe de 32 coeurs, on conclut des résultats présentés dans le tableau 4.4 que le temps de calcul requis pour calculer l'oracle d'Eclipse est plusieurs milliers de fois plus élevé que celui requis pour calculer celui de Tomcat, même si leur taille est à l'intérieur d'un facteur 10. Deux facteurs expliquent cette différence : la grande différence entre le nombre de fragments et la différence dans la longueur moyenne des fragments. La cause définitive de cette énorme différence n'a pas été découverte et nécessite une plus grande collecte de données.

Les figures 4.4 et 4.5 montrent les distributions en fréquence des pairs de clones en fonction de la distance entre les fragments. La distance a été calculée comme mentionnée à la section 4.3. La distribution de Tomcat ressemble à une cloche inversée avec un maximum dans la classe

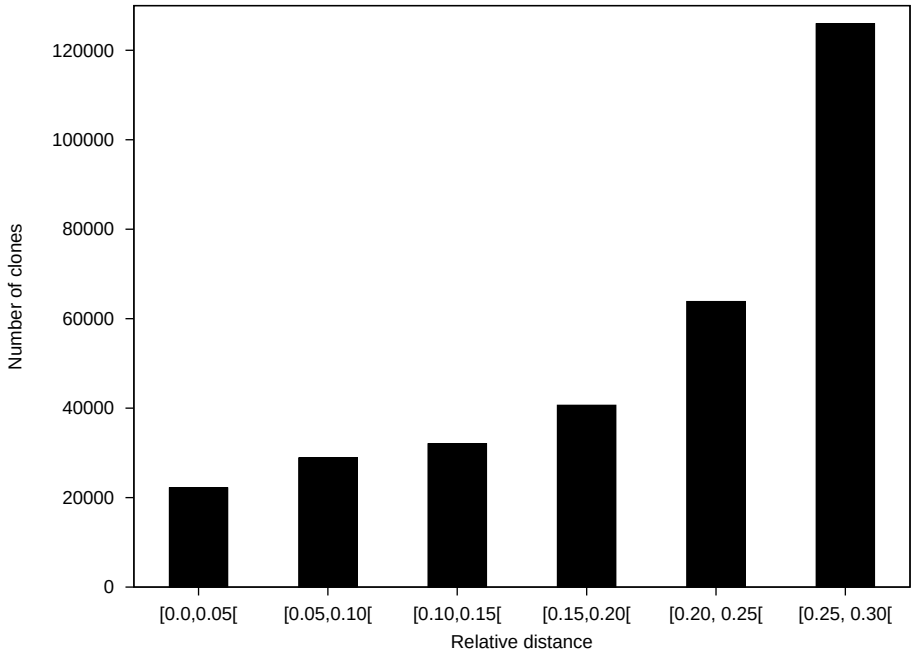


Figure 4.5 Distribution des pairs de clones en fonction de leur distance dans Eclipse

$[0.20, 0.25)$. La distribution d'Eclipse augmente de manière monotone. Dans les deux cas, la distribution suggère que l'existence de clones strictement de type-3 ce qui justifie l'étude de ce type de manière séparée. L'analyse des distributions n'étant pas l'objet de cette recherche, elle n'a pas été analysée davantage.

Un exemple de clone trouvé dans l'oracle de Tomcat est présenté à la figure 4.6. La longueur des fragments est respectivement de 177 pour 4.6a et de 145 pour 4.6b. La distance de Levenshtein entre les fragments est de 34 ou de manière équivalente de 0.192 selon l'équation 4.3. Cet exemple présente plusieurs caractéristiques des clones de type-3. D'abord, un préfixe a été ajouté au fragment le plus long. Ensuite, un suffixe y a été ajouté. Finalement, La majorité du code du petit fragment a été imbriqué dans une boucle dans le fragment le plus long. Ces différences ont été correctement identifiées par l'alignement, tel qu'indiqué en gras sur la figure.

