Recherche de similarité dans du code source

Michel Chilowicz

Thèse de doctorat de l'Université Paris-Est

École doctorale MSTIC — Laboratoire d'Informatique Gaspard-Monge (LIGM)

Spécialité : informatique

Thèse présentée par Michel CHILOWICZ
Soutenue le 25 novembre 2010

Recherche de similarité dans du code source

Looking for Similarity in Source Code

Jury

Gilles ROUSSEL	Professeur	Université Paris-Est Marne-la-Vallée	Directeur
Thierry LECROQ	Professeur	Université de Rouen	Rapporteur
Didier PARIGOT	Chargé de Recherche	INRIA Sophia Antipolis	Rapporteur
Julien ALLALI	Maître de Conférences	Université de Bordeaux 1	Examinateur

UNIVERSITÉ PARIS-EST

Remerciements

Une thèse comprend des moments d'adversité et de joie, les seconds existant par contraste aux premiers. Au-delà du contenu même de ce document coexistent ces moments ainsi que la marque de personnes ayant aidé à son élaboration.

Je voudrais ainsi tout d'abord exprimer toute ma gratitude à mon directeur de thèse, Gilles Roussel, m'ayant fait confiance dans cette aventure en me proposant ce sujet intéressant et ouvert. Mes remerciements vont également à Étienne Duris pour ses idées, son aide précieuse et ses conseils avisés en toute circonstance (avec une mention spéciale pour l'expédition britannico-colombienne).

Pour la dimension enseignante, je tiens à adresser mes remerciements à (encore) Étienne Duris ainsi que Marc Zipstein, Sylvain Lombardy, Nicolas Bedon, Stéphane Lohier, Sébastien Paumier, Marjorie Grzeskowiak et Cyril Nicaud ainsi que les étudiants dont je fus amené à croiser la route.

Mes remerciements ne sauraient être complets sans mentionner l'ensemble des personnes qui ont participé, consciemment ou inconsciemment et à des degrés divers à ce travail. Il s'agit bien sûr de l'ensemble des personnes dont les noms sont cités dans la bibliographie clôturant ce document. Que soient également remerciés l'ensemble des membres du LIGM (HP, Line...) que je ne peux citer exhaustivement par manque d'espace ainsi que les doctorants dont j'ai partagé le bureau au quatrième étage du bâtiment Copernic (Gautier, François, Elsa, Julien, Hieu, Nadia et Ibtissem) pendant ces années.

Merci à toute famille de m'avoir supporté avec une pensée particulière à mes parents.

Pour terminer, il est nécessaire de remercier d'avance le lecteur de ce mémoire et de lui demander toute sa mansuétude pour les probables erreurs, inexactitudes et points obscurs qui subsisteraient. Qu'il n'hésite pas à me contacter (par courrier électronique) pour les signaler. Parmi les infatigables lecteurs de cette thèse, trois méritent une attention toute particulière. Il s'agit de Thierry Lecroq et Didier Parigot que je remercie pour avoir accepté de rapporter ma thèse et pour leurs remarques sur le manuscrit et conseils de pré-soutenance. De nombreux mercis également à Julien Allali pour son rôle d'examinateur.

American English thematic key-words

General	source code, similarity, clone, matches, plagiarism, similarity metrics
Source representation	token sequence, n-gram, (abstract) syntax tree, AST, call graph
Process	tokenization, lexical analysis, lexing, syntactic analysis, parsing, suffix indexing, factorization, hashing, match consolidation, clustering
Data structures	suffix table, suffix tree, interval tree, maximal repeated factors graph

Abstract

Several phenomena cause source code duplication like inter-project copying and adaptation or cloning inside a same project. Looking for code matches allows to factorize them inside a project or to highlight plagiarism cases. We study statical similarity retrieval methods on source code that may be transformed via edit operations like insertion, deletion, transposition, in- or out-lining of functions.

Sequence similarity retrieval methods inspired from genomics are studied and adapted to find common chunks of tokenized source. After an explanation on alignment and k-grams lookup techniques, we present a factorization method that merges function call graphs of projects to a single graph with the creation of synthetic functions modeling nested matches. It relies on the use of suffix indexation structures to find repeated token factors.

Syntax tree indexation is explored to handle huge code bases allowing to lookup similar sub-trees with their hash values computed via heterogeneous abstraction profiles. Exact copies of sub-trees close in their host trees may be merged to get approximate and extended matches.

Before and after match retrieval, we define similarity metrics to preselect interesting code spots, refine the search process or enhance the human understanding of results.

Mots-clés thématiques français

Général	code source, similarité, similitude, clone, correspondances, plagiat, métrique de similarité
Représentation de code	séquence de lexèmes, n-gram, arbre de syntaxe, AST, graphe d'appel
Processus	lexémisation, analyse lexicale, analyse syntaxique, indexation de suffixes, factorisation, hachage, consolidation de correspondances, regroupement
Structures de données	table de suffixes, arbre de suffixes, farmax

Résumé

La duplication de code source a de nombreuses origines : copie et adaptation inter-projets ou clonage au sein d'un même projet. Rechercher des correspondances de code copié permet de le factoriser dans un projet ou de mettre en évidence des situations de plagiat. Nous étudions des méthodes statiques de recherche de similarité sur du code ayant potentiellement subi des opérations d'édition telle que l'insertion, la suppression, la transposition ainsi que la factorisation et le développement de fonctions.

Des techniques d'identification de similarité génomique sont examinées et adaptées au contexte de la recherche de clones de code source sous forme lexemisée. Après une discussion sur des procédés d'alignement de lexèmes et de recherche par empreintes de k-grams, est présentée une méthode de factorisation fusionnant les graphes d'appels de fonctions de projets au sein d'un graphe unique avec introduction de fonctions synthétiques exprimant les correspondances imbriquées. Elle utilise des structures d'indexation de suffixes pour la détermination de facteurs répétés.

Une autre voie d'exploration permettant de manipuler de grandes bases indexées de code par arbre de syntaxe est abordée avec la recherche de sous-arbres similaires par leur hachage et leur indexation selon des profils d'abstraction variables. Des clones exacts de sous-arbres de forte proximité dans leurs arbres d'extraction peuvent alors être consolidés afin d'obtenir des correspondances approchées et étendues.

En amont et en aval de la recherche de correspondances, des métriques de similarité sont définies afin de préselectionner les zones d'examen, affiner la recherche ou mieux représenter les résultats.

Table des matières

Notations

Suites et chaînes

Soit un alphabet de lexèmes énumérables $\Sigma = \{\sigma_1, \sigma_2, \cdots\}$:

- $|\Sigma|$ est la cardinalité de cet alphabet.
- Σ^k est l'ensemble des chaînes de longueur k, $\Sigma^{<k}$ l'ensemble des chaînes de longueur strictement inférieure à k.
- $\Sigma^* = \Sigma^{\geq 0}$ est l'ensemble des chaînes sur Σ de longueur finie, $\Sigma^+ = \Sigma^{\geq 1}$ l'ensemble des chaînes de longueur finie non nulle.

Soit u une chaîne (ou suite finie) de Σ^*, $u = u_1 u_2 \cdots u_n = u[1]u[2]\cdots u[n]$ de longueur n, nous définissons les notations suivantes :

- $u[i..j]$ (avec $1 \leq i \leq j \leq n$) est le facteur $u_i u_{i+1} \cdots u_j$ de longueur $j - i + 1$. Ce facteur est également noté $u\,[i|j - i + 1]$.
- $u[i..]$ est le suffixe de u, $u_i u_{i+1} \cdots u_n$, débutant à l'indice i.
- $|u|$ est la longueur de u, i.e. $|u| = n$.
- ϵ est la chaîne de longueur nulle. Par convention, $u[i..i-1] = u[0..0] = \epsilon$.

Soient u et v deux chaînes dont les éléments disposent d'une relation d'égalité $=$, nous avons :

- $u = v$ ssi $|u| = |v|$ et pour tout $i \in [1..|u|]$, $u_i = v_i$ (égalité).
- $u \cdot v = u_1 u_2 \cdots u_n v_1 v_2 \cdots v_n$ (concaténation).
- v est préfixe de u ssi $v = u[1..|v|]$.
- $lcp(u, v)$ est le plus long préfixe commun (en anglais *Longest Common Prefix*) de u et v. Sa longueur est l et nous avons (pour $u \neq v$) $u[1..l] = v[1..l]$ et $u_{l+1} \neq v_{l+1}$. Si $u_1 \neq v_1$, $lcp(u, v) = \epsilon$; si $u = v$, $lcp(u, v) = u = v$.
- v est un facteur de u ($v \in \text{fact}\,(u)$) ssi il existe au moins un couple (i, j) tel que $v = u[i..j]$. ϵ est facteur de toute suite finie.
- v est un suffixe de u ($v \in \text{suff}\,(u)$) ssi $v = u[|u| - |v| + 1..|u|]$. ϵ est suffixe de toute chaîne.
- v est un préfixe, suffixe ou facteur *propre* de u ssi v est respectivement un préfixe, suffixe ou facteur de u et $u \neq v$.
- v est une suite extraite (ou sous-séquence) de u ($v \in \text{extr}\,(u)$) ssi il existe des indices $i_1, i_2, \cdots, i_{|v|}$ tels que $v[1] = u[i_1], v[2] = u[i_2], \ldots, v[|v|] = u[i_{|v|}]$ avec $1 \leq i_1 < i_2 < \cdots < i_{|v|} \leq |u|$. ϵ est une suite extraite de toute suite. *A fortiori*, un facteur de u est une suite extraite de u.

Soit $T = \{t_1, t_2, \cdots, t_k\}$ un ensemble ordonné de k chaînes, nous avons pour $1 < i < k$, $T_{<i}$ (resp. $T_{>i}$) désigne les chaînes d'indice strictement inférieur à i (resp. d'indice strictement

supérieur à i).

Graphes et arbres

Soit G un graphe défini par ses nœuds et un ensemble d'arcs orientés d'un nœud vers un autre. Nous notons :

- $|G|$ le nombre de nœuds du graphe G.
- $\mathcal{S}(\alpha)$ l'ensemble des successeurs directs du nœud α, i.e. les nœuds de destination des arcs partant de α ; $\leftarrow(\alpha)$ est le cardinal de cet ensemble (l'arité sortante du nœud α).
- $\mathcal{P}(\alpha)$ l'ensemble des prédécesseurs directs du nœud α, i.e. les nœuds d'origine des arcs atteignant α ; $\rightarrow(\alpha)$ est le cardinal de cet ensemble (l'arité entrante du nœud α).

$A = a(a_1, a_2, \cdots, a_n)$ est l'arbre de racine a et comportant pour sous-arbres enfants $a_1, a_2, \cdots, a_n$. Nous notons :

- hauteur(A) la hauteur de l'arbre A égale à $1 + \max(\text{hauteur}(a_1), \text{hauteur}(a_2), \cdots, \text{hauteur}(a_n))$. La hauteur d'un arbre sans sous-arbre enfant est 1.
- $A[k]$ $(1 \leq k \leq |A|)$ le k^{e} nœud de l'arbre A obtenu par un parcours en largeur. Par exemple, pour $2 \leq k \leq n+1$, $A[k]$ est la racine de a_{k-1}.
- $\mathcal{S}(A) = \{a_1, a_2, \cdots, a_n\}$ l'ensemble des successeurs (enfants) de A.
- $\mathcal{P}(a_k) = A$ le parent de a_k.
- racine(A) la racine de A, à savoir a.
- feuilles(A) l'ensemble des feuilles de l'arbre A, i.e. l'ensemble de ses nœuds d'arité sortante nulle.

Introduction

Introduction à la problématique de la recherche de similarité dans du code source

Depuis la nuit des temps, l'homme, pour survivre et améliorer ses conditions de vie a dû apprendre à reconnaître et formaliser son environnement pour mieux le maîtriser. Qu'il s'agisse de la reconnaissance d'un danger imminent ou la conception de théories déductives basées sur l'observation, le cerveau humain dispose d'une capacité impressionnante de mise en relation d'informations acquises, par analogie ou au contraire par contraste. Tout comme le langage et l'écriture ont constitué des étapes essentielles de l'évolution de l'humanité, le développement de l'informatique a marqué un tournant essentiel en permettant le traitement de masses d'information qu'il aurait été impossible de manipuler pour un cerveau humain. Mettre en lumière la similarité ou les différences au sein de données a été et demeure un problème central en informatique. Si pour certains problèmes bien formalisés nous disposons de méthodes efficaces et exactes de détermination de similarité, ceci n'est pas le cas pour tout type de données où la définition de similarité repose sur une subjectivité humaine difficilement formalisable pour être comprise par une machine. Ainsi, rechercher un facteur exact sur un volume conséquent de chaînes de caractères de longueurs importantes ne pose aujourd'hui pas de problème majeur alors qu'identifier la présence d'un élément sur une photographie est plus hasardeux.

Les langages informatiques, tout comme les langages que nous qualifions de naturels, sont des constructions humaines, mais à la différence des seconds ceux-ci ont été élaborés sur un court laps de temps, par un nombre limité de personnes avec un certain niveau de formalisme. Ces langages, qui interprétés ou compilés en un langage directement compréhensible par la machine, permettent d'exécuter une série d'instructions conditionnées par des données en entrée. Dès lors, comment définir la similarité entre deux programmes ? S'il s'agit de considérer comme équivalente la similarité entre programmes et la similarité de leurs sorties pour des entrées identiques, des tests d'exécution sur toutes les entrées sont nécessaires. Si ce domaine est infini, le temps nécessaire à la détermination de leur similarité le sera aussi. Nous nous intéressons donc plutôt à des tests statiques par la seule comparaison de la description des programmes dans leur langage, i.e. leur code source. Rice, par la généralisation du problème de l'arrêt, montre que la vérification d'une propriété non-triviale d'un programme définissant une machine de Turing est indécidable [18]. Il n'existe donc pas de programmes vérifiant la similarité algorithmique de programmes en temps fini. La notion de similarité doit donc être définie autrement pour pouvoir être manipulée par une machine.

Mais pourquoi rechercher des similitudes au sein d'un code source de programme ou entre codes sources de différents programmes ? Deux problématiques majeures sont soulevées. La

première consiste à relever des duplications plus ou moins involontaires au sein d'un projet afin de les éliminer dans une optique d'amélioration de sa maintenance. La seconde a pour objectif de détecter des copies volontaires entre différents projets.

La complexité croissante de certains projets conduit inévitablement à l'apparition de code dupliqué en leur sein. Ainsi, généralement, lorsqu'un développeur doit gérer un problème de nature similaire au sein d'un contexte différent, il peut s'avérer tentant de copier du code existant dans le projet et d'y réaliser quelques modifications plutôt que de le généraliser pour le factoriser entre les différents contextes. D'autre part des clones accidentels entre différents programmeurs peuvent survenir au cours du cycle de développement. Réduire le volume de code par factorisation de code dupliqué permet d'améliorer ses performances et sa sécurité dès lors que l'effort de maintenance est lui aussi factorisé [66]. Certaines études empiriques de recherche de clones ont ainsi montré que certains projets Open Source étaient caractérisés par un taux de duplication de 5 à 20%.

La recherche de similarité sur du code source rejoint donc les problématiques de compression non destructive de données : il s'agit d'identifier les données redondantes pour ensuite adopter une nouvelle représentation par factorisation les éliminant. Il n'est donc pas étonnant que certains outils de recherche de similitudes sur du code source s'inspirent de méthodes populaires de compression de chaînes de lexèmes tel que Lempel Ziv 77 [24] recherchant des facteurs répétés.

Notre problématique possède également des liens forts avec l'identification de séquences biologiques (ADN, ARN ou protéines) similaires. De même qu'il est possible d'établir une taxonomie des êtres vivants par l'analogie de leur séquences biologiques, il est possible de classifier des projets informatiques communs. Des méthodes telles que l'alignement de séquences par programmation dynamique ou la recherche sur base par k-grams, initialement utilisées en génomique, sont employées avec un certain succès pour la recherche de clones de code.

Mais tout comme une bactérie peut apporter sa contribution à l'évolution d'un autre être vivant, de petits morceaux de code d'un projet informatique peuvent être intégrés dans un autre. Ces copies inter-projets peuvent être légitimes lorsque les licences respectives des projets impliqués l'autorisent ou alors illégitimes. Mettre en évidence des clones, peut, dans ce second cas, mettre en lumière des violations de licence consistant par exemple à l'emprunt d'un morceau de code sous licence libre par un projet propriétaire ou vice-versa.

Mais la motivation liée à la copie illégitime de code source entre projets n'a pas nécessairement de nature économique ou commerciale. La copie peut également avoir pour objectif de s'attribuer le travail d'autrui dans un contexte académique. Si l'on élargit ce problème à la copie de tout texte en langue naturelle, le plagiat de publications de recherche ou de rapports d'étudiants est un phénomène réel mais difficilement quantifiable. Si l'échange d'idées ou l'emprunt de citations est incontournable et même souhaitable, celles-ci doivent être accompagnées de leur source ce qui n'est pas toujours le cas. Nous pouvons, pour relever ces emprunts, rechercher des chaînes de caractères exactement répétées sur un texte en langue naturelle comme le proposent certains sites web disposant de banques de textes ou utilisant

des index de moteurs de recherche généralistes sur le web. Lorsque des synonymes ou paraphrases sont utilisés pour masquer le plagiat, la tâche s'avère plus ardue. Dans le cadre de code source, des cas de plagiat entre projets assignés à des étudiants peuvent impliquer des méthodes d'obfuscation plus ou moins sophistiquées alors qu'une copie légitime intra-projet n'entraîne généralement que des modifications mineures.

Il est nécessaire de souligner que l'utilisation illégitime de code source ou de texte en langue naturelle peut prendre de multiples formes. Lorsque le sujet proposé a déjà été couramment traité, il peut être possible d'emprunter des morceaux de code sous licence Open Source librement disponibles. Reproposer des énoncés similaires sur plusieurs années peut permettre une copie sur des projets d'étudiants d'années antérieures. Enfin, il est possible que sur une même promotion, plusieurs rendus supposés indépendants contiennent des morceaux clonés. Idéalement, ces pratiques pourraient être évitées en proposant aux étudiants des sujets inédits et différents pour chacun d'eux. Toutefois, même si cette condition est remplie, le recours à des tiers pour la réalisation des projets est possible et reste indétectable par un outil de recherche de similarité si le travail réalisé est original. Seule une hypothétique analyse du texte [128] ou du code source [131, 129] visant à caractériser par des métriques le style d'écriture de l'auteur pourrait permettre de suspecter l'usage d'un tiers. Toutefois si les métriques étudiées sont connues, celles-ci peuvent être aisément manipulées [130].