4.5 Discussions

Les deux oracles construits et présentés dans la section 4.4 sont deux oracles complets pour la métrique de Levenshtein : ils contiennent un ensemble de pairs de clones pour lesquels la distance entre les fragments est inférieure au seuil choisi et cet ensemble serait identique s'il avait été obtenu par une recherche exhaustive des pairs satisfaisant ce critère. Il s'agit des deux premiers oracles produits automatiquement sur de grands systèmes suivant un critère objectif de similitude largement

accepté. En acceptant que la métrique de Levenshtein produise de bons résultats, alors on peut conclure que les oracles construits sont d'une excellente qualité pour les clones de type 3.

En comparaison avec les oracles construits par validation manuelle cités dans la section 4.2, la méthode présentée ici à l'avantage principal de pouvoir être applicable à des systèmes de grandes tailles en un temps raisonnable. À titre comparatif, l'oracle produit par Bellon a été construit en 77 heures ou environ 10 jours ouvrables de travail. À l'intérieur de ce temps, quelque 6 500 candidats ont été classés. L'oracle construit ici pour Eclipse, malgré qu'il ait été nécessaire le construire sur une grappe parallèle, a classé plus de 8 milliards de candidats avec la métrique de Levenshtein en 6.2 jours. Il est évident que ce gain est significatif par rapport au temps requis par la validation manuelle. De plus, la méthode proposée ne repose sur aucun ensemble préconstruit de clones, ni sur les outils existants, ni sur des clones injectés artificiellement. Par conséquent, ces oracles sont plus indépendants.

Les oracles des deux systèmes sont de tailles relativement petites en comparaison du nombre total de pairs de fragments. Pour Tomcat, il y existe 2 920 986 pairs de fragments desquels seulement 2 933 se qualifient comme pair de clones selon le critère choisi ; cela donne un taux de clonage de 0.02%. Pour Eclipse, des 8 353 750 653 pairs de fragments, seulement 316 728 sont des pairs de clones ce qui donne un taux de clonage de 0.004%. Ces nombres indiquent un taux très bas de vrais positifs à l'intérieur de l'espace de recherche et expliquent partiellement la bonne performance des arbres de métriques. Le faible taux de proximité à travers les fragments doit conduire à un meilleur partitionnement de l'espace et un meilleur taux d'élimination des branches lors des recherches dans l'arbre.

Au moment d'écrire ce mémoire, la construction d'un oracle tel que proposé pour un système de plus de 200 KLOCS semble prendre plus d'une semaine sur un ordinateur de bureau. Cependant, la grande majorité des ordinateurs ont aujourd'hui entre 4 et 8 coeurs. Comme le calcul proposé ne produit pas de résultats interdépendants, séparer le calcul peut être fait à la main et la computation peut être placée sur plusieurs machines physiques avec un effort minimal. Étant donné qu'il a été démontré que le calcul peut être effectué sur 32 coeurs avec un temps d'exécution raisonnable, l'utilisation de 4 à 8 ordinateurs dotés de processeurs modernes serait suffisant pour reproduire l'expérience. Par conséquent, la puissance computationnelle requise est un facteur limitant très faible.

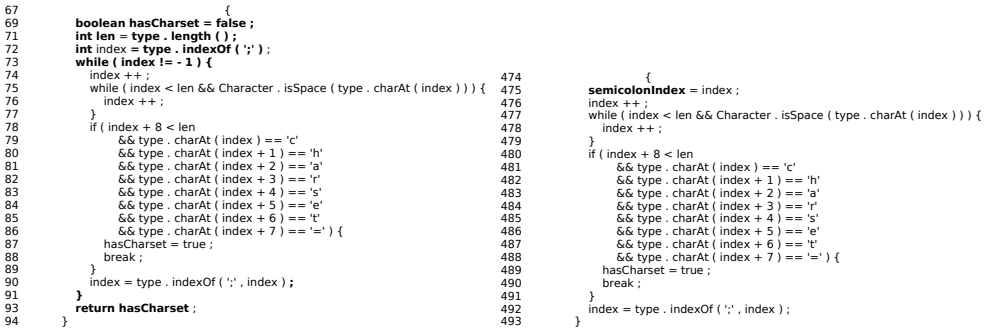
Les oracles construits reposent sur la métrique de Levenshtein et l'hypothèse qu'elle identifie des clones de bonne qualité. Certains chercheurs pourraient contester la validité de certains clones découverts par l'application de cette métrique en fonction de la définition de clonage choisie. Par observation des résultats, pour des distances élevées, les résultats pourraient être à la fois raisonnablement rejetés et acceptés, ce qui n'est pas vrai des clones à faible distance. La validation manuelle reste donc pertinente même avec la possibilité de produire des oracles de manière automatisée. En