Dans le cadre de notre travail, nous nous intéresserons prioritairement à la recherche de similarité dans du code source lorsque des tentatives d'obfuscation sont mises en œuvre. L'analyse de traces d'exécution de programmes pour différentes entrées peut apporter de précieuses informations. Ce vaste champ d'exploration qu'est l'analyse dynamique ne sera cependant pas abordé dans le cadre de notre travail. Nous nous limiterons à l'étude de méthodes de recherche purement statiques où seules la syntaxe du code source agrémenté éventuellement de notions sémantiques est considérée.

L'obfuscation peut être protéiforme : nous discuterons de certaines de ces techniques au chapitre 4. Elles peuvent être automatisées par la conception d'obfuscateurs. Toutefois idéalement l'efficacité d'un outil de recherche de similarité ne saurait reposer sur une sécurité uniquement basée sur l'obscurité : la connaissance des méthodes contre-obfuscatoires employées peut amener un éventuel plagiaire à améliorer le camouflage de sa copie. L'objectif ne reste cependant pas d'empêcher la copie et son obfuscation mais de la rendre désavantageuse par un coût prohibitif, un coût qui serait idéalement équivalent à celui de comprendre le code original et de le réécrire *ex-nihilo*. La mise au point d'outils de recherche de code copié et obfusqué permet aussi, *a fortiori*, d'aider à la recherche de clones intra-projet présentant de nombreuses opérations d'édition, dans une optique de refactorisation.

Structure du document

Nous évoquons maintenant la structure de ce document avec les différentes contributions nouvelles apportées. Nous nous sommes attachés à adopter une démarche progressive et thématique plutôt que de segmenter méthodes existantes et approches nouvelles. Une lecture linéaire est donc adaptée même si certaines parties nécessitent d'inévitables références vers des sections ultérieures. Ainsi par exemple les chapitres 12 sur les métriques directes et consolidées et 13

sur le groupement de correspondances ou d'unités structurelles abordent des notions utiles à la compréhension du chapitre antérieur 7 consacré à la méthode de factorisation.

Première partie Dans une première partie, nous nous intéressons à la définition de la problématique de la recherche de similarité sur du code source.

Nous examinons tout d'abord dans le chapitre 1 les principales applications liées à la recherche de similarité dans du code ainsi que les méthodes existantes couramment utilisées.

Il est ensuite nécessaire de s'interroger sur une définition phénoménologique des clones dans le chapitre 2 avec quelques mesures quantitatives permettant de les évaluer.

L'utilisation de chaînes de caractères brutes est inadaptée pour la recherche de similarité en présence de la moindre opération d'édition : nous examinons ainsi dans le chapitre 3 des représentations du code source permettant de limiter l'influence de certaines modifications sur le code source original. Nous discutons en particulier de la représentation par chaînes de lexèmes ainsi que par arbres de syntaxe sur lesquels nous rappellerons et développerons des méthodes de recherche de similarité. Ces représentations peuvent être enrichies par la donnée des graphes d'appels des projets manipulés. Nous évoquerons également succinctement la recherche de zones de code similaires par la détermination de sous-graphes de dépendances isomorphes.

Le chapitre 4 de cette première partie est lui consacré à l'étude de quelques opérations de modification de code potentiellement utilisables par des obfuscateurs. Parmi ces opérations, celles qui retiendront le plus notre attention seront les modifications de code par ajout ou suppression de code source inutile, réécriture d'expression ainsi que le développement ou la factorisation de fonctions par externalisation. Nous ne nous attarderons pas sur les méthodes d'obfuscation faisant appel à des capacités dynamiques des langages sortant de notre champ d'étude restreint à l'analyse statique du code.

Deuxième partie La seconde partie s'intéresse aux méthodes de recherche de similarité utilisant des chaînes de lexèmes qui furent historiquement les premières à être utilisées pour l'implantation d'outils efficaces de recherche de clones, si l'on exclut toutefois les outils s'appuyant sur des métriques vectorielles.

Le chapitre 5 rappelle essentiellement des méthodes d'alignement globales et locales connues sur des chaînes de lexèmes. Ces méthodes s'appuient sur des techniques de programmation dynamique : elles permettent le calcul de distances d'édition entre chaînes de lexèmes et peuvent être étendues à la comparaison d'arbres de syntaxe. Leur complexité fondamentalement quadratique en nombre de lexèmes manipulés les pénalise dans un usage de comparaison d'un grand nombre de projets ou de projets conséquents : elle les restreint à la comparaison, en seconde intention, d'unités structurelles déjà présumées similaires.

Ensuite, le chapitre 6 aborde la mise en correspondance de facteurs répétés au sein de chaînes de lexèmes à l'aide de structures d'indexation de suffixes. Ces structures, que sont les arbres ou tables de suffixes ainsi que des structures hybrides, ordonnent les suffixes d'une chaîne de lexèmes par ordre lexicographique. Elles peuvent être construites en temps linéaire de la longueur des chaînes manipulées et permettent aisément la détermination de facteurs répétés sur celles-ci. Nous nous intéressons plus particulièrement à la table de suffixes, structure conceptuellement et mémoriellement plus légère que l'arbre de suffixes mais autorisant des opérations similaires à l'aide de l'adjonction de structures connexes telles que l'arbre des intervalles. L'objectif sous-jacent est de pouvoir déterminer l'ensemble des facteurs répétés d'un jeu de chaînes de lexèmes avec un coût temporel et mémoriel optimal. Les facteurs répétés trouvés peuvent être utilisés directement pour le report de correspondances dans le code source ou bien servir de base à la méthode de factorisation qui sera développée dans le chapitre 7.

Le chapitre 7 est consacré à l'introduction d'une méthode de factorisation multi-itérative de chaînes de lexèmes. Son objectif est d'extraire des facteurs communs répétés d'un ensemble de fonctions de chaînes de lexèmes issues de la lexémisation de projets. Nous obtenons alors un graphe d'appels factorisé sur le jeu de projets considéré : ce graphe peut être utilisé pour la visualisation de similitudes ou pour le calcul de métriques de similarité sur les fonctions prenant en considération non seulement le corps de la fonction mais aussi l'ensemble du code qu'elle couvre. Cette méthode nouvelle est relativement insensible aux opérations de factorisation ou développement de fonctions.

Le dernier chapitre de la partie (chapitre 8) explore l'utilisation de la méta-lexémisation de chaînes de lexèmes. Celle-ci consiste à manipuler des k-grams (avec $k > 1$) de lexèmes consécutifs au lieu de 1-grams. L'alphabet des méta-lexèmes manipulés est donc plus large que l'alphabet initial mais la fréquence de chaque méta-lexème est diminuée. Ceci permet l'indexation de k-grams via des empreintes sélectionnées dans des bases afin de permettre la recherche de similitudes portant au moins sur k lexèmes. Dans un second temps, nous abordons la recherche de correspondances non-chevauchantes de facteurs de lexèmes exactement similaires via l'algorithme Running Karp-Rabin Greedy String Tiling utilisant des méta-lexèmes de taille variable. Nous discutons de quelques pistes nouvelles afin d'améliorer la complexité temporelle de cet algorithme.

Troisième partie La troisième partie est consacrée à l'examen de méthodes de recherche de similarité utilisant une représentation par arbres de syntaxes, plus riche qu'une modélisation du code source par séquences de lexèmes où l'information sur les frontières des unités structurelles n'est pas exploitable. Une telle représentation permet par ailleurs l'usage de techniques de normalisation ainsi que d'abstraction éliminant directement les effets de certaines modifications simples. Une analyse statique plus approfondie à un niveau sémantique autorise également la localisation de portions de code trivialement non couvertes qui peuvent être éliminées.

Afin de déterminer des classes d'équivalence de sous-arbres similaires sur un ensemble d'arbres, attribuer une empreinte à chaque sous-arbre via une fonction de hachage se révèle incontournable. Dans le chapitre 9, nous nous intéressons à certaines fonctions de hachage sur

l'espace des sous-arbres. Nous étudions des méthodes de hachage faisant appel à des fonctions de hachage cryptographique ainsi qu'à des fonctions polynomiales de type Karp-Rabin sur des représentations sérialisées des arbres avec pour principal objectif la réduction des collisions entre sous-arbres différents de même valeur de hachage. Certaines stratégies d'abstraction sur les arbres de syntaxe (confusion de types, suppressions de noeuds, ignorance de certains sous-arbres) sont introduites et passées en revue.

À l'aide des fonctions de hachage examinées, nous générons des empreintes et indexons les sous-arbres selon différents profils d'abstraction dans le chapitre 10. Contrairement aux approches existantes, cette indexation nous permet ensuite de rechercher des sous-arbres identiques à un sous-arbre requête selon un profil d'abstraction adapté. Nous utilisons également des structures d'indexation de suffixes afin de déterminer, sur un arbre requête, des chaînes de sous-arbres frères similaires sur la base d'indexation. Lorsqu'un certain niveau d'abstraction a été utilisé, nous pouvons déterminer une métrique d'exactitude entre les résultats obtenus sur la base et les composantes de l'arbre requête correspondantes.

Étant données des correspondances identifiées par des valeurs de hachage identiques, il peut être intéressant de les regrouper par proximité dans leur arbre de syntaxe afin d'obtenir un nombre réduit de correspondances approximatives portant sur un volume de code plus important. Nous présentons dans le chapitre 11 une heuristique de consolidation de correspondances utilisant les arbres de syntaxe mais aussi les graphes d'appel afin d'identifier des fonctions présentant une couverture de code commune.

Quatrième partie Quelques problématiques connexes intervenant en amont ou en aval de la recherche de correspondances sont présentées dans la quatrième partie.

Nous discutons au chapitre 12 de métriques de similarité et de distance pour des correspondances ou des unités structurelles. Ces métriques peuvent servir de base au regroupement de correspondances ou unités tel que présenté dans le chapitre suivant. Elles permettent à un évaluateur humain de prioriser les paires d'unités à examiner. Nous présentons des métriques directes sur des chaînes de lexèmes utilisant des structures d'indexation de suffixes afin d'établir une matrice de similarité à partir d'un jeu de chaînes en bénéficiant d'une complexité temporelle plus favorable qu'une comparaison paire par paire. Nous examinons également différentes méthodes afin d'estimer la quantité d'information partagée entre deux unités structurelles établissant une métrique de similarité entre unités.

Le chapitre 13 est le prolongement logique du chapitre précédent : nous cherchons à regrouper correspondances et unités directement par l'utilisation de matrices de similarité ou par anti-unification.

Enfin nous nous intéressons finalement au chapitre 14 aux techniques de visualisation les plus communes permettant l'obtention d'une vue globale et conviviale des similarités entre différents projets ainsi qu'une vue plus localisée sur certaines zones semblables.

Annexes L'outil Plade réalisé dans le cadre de notre travail et implantant différentes méthodes de recherche de similarité est brièvement décrit en annexe. Une liste (non exhaustive) d'outils de recherche de similarité publiquement disponibles (en service web fermé, en distribution binaire fermée ou Open Source) est également mentionnée. Ces implantations pourront servir de base à la réalisation ultérieure de comparatifs plus approfondis.

Première partie

Problématiques et définitions

1

Applications et principales méthodes de la recherche de similitudes

Sommaire

Nous présentons ici les problématiques et applications majeures de la recherche statique de similitudes sur du code source. La recherche de correspondances ainsi que la mise au point de métriques de comparaison entre unités de compilation trouvent leurs applications dans des domaines tels que la recherche de plagiat sur des logiciels commerciaux, l'évaluation de projets d'étudiants ou la factorisation de code intra-projet pour faciliter la maintenance. Nous évoquons également quelques méthodes généralement utilisées pour ces différentes applications ; certaines d'entre-elles seront plus précisément décrites dans les parties II et III consacrées à la recherche de similarité sur du code source respectivement représenté par une séquence de lexèmes ou un arbre syntaxique.

1.1 Recherche de clones intra-projet

1.1.1 Factorisation par élimination des clones

Un bon style de programmation requiert d'éviter dans la mesure du possible l'existence de code dupliqué au sein de projets dans une vision à long terme. En effet, la présence de clones entraîne une augmentation du volume du projet et nécessite ainsi un coût de maintenance plus important. Elle occasionne des problèmes potentiels concernant la fiabilité et la sécurité du projet puisqu'un problème éventuel survenant sur un exemplaire d'une portion de code dupliqué donnant lieu à un correctif pourra ne pas être examiné sur d'autres exemplaires copiés. L'introduction de code dupliqué peut toutefois présenter un intérêt à court terme [99] (par exemple dans le cadre de variations expérimentales de code avec amélioration de performance, d'adaptation rapide à différentes architectures matérielles...). Si une bonne discipline du programmeur permet d'éviter les clones les plus importants, il demeure possible, sur des projets réalisés par de multiples protagonistes, de noter l'apparition de clones accidentels, i.e. des portions de code dont le rôle est identique avec une implantation similaire. Il est alors souhaitable d'organiser et de factoriser ces morceaux de code au sein de bibliothèques.

Il est ainsi nécessaire dans un premier temps de parvenir à la détection des portions de code dupliquées. En règle générale, les clones sont générés par copier-coller suivi de quelques éditions afin de pouvoir les adapter à un nouveau contexte (modification de types de variables, ajout ou suppression d'instructions, réécriture de certaines expressions, ...). Si la détection de clones exacts est relativement aisée pour une représentation donnée du code, la recherche de correspondances approchées nécessite l'utilisation d'algorithmes de complexité plus importante ou alors l'emploi de certaines heuristiques.

Les clones (qu'ils soient exacts ou approchés) reportés, il demeure important de s'interroger sur leur pertinence dans une optique de factorisation de code. Dans certains cas, il est possible d'automatiser la factorisation de clones exacts par la création de nouvelle fonction et le remplacement de chaque exemplaire du clone par une appel à cette fonction. Cette opération peut potentiellement occasionner une dégradation du temps d'exécution liée au surcoût des appels de fonction. Néanmoins, il est à noter que si le code est compilé, le code objet généré peut remplacer les sites d'appel par un développement du corps de la fonction à des fins d'optimisation.

Dans un contexte général, l'automatisation de la factorisation de clones approchés n'est pas possible car nécessitant un degré de compréhension sémantique. Il est possible qu'un clone reporté ne soit pas pertinent selon le degré d'abstraction utilisé pour la recherche de similarité. Dans le cas contraire, le clone peut faire l'objet d'une élimination par factorisation mais peut nécessiter une compréhension sémantique humaine pour la mise au point des fonctions partagées. Un outil de détection de code similaire apparaît alors comme une aide au repérage de zones pouvant potentiellement être concernées par une procédure de factorisation.

1.1.2 Recherche de patrons de conception

Au delà de la recherche de clones, il peut être également intéressant, pour une meilleure compréhension du code de rechercher des patrons de conception similaires au sein d'un même

projet. Il s'agit alors de détecter et classifier des similarités structurelles [62, 60] entre unités de compilation, classes (pour les langages orientés objet) ou paquetages avec une possible prise en compte de métriques liées à l'interdépendance et à la spécialisation des classes et unités manipulées [59]. Cette opération peut conduire à la mise au point d'outils de génération automatique de code afin de pouvoir représenter sous une forme plus compacte certaines structures suivant un schéma de conception particulier. L'objectif reste, là encore, de réduire la quantité de code et d'économiser le temps de développement futur par formalisation de certains patrons de conception.

1.1.3 Identification d'aspects

Une bonne organisation modulaire du code participe à la limitation de redondances de code au sein d'un projet. Il demeure néanmoins difficile d'isoler des portions de code liées à des préoccupations transversales de forte ubiquité dans les différents modules (e.g. : gestion d'erreurs, journalisation d'événements, gestion d'autorisations...). Il en émerge des morceaux de code idiomatiques présents partagés en de nombreux exemplaires à travers les modules, avec plus ou moins d'opérations d'éditions, pouvant être potentiellement recherchés par des outils de recherche de similitude [97] lorsque leur localisation n'est pas formalisée par l'usage d'un langage orienté aspect.

1.1.4 Modélisation d'évolution d'un projet

Une autre motivation pouvant conduire à la recherche de similarité au sein d'un même projet est le suivi d'évolution d'un projet entre plusieurs versions [80]. En effet, l'évolution d'un projet n'est pas nécessairement modélisée par une succession linéaire de versions : l'apparition de plusieurs branches d'évolution indépendante est également possible [98]. Dans ces conditions, le suivi des éditions réalisées entre des portions de code similaires entre les versions d'une même branche ou de branches différentes demeure indispensable. Outre la modélisation des évolutions destinée à être lisible par un humain, il permet la conception de formats de stockage pour limiter la taille de référentiels de code source. Les méthodes d'analyse de similarités et de différences entre sources peuvent être d'application globale par l'analyse en une unique passe d'un arbre de versions d'un projet, ou bien être incrémentales par la mise à jour de structures de données à chaque nouvelle version ajoutée.

Une méthode classique de suivi d'évolution des sources d'un projet consiste à déterminer les différences entre versions successives d'une même branche d'un projet. Cette opération est généralement réalisée par l'utilisation d'une méthode d'alignement globale (cf section 5.3 sur les lignes entre unités de compilation de localisation identique). Cette méthode incrémentale est utilisé par la majorité des gestionnaires de version centralisés tels que CVS [134] ou Subversion [137], ou décentralisés tel que Mercurial [135] afin de stocker une représentation compacte des différences entre versions successives. Elle présente de nombreuses limitations dans la mesure où ne considérant pas la syntaxe des sources, elle est sensible à des opérations de reformatage simples. D'autre part, le code dupliqué totalement ou partiellement entre unités de compilation de localisation différente n'est pas pris en compte. Enfin, le suivi de clones inter-branches n'est pas réalisé.