fait, ces deux techniques peuvent être perçus comme complémentaires. Il est, toute proportion gardée, plus simple de valider manuellement un espace réduit par une technique automatique que de valider un espace de recherche entier.

Il existe quelques limitations aux résultats présentés. Étant donné qu'il est impossible de comparer les résultats obtenus avec ceux d'une recherche exhaustive aux fins de validation, la confiance dans les résultats ne peut être établie que par leur caractéristique qui semblent plausibles aux yeux de l'auteur. Le choix de la métrique de Levenshtein est en soi une limitation. Elle restreint l'oracle aux clones possédant des caractéristiques de similitude dans leur séquence de jetons. Même les clones partageant des structures syntaxiques contiennent des similitudes dans leurs séquences de jetons et seraient contenus dans l'oracle, l'oracle peut contenir des clones qui partagent peu de ressemblances syntaxiques, mais de fortes ressemblances textuelles. Il est donc possible que l'oracle contient des faux positifs selon certaines définitions du clonage. Néanmoins, les oracles ne peuvent contenir de faux négatifs à cause de la relation établie entre la ressemblance des jetons et la ressemblance de la syntaxe. Les oracles devraient donc être d'une bonne qualité pour les clones syntaxiques également. Une limite concrète de l'outil est son incapacité de traiter des clones de type 4, car la distance de Levenshtein ignore la sémantique des fragments. L'objectif étant de construire un oracle de qualité pour les clones de type 3, cette remarque n'affecte en rien la qualité des résultats. La construction d'un oracle pour les clones de type 4 est un problème fort différent et toujours ouvert.

L'exactitude pratique de la métrique de Levenshtein est également une limite. Quoiqu'elle soit considérée comme bonne par plusieurs auteurs (voir section 4.2) et qu'elle soit l'alignement mathématique optimal entre deux chaînes de caractères, des études empiriques devraient être effectuées pour en confirmer la pertinence pratique.

Comme dans toutes les sections de ce mémoire, la nature et le nombre de systèmes choisis est une limitation fondamentale de l'expérimentation. Les résultats montrent clairement une sensibilité aux caractéristiques des systèmes et par conséquent on peut difficilement extrapoler les résultats à d'autres systèmes, faute de validation statistique adéquate. De plus, des expériences sur des systèmes plus gros seraient nécessaires pour valider le passage à l'échelle possible de la méthode. Tous les fragments analysés étaient écrits en Java. Étant donné que différents langages peuvent présenter des caractéristiques de clonage différentes, la méthode pourrait ne pas être généralisable à d'autres langages. Cependant, elle est basée sur les jetons et les caractéristiques lexicales. L'écriture des analyseurs lexicaux étant plus facile que celle des analyseurs syntaxiques et l'alignement de séquences de jetons étant une opération indépendante du langage, la méthode serait cependant facilement applicable sur des systèmes écrits dans plusieurs langages impératifs et des langages de scripts.



```
67                                  {
69          boolean hasCharset = false ;
71          int len = type . length ( ) ;
72          int index = type . indexOf ( ';' ) ;
73          while ( index != - 1 ) {
74             index ++ ;
75             while ( index < len && Character . isSpace ( type . charAt ( index ) ) ) {
76                index ++ ;
77             }
78             if ( index + 8 < len
79                   && type . charAt ( index ) == 'c'
80                   && type . charAt ( index + 1 ) == 'h'
81                   && type . charAt ( index + 2 ) == 'a'
82                   && type . charAt ( index + 3 ) == 'r'
83                   && type . charAt ( index + 4 ) == 's'
84                   && type . charAt ( index + 5 ) == 'e'
85                   && type . charAt ( index + 6 ) == 't'
86                   && type . charAt ( index + 7 ) == '=' ) {
87                hasCharset = true ;
88                break ;
89             }
90             index = type . indexOf ( ';' , index ) ;
91          }
93          return hasCharset ;
94       }
```