1.2 Comparaison de projets

1.2.1 Problématique

Lorsqu'un projet présente une importante suspicion de plagiat sur un autre, il peut être intéressant de comparer ces deux projets pour la recherche d'éventuelles similarités. Cette situation peut notamment intervenir pour deux projets commerciaux ou alors entre projets présentant des licences incompatibles, par exemple entre un logiciel propriétaire et un logiciel de licence Open Source imposant une distribution des œuvres dérivées sous des conditions identiques à l'œuvre originale (telle que la licence GPL par exemple). Contrairement à la recherche de clones intra-projet, il est légitime de s'attendre à la présence de code dupliqué ayant fait l'objet de nombreuses modifications (obfuscations) afin de complexifier le processus de recherche de similarité. Il est nécessaire alors de pouvoir mener une recherche de portions de code approchées avec une bonne résistance aux différents procédés d'obfuscation. On pourra se reporter au chapitre 4 pour la description de certains de ces procédés. Naturellement, l'usage d'un outil de recherche de similarité permet de mettre en évidence des portions de code similaire et de calculer éventuellement des métriques de similitude entre zones de code. Il appartient ensuite à un humain de statuer sur la légitimité des similarités repérées. Pour ce faire, on devra notamment s'interroger sur le niveau de trivialité du code dupliqué ainsi que sur la compatibilité des licences des projets.

1.2.2 Approches envisageables

Nous présentons ici un aperçu de quelques méthodes utilisées pour la recherche de similitudes entre deux projets. Celles-ci seront explicitées plus en détails dans le reste de ce document.

Distance d'édition entre chaînes et arbres

Dans certains cas, la recherche de similitudes entre deux projets peut justifier une complexité mémorielle et temporelle élevée. Il est alors possible de comparer des structures unidimensionnelles (séquences) ou bidimensionnelles (arbres) à l'aide de techniques de programmation dynamique. Ainsi, le code source de chacun des projets peut être représenté sous forme de séquences de lexèmes ; ces séquences étant ensuite comparées pour y déterminer leur distance d'édition par alignement global. Une distance d'édition est ainsi obtenue en déterminant la séquence d'opérations élémentaires de coût minimal permettant de transformer une chaîne en une autre. Il est préférable de recourir à la détermination d'alignements locaux dans la mesure où les zones de similarité ne suivent pas un ordre préétabli : du code similaire peut être déplacé entre les deux projets. Cette méthode sera plus précisément décrite avec le chapitre 5. De façon similaire, des méthodes de programmation dynamique permettent le calcul d'une distance d'édition entre deux arbres ordonnés (cf section 5.5). Pour des projets représentés par des chaînes de n lexèmes, l'alignement global ou local peut être réalisé en temps $O(n^2)$ tandis que la comparaison d'arbres peut être menée en $O(n^3)$ pour des arbres de n nœuds. Ces méthodes permettent la récupération de zones de code de similarités approchées comportant des portions non-similaires.

Sous-graphes homomorphes de graphes de dépendances

Un graphe de dépendances d'un programme permet de modéliser les dépendances entre différentes unités syntaxiques (telles que des instructions). Ainsi, par exemple, la relation de dépendance entre une instruction J dépendant immédiatement sur un chemin d'exécution d'une autre instruction I modifiant par exemple une variable qu'elle utilise est représentée par une arête de J vers I. La recherche de sous-graphes homomorphes de deux graphes de dépendances représentant une paire de projets permet de localiser et quantifier la similarité pour cette paire. L'utilisation de graphe de dépendances permet aisément d'ignorer d'éventuelles zones de code inutile ajoutées ou supprimées entre deux copies ainsi que des morceaux de code transposés et des modifications de structure de contrôle. La recherche de sous-graphes isomorphiques étant dans le cas général un problème NP-complet [11], une recherche de similitudes via cette méthode est temporellement coûteuse, même si des méthodes de filtrage existent pour présélectionner les paires de sous-graphes à comparer.

1.2.3 Recherche de similarité sur un jeu fixe de projets

La comparaison d'un jeu de projets est généralement mise en œuvre pour l'évaluation de projets d'étudiants afin de dépister des cas de plagiat au sein d'une même promotion. Il est nécessaire de pouvoir détecter des paires de zones dupliquées voire des groupes de zones dupliquées avec l'emploi éventuel de moyens d'obfuscation. Nous nous intéresserons à cette problématique dans le chapitre 7 en proposant une méthode permettant la factorisation des graphes d'appels d'un jeu de projets. Nous passons en revue ici d'autres techniques couramment employées.

Utilisation d'une technique de comparaison de paires

La première réponse triviale à la problématique de recherche de similarité sur un jeu de k projets réside dans la comparaison de chaque paire de projets, soit $\frac{k(k-1)}{2}$ comparaisons. Soit $\mathcal{C}(n)$ la complexité moyenne de comparaison d'une paire de projets de taille n, la comparaison du jeu de k projets est donc menée en $O\left(k^2\mathcal{C}(n)\right)$. Cette complexité désavantageuse cache un travail de comparaison redondant dans la mesure où, si l'on considère une relation transitive de similarité entre clones (valide pour des clones exacts), il serait plus rapide de procéder à la récupération de groupes de correspondances plutôt que de paires nécessitant une étape ultérieure de regroupement. Néanmoins, il paraît difficile de définir une relation de similarité approchée transitive (basée par exemple sur une distance d'édition) : dans de tels cas, la comparaison de paires semble, en première approche, incontournable.

Récupération de groupes de clones exacts

La recherche exhaustive de groupes de clones exacts peut être réalisée en temps $O(kn)$ pour un jeu de k projets de taille n à l'aide de structures d'indexation de suffixes telles que l'arbre ou la table de suffixes qui seront présentées au chapitre 6. Les projets étant représentés par des chaînes de lexèmes, ces structures permettent la récupération rapide de tous les groupes de sous-chaînes répétées. En considérant le niveau d'abstraction apporté par la représentation intermédiaire par chaînes de lexèmes, cette méthode ne permet pas la récupération de correspondances approchées avec lexèmes insérés ou supprimés. Il demeure néanmoins possible de

soumettre les paires de projets concernées par de nombreux groupes de correspondances à des méthodes d'alignement local plus coûteuses.

1.2.4 Recherche de similarité sur une base de projets

Propriétés et applications

Il peut être intéressant de rechercher des similarités sur un jeu de projets pouvant être étendu avec le temps. Certaines méthodes de recherche de clones exacts sur jeu de projets peuvent être aisément adaptées à des jeux évolutifs : c'est notamment le cas de la structure d'arbre de suffixes dont les chaînes peuvent être supprimées ou ajoutées sans recalcul total. L'utilisation d'une structure d'indexation incrémentale s'avère ainsi indispensable.

La recherche sur une base de projets présente des applications se recoupant avec toutes celles précédemment décrites. L'utilisation de structures d'indexation incrémentales se prête ainsi à la recherche de code dupliqué en temps réel au sein d'un projet. Elle peut également être utilisée préventivement pour rechercher au sein d'un logiciel des portions de code copiées sous licence incompatible. À cet effet, il pourrait être envisagé de parcourir et indexer l'ensemble des référentiels de logiciels Open Source publiquement accessibles depuis Internet.

Méthodes d'indexation

Les méthodes d'indexation, qu'elles soient menées sur des représentations linéaires ou sous forme d'arbre du code, reposent sur la génération d'empreintes pour des éléments atomiques ou composés de la représentation. La technique d'indexation la plus utilisée consiste à représenter le code sous forme d'une séquence de lexèmes et à générer des empreintes pour chacune des sous-chaînes de k caractères (k étant fixé) appelées k-grams. Nous étudierons au chapitre 9 des méthodes de hachage pour la génération d'empreintes pour des sous-arbres de syntaxe représentant le code. Ces empreintes, liées avec des informations permettant la localisation de la portion de code dont elles sont issues, permettent ensuite l'interrogation d'une base à l'aide d'empreintes issues d'un code source requête.

1.3 Récapitulatif des méthodes

Nous présentons dans le tableau 1.1 un récapitulatif des différentes méthodes de recherche de similitudes. Le lecteur pourra se reporter à l'annexe A pour une classification des principaux outils de détection actuellement disponibles. Dans le cadre de notre travail, nous nous positionnons principalement dans l'optique de la recherche de similitudes sur un jeu fixe de projets (chapitre 7 sur la factorisation) ou sur une base de projets (chapitres 9, 10 et 11).

Méthode	Représentation	Approximation	Groupement	Incrémentalité	Complexité temporelle
Alignement local	Chaînes	Oui (sous-séquences)	Non (paires)	Non	$O(k^2n^2)$
Distance d'édition d'arbres	Arbres ordonnés	Oui	Non	Non	$O(k^2n^3)$
Recherche d'empreintes partagées	Chaînes	Non	Non	Oui	$O(kn \log kn)$
Méta-lexémisation variable	Chaînes	Non	Non (paires)	Non	$O(kn \log^2 kn)$
Recherche de sous-graphes homomorphes	Graphes de dépendances	Non	Non	Non	$O(k^2e^n)$
Factorisation	Chaînes	Oui (insertions/délétions)	Oui	Non	$O(kn)$
Indexation de suffixes	Chaînes	Non (sous-chaînes)	Oui	Possible	$O(kn)$
Indexation d'empreintes d'arbres	Arbres	Non	Oui	Oui	$O(kn \log kn)$

FIG. 1.1 – Récapitulatif des propriétés des méthodes de recherche de similarité

Notes :
- Représentation : forme du code source utilisée par la méthode
- Approximation : étant donnée la représentation, la méthode peut-elle détecter des correspondances approchées ?
- Groupement : la méthode peut-elle directement grouper les correspondances similaires ?
- Incrémentalité : la méthode utilise-t-elle des structures pouvant être mises à jour incrémentalement lors de l'ajout de nouveaux projets ?
- Complexité : complexité temporelle expérimentale moyenne pour la comparaison de k projets de taille $O(n)$ ($O(n)$ lexèmes sur un alphabet Σ, arbre syntaxique de $O(n)$ nœuds ou graphe de dépendances de $O(n)$ nœuds). Certains paramètres algorithmiques ou certaines propriétés du code (non précisés ici à des fins de simplification) peuvent avoir une influence sur la complexité.

2

Origine et évaluation des clones

Sommaire

Les outils de recherche de similarités dans du code source s'intéressent à la récupération de zones de similarité généralement dénommées clones ou correspondances (en anglais *matches*). Il est néanmoins préalablement nécessaire de s'interroger sur une définition de clone afin de pouvoir définir plus formellement le champ d'application de ces outils.

Pour une première approche, un clone de code source est une portion de code présentant un certain degré de similitude avec une autre portion de code, ces deux portions formant une paire de clones. On définit alors un groupe de clones comme un ensemble de portions de code dont chacune des paires présente un certain degré de similarité.

Nous supposons que nous manipulons des langages de programmation impératifs présentant pour unité de base la fonction retournant un résultat (éventuellement vide) pour un paramètre communiqué (n-uplet de valeurs). Ces langages peuvent ensuite définir des éléments structurels englobant les fonctions tels que des classes ou objets, des unités de compilation ou des paquetages. Nous ne nous intéressons donc pas aux langages déclaratifs (tel que XML) ou aux langages logiques (tel que Prolog).

2.1 Taxonomie des clones

Dans cette section, nous classons les clones selon deux critères principaux. Le premier critère porte sur l'origine du clone : s'agit-il d'un clone effectif lié à une opération de copie intra-projet, inter-projets et avec quelle motivation ? Dans un second temps, nous étudions les niveaux des clones selon leur étendue sur le code source. Nous ne nous intéressons pas ici au degré d'approximation des clones qui sera analysé dans le chapitre suivant consacré aux opérations d'édition et d'obfuscation sur les clones.

2.1.1 Clones phénoménologiques

Quels sont les processus à l'origine de l'apparition de clones intra- et inter-projets ? Une première possibilité est liée à un procédé de copie de code par le développeur dont les motivations peuvent être diverses. Au sein d'un même projet, cette opération permet de réutiliser rapidement du code dans un nouveau contexte. Lorsque la copie est réalisée d'un projet extérieur, celle-ci peut être légitime ou non. Dans ce deuxième cas, il s'agit d'une opération de plagiat généralement suivie de procédés d'obfuscation afin de rendre la détection du clone moins aisée. Enfin d'autres clones de niveau structurel (architecture de paquetages ou unités de compilation) peuvent témoigner de l'utilisation de patrons de conception classiques.

Clones intra-projet

La copie de morceaux de code dans l'optique d'une réutilisation dans de nouveaux contextes est un processus (malheureusement) courant chez la plupart des programmeurs. Il apparaît ainsi selon plusieurs études [103, 102, 104] sur certains projets Open Source que leur volume pourrait être réduit jusqu'à 15% par la factorisation de morceaux de code copiés. Un moyen simple de détecter à leur source ces clones phénoménologiques pourrait être de surveiller les opérations de copie-collage de code au sein de l'environnement de développement utilisé [68]. Il serait ainsi possible d'inciter le programmeur à factoriser son code pour éviter toute redondance. Cependant, si dans certains cas la factorisation est possible, dans d'autres contextes celle-ci demeure impossible à cause des limitations du langage utilisé. Ainsi par exemple, un langage tel que Java n'offrant pas de mécanisme de généricité de type primitif nécessite la copie de code avec adaptation de types pour utiliser du code avec des types primitifs différents[1]. Dans certains cas, en programmation objet, l'intégration de plusieurs fonctionnalités déjà implantées peut se heurter à l'absence d'héritage multiple. Une autre situation problématique est rencontrée lorsque le code correspondant à une fonctionnalité est éparpillé au sein d'un projet (aspect). Une solution serait d'étendre le langage pour surmonter ces limitations (introduction du support de composition de caractéristiques [88] ainsi que des aspects) ou

[1] On pourra se reporter à de nombreux exemples du JDK notamment avec les méthodes de la classe Arrays nécessitant la réimplantation de méthodes de tri pour tous les types primitifs.

alors d'utiliser des outils de génération automatique de code (pratique couramment utilisée pour le développement de *Software Product Lines*).

La réduction du volume de code d'un projet par la factorisation de clones permet principalement une diminution de l'effort de maintenance généralement justifiée par un souci temporel et financier. La fiabilité et la sécurité du code produit est également un critère important car un bug causé par un exemplaire de code cloné a une faible probabilité d'être corrigé spontanément sur les autres exemplaires. D'autre part, une réécriture optimisée d'un exemplaire de clone peut ne pas être reportée. Dès lors le choix délibéré d'introduire une duplication de code ne peut être rationnellement motivé que par un gain de productivité à court terme qui surpasserait le gain à long terme d'un code factorisé. Ce choix est notamment réalisé lorsque la production d'un code abstrait nécessite un effort de développement important et/ou lorsque sa compréhension serait difficile. Introduire des clones peut également être utile dans des cas d'optimisation du code à l'exécution ou pour l'adaptation de composantes sur des architectures différentes (tel que par exemple le module multi-thread du serveur web Apache [124]).

Pour notre travail, nous nous intéressons principalement à la recherche de zones de code dupliqué sur un référentiel de code à un instant donné. L'étude de l'évolution de zones dupliquées au cours de la vie d'un projet [98] présente néanmoins un réel intérêt afin de juger du rôle fonctionnel des clones et de la nécessité de refactorisation. Un groupe de clones apparaissant entre deux versions peut être caractérisé, pour les versions suivantes du projet, par divers destins évolutionnistes. Des nouveaux exemplaires de code dupliqué peuvent se greffer au groupe (nouvelles opérations de copie-collage) ou au contraire être supprimés par factorisation. Des exemplaires peuvent faire l'objet de modifications homogènes ou au contraire évoluer indépendamment entre deux versions. Une évolution indépendante peut être la manifestation de corrections de problèmes non propagés par oubli à d'autres exemplaires ou alors plus légitimement le reflet d'une profonde adaptation contextuelle ; la copie initiale fournit alors un code squelette destiné à être remodelé. Une évolution indépendante n'exclut cependant pas une reconvergence ultérieure du code. L'opération de clonage de code est alors un choix de développement qui constitue une solution transitoire choisie dans l'incertitude des fonctionnalités et de l'architecture globale du projet. On notera qu'un des principaux écueils du suivi de groupes de clones entre versions reste de tracer temporellement les morceaux de code similaires.

Clones inter-projets légitimes

L'emprunt de code d'un projet vers un autre permet généralement un gain de temps de développement même si quelquefois certaines adaptations plus ou moins importantes doivent être menées. Cette pratique, pouvant consister à la réutilisation de portions plus [101] ou moins importantes de code (méthodes, classes et unité de compilation) peut être légitime si la licence du projet source du clone l'autorise. C'est le cas notamment des licences Open Source[2] autorisant les œuvres dérivées et donc l'emprunt de code. Cependant une licence Open Source est libre de fixer ses propres conditions quant à la licence de l'œuvre dérivée : ainsi si les licences de type BSD n'imposent aucune condition quant au projet destination du clone (si ce n'est de conserver les informations concernant l'auteur du code original), les licences de type GPL conditionnent la redistribution d'œuvres dérivées sous les mêmes termes. Il convient donc de choisir judicieusement la licence du projet destination. La maintenance du code copié peut

[2] Au sens de l'Open Source Initiative (`http://www.osi.org/`). Ces licences sont également qualifiées de libres.

Licence du projet destination → ↓ Licence du projet source	OS+SA	OS	!OS
OS+SA (ex : (A)GPL, CeCILL ...)	Oui	Non	Non
OS (ex : BSD, Apache, MIT, Artistic ...)	Oui	Oui	Oui
!OS	Non	Non	Non

FIG. 2.1 – Matrice de compatibilité de licences de logiciels

ensuite être menée indépendamment du projet source ou alors les nouvelles versions du projet source peuvent être utilisées avec des risques d'incompatibilité.

Dans le cadre de clones inter-projets légitimes, une adaptation du code au contexte du projet destination peut être attendue mais sans tentatives d'obfuscation des modifications. Dans la plupart des cas, la licence du projet source nécessite de laisser dans le projet destination les informations concernant la paternité du morceau copié.

Clones inter-projets illégitimes

Non respect du droit de copie La réutilisation illégitime de morceaux entre différents projets intervient lorsque les projets source et destination du clone n'ont pas le même titulaire des droits et ne disposent pas de licences compatibles autorisant la copie. Nous pouvons ainsi schématiquement classer les licences généralement utilisées pour des projets informatiques en trois grandes catégories :

1. Licences Open Source permissives (OS). Ces licences n'imposent pas de restriction sur l'utilisation du programme issu du code, imposent la disponibilité du code source sous une forme lisible par un humain avec la possibilité de redistribuer des copies exactes ou dérivées sans restriction de licence sur ses œuvres dérivées.
2. Licences Open Source avec redistribution sous des conditions identiques (OS+SA — *Share Alike*). La redistribution d'œuvres dérivées du projet nécessite d'utiliser une licence avec des conditions identiques de liberté d'utilisation, de disponibilité du code source et de redistribution d'œuvres dérivées.
3. Licences non Open Source (!OS). Elles sont définies négativement comme licences ne satisfaisant pas les libertés d'utilisation, de disponibilité du source et de redistribution des licences Open Source. Ces licences sont généralement qualifiées de privatrices ou propriétaires. Par défaut, un projet ne disposant pas de licence explicite entre dans cette catégorie.