(a) org/apache/coyote/Response.java lines 474-493

```
474                 {
475          semicolonIndex = index ;
476          index ++ ;
477          while ( index < len && Character . isSpace ( type . charAt ( index ) ) ) {
478             index ++ ;
479          }
480          if ( index + 8 < len
481                && type . charAt ( index ) == 'c'
482                && type . charAt ( index + 1 ) == 'h'
483                && type . charAt ( index + 2 ) == 'a'
484                && type . charAt ( index + 3 ) == 'r'
485                && type . charAt ( index + 4 ) == 's'
486                && type . charAt ( index + 5 ) == 'e'
487                && type . charAt ( index + 6 ) == 't'
488                && type . charAt ( index + 7 ) == '=' ) {
489             hasCharset = true ;
490             break ;
491          }
492          index = type . indexOf ( ';' , index ) ;
493       }
```

(b) org/apache/tomcat/util/http/ContentType.java lines 67-94

Figure 4.6 Exemple de pairs de clone provenant de l'oracle de Tomcat. La distance entre les fragments est de 0.192. Les différences dans l'alignement sont indiquées en gras.

4.6 Conclusions

Cette section a présenté une méthode pour construire des oracles pour évaluer les performances des outils de détection de clonage sur des systèmes de l'ordre des MLOC sans utiliser de propriétés statistiques. La technique est complètement automatisée. Les oracles présentés forment une référence des clones de type 3 dans les systèmes analysées. Les oracles de Tomcat et Eclipse ont été construits et leurs caractéristiques ont été présentées.

Des recherches futures pourront mettrent l'emphase sur l'amélioration du temps de calculs en utilisant soit des alternatives aux arbres de métriques ou en exploitant de nouvelles architectures parallèles comme les grappes massives ou les GPU. L'objectif principal à accomplir est de réduire le temps de calculs jusqu'à le rendre acceptable pour la production d'oracles de grands systèmes avec un minimum de ressources.

CHAPITRE 5

CONCLUSIONS

5.1 Synthèse des travaux

Les travaux présentés ont atteint les trois objectifs de recherche proposés :

– établir l'existence et l'utilité d'étudier les relations syntaxiques entre les clones autres que la relation fonction à fonction ;
– déterminer l'applicabilité d'une approche de filtrage sur GPU des clones, particulièrement en terme de gain computationnel ;
– décrire une technique de construction automatisée d'une base de données de référence, un oracle, pour les clones de type 3.

À chaque objectif, on peut associer un résultat clair :

– il existe bel et bien plusieurs relations syntaxiques entre les clones : fonction à fonction, bloc à fonction et bloc à bloc. Cette relation est utile et apporte une information qui n'est pas capturée par la relation fonction à fonction ;
– le GPU a une applicabilité très limitée dans le problème de filtrage des clones par LCS ;
– il est possible de construire des oracles automatisés pour les clones de type 3 en utilisant un arbre de métriques et la distance de Levenshtein.

Les objectifs de recherche ont donc été atteints et plusieurs propriétés structurelles et computationnelles ont été établies. En plus, une base de données exhaustive de références aux fins de comparaison a été produite dans le but d'améliorer l'état des connaissances sur la fiabilité des résultats produits par les différentes analyses de clonage. Par extension, la méthode d'évaluation présentée constitue en elle-même une nouvelle technique originale de détection des clones.

5.2 Limitations des résultats présentés

La principale limitation est la faible portée des résultats due à un choix restreint de systèmes ciblés par les expériences. La grande taille des systèmes contrebalance cette limitation et donne une force intéressante aux résultats obtenus.

5.3 Travaux futurs

L'élargissement de la base expérimentale est une amélioration à court terme à apporter à ces travaux. Dans le cas des relations syntaxiques de clonage, une étude approfondie de leur origine

et de leur impact pourrait améliorer la compréhension des systèmes dans lesquels on les trouve. Une exploration de différentes plateformes GPU pourrait fournir davantage de pistes d'explication des performances mitigées obtenues ici. Finalement, l'amélioration de la technique basée sur les arbres de métriques pourrait conduire à une technique de détection de clonage en soi et fournir un excellent outil de détection des clones de type 3.

RÉFÉRENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering - IEEE Computer Society Press*, 33(9) :577–591, 2007.

[2] Ettore Merlo and Thierry Lavoie. Computing structural types of clone syntactic blocks. In *Working Conference on Reverse Engineering*, pages 274–278. IEEE, 2009.

[3] T. Lavoie, M. Eilers-Smith, and E. Merlo. Challenging cloning related problems with GPU-based algorithms. In *IWSC10 Proceedings of the 4th International Workshop on Software Clones*, pages 25–32, 2010.

[4] T. Lavoie, , and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *IWSC11 Proceedings of the 5th International Workshop on Software Clones*, pages 34–40, 2011.

[5] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 244–253, Monterey, CA, Nov 1996.

[6] Ettore Merlo, Giuliano Antoniol, Massimiliano Di Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 412–416. IEEE Computer Society Press, 2004.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis as a basis for object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 98–107. IEEE Computer Society Press, 2000.

[8] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1037–1043. ACM Press, 2006.

[9] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3 :77–108, March 1996.

[10] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 314–321, 1997.

[11] E. Merlo. Detection of plagiarism in university projects using metrics-based spectral similarity. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. IBFI.

[12] B. Baker. Finding clones with dup : Analysis of an experiment. *IEEE Transactions on Software Engineering - IEEE Computer Society Press*, 2007.

[13] I.D. Baxter, A. Yahin, l : Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 368–377, 1998.

[14] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution : Research and Practice - Wiley InterScience*, (18) :37–58, 2006.

[15] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 219–228. IEEE Computer Society, 2009.

[16] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183, October 1993.

[17] T. Kamiya. Variation analysis of context-sharing identifiers with code clone. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*. IEEE Computer Society Press, 2008.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder : A multi-linguistic token-based code clone detection system for large scale source code. volume 28, pages 654–670. IEEE Computer Society Press, 2002.

[19] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering - IEEE Computer Society Press*, pages 1–17, 2006.

[20] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE '01 : Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 107, Washington, DC, USA, 2001. IEEE Computer Society.

[21] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident : An empirical study of source code cloning across software systems. In *International Symposium on Empirical Software Engineering*, 2005.

[22] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained : An empirical study. In *European Conference on Software Maintenance and Reengineering*, 2007.

[23] R. Falke, P.Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering Journal*, 13(6) :601–643, 2008.

[24] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2005.

[25] C.K. Roy and J.R. Cordy. Scenario-based comparison of clone detection techniques. In *International Conference on Program Comprehension*, pages 153–162. IEEE Computer Society Press, 2008.

[26] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the Working Conference on Reverse Engineering*, 2008.

[27] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2007.

[28] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *Workshop on Source Code Analysis and Manipulation*, 2007.

[29] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing clone evolution in the linux kernel. *Information and Software Technology*, pages 755–765, 2002.

[30] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, 2007.

[31] J. Guo and Y. Zou. Detecting clones in business applications. In *Proceedings of the Working Conference on Reverse Engineering*, 2008.