Nous présentons en figure 2.1 une matrice schématisant les cas de légitimité de copie de code selon les types de licence des projets source et destination. Des cas de copie illégitime peuvent survenir entre projets Open Source lorsque le projet source requiert une redistribution sous conditions identiques et pas le projet destination. Des réutilisations illégitimes de code peuvent avoir pour cadre l'échange de code entre projets non-OpenSource et projets Open Source. Concernant l'emprunt de code OS+SA par des projets !OS, l'interrogation de bases indexant des projets Open Source pourrait mettre en évidence ce phénomène. Certaines sociétés [139, 138] se sont spécialisées dans la mise à disposition d'un tel service pour des éditeurs de logiciels commerciaux.

```
1 int c1, c2; int tmp;
  ...
  tmp = c1; c1 = c2; c2 = tmp;
```

(a) Échange de variables *int*

```
1 char a, b; char temp;
  ...
  tmp = a; a = b; b = temp;
```

(b) Échange de variables *char*

FIG. 2.2 – Deux exemplaire de code d'échange de variables en C

Plagiat Un autre cas étranger aux problématiques de droits de copie concerne les cas de plagiat intervenant typiquement au sein de projets réalisés par des étudiants dans le cadre de leurs études. Si une copie volontaire de code entre plusieurs projets apparaît illégitime, il est également possible de noter des clones involontaires liés à la finalité fonctionnelle similaire des projets soumis. Il est également possible d'observer des cas de copie avec des projets extérieurs (projets préexistants ou projet réalisé par un tiers sur demande de l'étudiant) qui peuvent plus ou moins être tolérés en fonction de la taille du code emprunté et du bon signalement de l'auteur original de l'œuvre. Certaines mesures préventives peuvent toutefois être employées pour limiter les cas de code copié en proposant par exemple plusieurs sujets de projet inédits pour lesquels la réutilisation de code existant ainsi que la copie entre étudiants de la même promotion sera difficile.

Obfuscation Dans le contexte de clones illégitimes, il est fortement probable que le plagiaire cherche à rendre la détection de l'emprunt du code plus difficile. Différentes méthodes d'obfuscation peuvent alors être utilisées dont certaines seront examinées dans le chapitre 4. Dans le cadre de notre travail, nous cherchons à permettre la détection de clones en présence d'utilisation de méthodes d'obfuscation en limitant le report de faux-positifs.

2.1.2 Niveaux des clones

Clones locaux

Un clone local concerne une portion de code similaire de taille modeste à l'échelle d'une fonction. La spécificité d'un clone local réside dans le fait que ses sous-éléments (typiquement des instructions) ne sont en règle générale pas commutatifs. Il peut néanmoins être possible de déterminer des dépendances entre instructions afin de segmenter le corps d'une fonction en différents groupements d'instructions indépendants. Nous présentons en figure 2.2 un exemple de deux exemplaires de clones concernant l'échange de valeurs de deux variables avec types différents et renommage d'identificateurs de variable. Ce clone pourrait être refactorisé par l'écriture d'une macro ou d'une fonction prenant pour paramètres les adresses des variables et leur taille mémoire.

Les outils de détection de clones étudiés s'intéressent dans leur majorité à la détection de clones locaux avec une tolérance plus ou moins importante quant aux opérations d'édition entre clones.

Clones structurels

Un clone structurel est un clone de taille plus étendue qu'un clone local : un tel clone peut concerner à différentes échelles plusieurs groupes d'instructions indépendantes d'une fonction,

un ensemble de fonctions, une classe, une unité de compilation, un paquetage voire un projet entier. Ces clones étant de taille importante, ils comportent généralement de nombreuses opérations d'édition. Les sous-éléments d'un groupe de clones structurels peuvent être dans un ordre différent selon les exemplaires. En effet, si l'ordre d'instructions présente une importance à l'échelle d'une fonction, les membres d'une classe ou d'une unité de compilation (variables membres, méthodes...) sont généralement commutatifs. Nous proposons au chapitre 11 une méthode permettant l'agrégation de clones locaux proches afin de créer des clones structurels approximatifs plus étendus : un exemple d'un tel clone (au niveau fonctionnel) révélé par cette méthode est spécifié en figure 2.3.

Clones creux

Contrairement aux clones structurels qui peuvent être décomposés comme une juxtaposition de clones locaux de localisations plus ou moins proches, les clones creux ne présentent comme similarité que la macro-structure du code source. Ainsi, par exemple, deux classes présentant des méthodes de signature équivalente (mêmes types de paramètres) sans que le corps de ces méthodes soit similaire peuvent être considérées comme des clones structurels creux. De même, deux portions de code présentant la même hiérarchie de structures de contrôle mais avec des instructions différentes peuvent être reportées comme des clones locaux creux.

Localiser les clones creux peut être utile dans une optique de réingénierie afin de réorganiser la hiérarchie des classes par la création de nouveaux ancêtres communs lorsque ceci est nécessaire. Ces clones structurels permettent de mettre en lumière des schémas de conception spécifiques et peuvent également permettre de classifier des unités de source de domaines fonctionnels proches mais d'implantations différentes.

La représentation la plus adaptée à l'étude des clones creux est l'arbre de syntaxe du code : on pourra alors les mettre en évidence par la recherche de sous-arbres homomorphes jusqu'à une certaine profondeur ou avec abstraction de certains nœuds.

2.2 Évaluation des clones détectés

Un clone détecté est un clone dont l'existence est découverte par l'utilisation d'un système de détection de clone, qu'il soit automatique ou humain. On remarque que les méthodes algorithmiques nécessitent l'utilisation de paramètres afin d'ajuster le report de clones conduisant à l'obtention de groupes de clones potentiellement différents. Quant à un juge humain, selon certains critères définis (degré de similarité, utilité de la factorisation, ...), il peut rechercher des clones présents au sein de projets avec une part inévitable de subjectivité.

Nous proposons maintenant quelques définitions pouvant servir de base à une approche quantitative et qualitative de mesure d'efficience d'un outil de recherche de clones. Il s'agit d'introduire les deux dimensions que sont l'extensibilité et la peuplabilité d'un groupe de clones et de les relier à la notion de pertinence.

```
72 public class Javac12 extends DefaultCompilerAdapter {
       public boolean execute() throws BuildException {
           attributes.log("Using␣classic␣compiler", Project.MSG_VERBOSE);
75         Commandline cmd = setupJavacCommand();
           OutputStream logstr = new LogOutputStream(attributes, Project.MSG_WARN);
           try {
               // Create an instance of the compiler, redirecting output to
               // the project log
80             Class c = Class.forName("sun.tools.javac.Main");
               Constructor cons = c.getConstructor(new Class[] { OutputStream.class, String.class });
               Object compiler = cons.newInstance(new Object[] { logstr, "javac" });
               // Call the compile() method
               Method compile = c.getMethod("compile", new Class [] { String[].class });
85             Boolean ok = (Boolean)compile.invoke(compiler, new Object[] {cmd.getArguments()});
               return ok.booleanValue(); }
           catch (ClassNotFoundException ex) {
               throw new BuildException("Cannot␣use␣classic␣compiler,␣as␣it␣is␣not␣available"+
                 "␣A␣common␣solution␣is␣to␣set␣the␣environment␣variable"+
90               "␣JAVA_HOME␣to␣your␣jdk␣directory.", location); }
           catch (Exception ex) {
               if (ex instanceof BuildException) {
                   throw (BuildException) ex;
               } else {
95                 throw new BuildException("Error␣starting␣classic␣compiler:␣", ex, location); }
           } finally {
               try {
                   logstr.close();
               } catch (IOException e) {
100                // plain impossible
                   throw new BuildException(e); } } } }
```

(a) ant.taskdefs.compilers.Javac12

```
67 public class KaffeRmic extends DefaultRmicAdapter {
       public boolean execute() throws BuildException {
           getRmic().log("Using␣Kaffe␣rmic", Project.MSG_VERBOSE);
70         Commandline cmd = setupRmicCommand();
           try {
               Class c = Class.forName("kaffe.rmi.rmic.RMIC");
               Constructor cons = c.getConstructor(new Class[] { String[].class });
               Object rmic = cons.newInstance(new Object[] { cmd.getArguments() });
75             Method doRmic = c.getMethod("run", null);
               String str[] = cmd.getArguments();
               Boolean ok = (Boolean)doRmic.invoke(rmic, null);
               return ok.booleanValue();
           } catch (ClassNotFoundException ex) {
80             throw new BuildException("Cannot␣use␣Kaffe␣rmic,␣as␣it␣is␣not␣available"+
                "␣A␣common␣solution␣is␣to␣set␣the␣environment␣variable"+
                "␣JAVA_HOME␣or␣CLASSPATH.", getRmic().getLocation() ); }
           catch (Exception ex) {
               if (ex instanceof BuildException) {
85                 throw (BuildException) ex;
               } else {
                   throw new BuildException("Error␣starting␣Kaffe␣rmic:␣", ex, getRmic().getLocation()); } } } }
```

(b) ant.taskdefs.rmic.KaffeRmic

FIG. 2.3 – Clone structurel intra-projet au niveau fonctionnel présent dans Eclipse-Ant (mis en évidence par consolidation de correspondances en 11.5)

2.2.1 Pertinence d'un groupe de clones

Définition 2.1. Un groupe de clones $c_{1..n} = (c_1, c_2, \cdots, c_n)$ est dit pertinent selon un oracle ψ ssi $\psi(c_{1..n}) =$ vrai. Un oracle d'évaluation de pertinence ψ' est dit plus restrictif que l'oracle ψ ssi pour tout $c_{1..n}$, $\psi'(c_{1..n}) \implies \psi(c_{1..n})$.