[32] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, S. Teuchert, and J. F. Girard. Clone detection in automotive model-based development. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, 2008.

[33] University of Alabama at Birmingham. Clone literature, 2011. ”http ://students.cis.uab.edu/tairasr/clones/literature”.

[34] C.K. Roy and J.R. Cordy. A survey on software clone detection research. Technical Report Technical Report 2007-541, School of Computing, Queen’s University, November 2007.

[35] Ettore Merlo and Thierry Lavoie. Detection of structural redundancy in clone relations. Technical Report EPM-RT-2009-05, Ecole Polytechnique of Montreal, 2009.

[36] Apache foundation. tomcat, 2011. ”http ://tomcat.apache.org”.

[37] Eclipse foundation. eclipse, 2011. ”http ://www.eclipse.org”.

[38] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. International Software Metrics Symposium*, pages 292–303. IEEE Computer Society Press, 1999.

[39] S.A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. In *BITS :Annual Meeting*, Naples, Italy, April 2007.

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, chapter 15. MIT Press, second edition, 2001.

[41] I. Buck and T. Purcell. *GPU Gems*, chapter 37, pages 621–636. Addison Wesley, 2004.

[42] P. Micikevicius. *GPU Gems 2*, chapter 43, pages 695–702. Addison Wesley, 2005.

[43] P. Kipfer and R. Westermann. *GPU Gems 2*, chapter 46, pages 733–746. Addison Wesley, 2005.

[44] M.C. Schatz and C. Trapnell. Fast exact string matching on gpu. http ://www.nvidia.com, 2007.

[45] E. Seamans and T. Alexander. *GPU Gems 3*, chapter 35, pages 771–783. Addison Wesley, 2008.

[46] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Symposium on Applications Specific Processors, 2008*, pages 101–107, 2008.

[47] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Allan Miller, and Michael Upton. Hyper-threading technology. 6, february 2002.

[48] NVIDIA. Nvidia cuda compute unified device architecture, programming guide version 1.0. http ://www.nvidia.com, 2007.

[49] AMD/ATI. Ati ctm guide technical reference manual, version 1.01. http ://ati.amd.com, 2006.

[50] Khronos Group. Opencl-the open standard for parallel programming of heterogeneous systems. http ://www.khronos.org/opencl/, 2009.

[51] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 244–253, Monterey, CA, Nov 1996.

[52] Ettore Merlo, Giuliano Antoniol, Massimiliano Di Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 412–416. IEEE Computer Society Press, 2004.

[53] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9) :577–591, 2007.

[54] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based code clone detection : incremental, distributed, scalable. *Software Maintenance, IEEE International Conference on*, 0 :1–9, 2010. doi : http://doi.ieeecomputersociety.org/10.1109/ICSM.2010.5609665.

[55] C.K. Roy, J.R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools : a qualitative approach. 74(7) :470–495, may 2009.

[56] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Workshop on Source Code Analysis and Manipulation*, pages 67–76. IEEE Computer Society Press, 2009.

[57] Chanchal K. Roy and James R. Cordy. A mutation / injection-based automatic framework for evaluating clone detection tools. In *ICSTW09 International Conference on Software Testing, Verification and Validation Workshops*, pages 157–166, 2009.

[58] C.K. Roy and J.R. Cordy. NICAD : Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *International Conference on Program Comprehension*, pages 172–181. IEEE Computer Society Press, 2008.

[59] T. Mende, R. Koschke, and F. Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution*, 21(2) :143–169, march-april 2009.

[60] T. Mende, F. Beckerwert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *CSMR '08 Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering - IEEE Computer Society Press*, pages 163–172, 2008.

[61] P. Ciaccia, M. Patella, and P. Zezula. M-tree : An efficient access method for similarity search in metric spaces. In *Proc. of 23rd International Conference on Very Large Data Bases*, pages 426–435. Morgan Kaufmann Publishers, 1997.