L'oracle ψ se présente en pratique généralement sous les traits d'un évaluateur humain qui n'est donc pas déterministe. Une autre alternative, libérée de toute subjectivité humaine, serait d'utiliser une méthode algorithmique déterministe pour juger de la pertinence des clones. Il paraît néanmoins difficile d'établir des critères d'évaluation algorithmique objectifs de clones.

Des oracles d'évaluation de pertinence algorithmiques peuvent être fournis par des outils de recherche automatique de similarité : un groupe de clones est alors pertinent ssi il est mis en évidence par l'outil. Nous définissons ainsi l'oracle ψ_R évaluant un groupe de clones $c_{1..n}$ comme pertinent ssi les exemplaires de clones $c_1, \cdots, c_n$ sont égaux deux à deux selon une représentation R choisie. Il est alors possible de proposer des méthodes algorithmiques pour implanter ψ_R où R peut être une forme lexemisée (recherche de groupes de facteurs exacts sur structures d'indexation de suffixes tel que vu au chapitre 6) ou une représentation par arbres de syntaxe (recherche de sous-arbres égaux par hachage, cf chapitre 10). Afin de limiter le report de clones trop peu volumineux, on pourra après avoir défini une fonction de volume sur clone poser un oracle $\psi_R^{\geq t}$ plus restrictif que l'oracle précédent, ajoutant comme condition à la détermination de pertinence un volume de clone plus grand ou égal à t.

Une opération d'évaluation humaine est quant à elle temporellement très coûteuse et peut présenter l'inconvénient d'être psychologiquement ennuyeuse sur un ensemble de groupes de clones conséquent. Ainsi, lorsque nous cherchons à évaluer la pertinence moyenne d'un ensemble de groupes de clones, nous pouvons ne sélectionner qu'un échantillon statistiquement significatif de groupes qui seront soumis à un oracle humain, en supposant le résultat extrapolable à l'ensemble des clones. Une technique d'échantillonnage a notamment été employée par Bellon [86] pour l'évaluation de différents outils de recherche de clones. [100] présente une expérience d'évaluation de pertinence d'une vingtaine de clones trouvés par un outil automatique de recherche, CCFinder [71], sur le projet Open Source de moteur de base de données PostgreSQL par huit juges humains : moins de la moitié des clones reportés faisaient l'objet d'une évaluation de pertinence (ou non-pertinence) consensuelle partagées au moins par sept juges.

2.2.2 Extensibilité et réductibilité des groupes de clones

Rechercher des similitudes exactes selon une représentation R (implantation de l'oracle ψ_R) présente des limitations lorsque des opérations d'édition sont attendues entre exemplaires de clones. Il peut donc être utile de s'interroger sur la pertinence de l'extension de clones à partir de germes que sont des clones exacts. À cet effet, nous introduisons la notion d'extensibilité et de réductibilité sur un groupe de clones. Intuitivement, il s'agit de déterminer si des morceaux de code participant à un groupe de clones peuvent être étendus tout en gardant le caractère pertinent de ce groupe ou au contraire réduits, si le groupe est non pertinent.

Définition 2.2. Deux morceaux de code c et d s'intersectent ssi c et d partagent un sous-morceau de code commun. c est inclus dans d ssi c et d s'intersectent avec pour morceau

de code commun c. Par extension deux groupes de morceaux de code $C = (c_1, c_2, \cdots, c_n)$ et $C' = (c'_1, c'_2, \cdots, c'_n)$ s'intersectent (resp. C est inclus dans D) ssi pour chaque $i \in [1..n]$, il existe j tel que c_i intersecte c'_j (resp. c_i est inclus dans c'_j). Étant donné un ensemble de groupes de morceaux de code, un morceau de code c est dit *propre* si celui-ci n'est inclus dans aucun autre morceau de code extrait de l'ensemble des groupes.

Ainsi par exemple, les morceaux de code *abc* et *bcd* s'intersectent par *bc* alors que *bcd* est inclus dans *abcde*. Si l'on considère ces quatre morceaux comme représentants de groupes de clones, seul *abcde* est un morceau propre.

Définition 2.3. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ pertinent est extensible ssi il est possible de trouver un groupe de clones pertinents $(c'_1, c'_2, \cdots, c'_n)$ tels que c_1 soit inclus dans c'_1, c_2 soit inclus dans c'_2, ..., c_n soit inclus dans c'_n. Dans le cas contraire, le groupe de clones est dit maximal.

Définition 2.4. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ non pertinent est réductible ssi il est possible de trouver des clones pertinents $(c'_1, c'_2, \cdots, c'_n)$ tels que c'_1 soit inclus dans c_1, c'_2 soit inclus dans c_2, ..., c'_n soit inclus dans c_n.

Il est donc souhaitable de proposer un outil de recherche de clones proposant des groupes pertinents maximaux.

2.2.3 Peuplabilité et dépeuplabilité des groupes de clones

Si les méthodes de recherche de clones approximatifs utilisant des techniques de consolidation s'intéressent uniquement à l'obtention de groupes de clones de cardinalité 2, des méthodes exactes peuvent construire directement des groupes de clones de cardinalité k quelconque sans faire usage d'étapes de regroupement séparées telles qu'envisagées au chapitre 13. Dans cette optique, il peut être intéressant de se demander s'il serait possible d'ajouter un clone à un groupe sans qu'il perde son caractère pertinent ou au contraire si un groupe non pertinent pourrait devenir pertinent grâce à la suppression d'un clone.

Définition 2.5. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ pertinent est peuplable ssi il existe un morceau de code c_{n+1} tel que $(c_1, c_2, \cdots, c_n, c_{n+1})$ soit également un groupe pertinent. Un groupe pertinent non-peuplable est dit complet.

Définition 2.6. Un groupe de clones $c_{1..n} = (c_1, c_2, \cdots, c_n)$ non pertinent est dépeuplable ssi il existe un sous-groupe de $c_{1..n}$ pertinent.

Un groupe peut être peuplable soit par intégration d'un morceau de code n'appartenant à aucun autre groupe, soit par intégration d'un sous-groupe ou d'un groupe entier externe. On peut remarquer que si deux groupes pertinents $c_{1..k}$ et $c'_{1..k'}$ possèdent deux morceaux c_i et $c_{i'}$ tel que $(c_i, c_{i'})$ soit un groupe pertinent, cela ne constitue pas une condition suffisante à la fusion de $c_{1..k}$ et $c_{1..k'}$ en un seul groupe pertinent, la relation de clone n'étant pas toujours transitive. Nous pouvons également noter qu'un même morceau peut appartenir à plusieurs groupes pertinents.

La figure 2.4 présente un groupe de trois morceaux de code lexemisés soit extensibles sur la droite de deux lexèmes, soit peuplable par l'ajout d'un nouveau morceau. Les deux groupes constitués par extension et peuplement peuvent coexister en tant que résultat d'une méthode de recherche de clones. Les trois morceaux du groupe étendu sont qualifiables de propres tandis que le groupe peuplé comporte comme unique morceau propre `int d = a`.

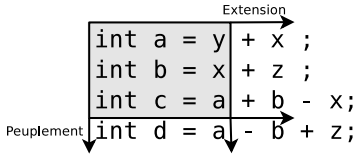


FIG. 2.4 – Peuplement et extension de groupes (représentation lexemisée avec abstraction des identificateurs)

2.2.4 Précision et rappel

Définition 2.7. La précision du résultat (ensemble de groupes de clones reporté) $\mathcal{C}$ notée $\mathcal{A}(\mathcal{C})$ est définie comme le ratio du volume global des clones pertinents détectés sur le volume global des clones détectés.

Définition 2.8. Le rappel du résultat $\mathcal{C}$ noté $\mathcal{R}(\mathcal{C})$ est défini comme le volume global des clones pertinents détectés sur le volume global des clones pertinents.

La difficulté de la détermination de la précision et du rappel d'un résultat est liée à l'obtention de l'ensemble des groupes de clones pertinents. Comme vu précédemment, la pertinence n'étant pas une donnée absolue, un oracle est utilisé qui pourra être un évaluateur humain, un autre outil de recherche voire un procédé de vote par une combinaison de méthodes. Lorsque le coût d'évaluation de pertinence est trop élevé, la précision et le rappel peuvent être statistiquement estimés. La précision d'un sous-ensemble significatif de l'ensemble reporté peut être calculée est extrapolée à l'ensemble total.

En ce qui concerne le rappel, nous pouvons prendre pour base de calcul un sous-ensemble de groupes de clones pertinents. Ainsi, le comparatif de Bellon [86] évalue le rappel en introduisant artificiellement des clones (avec différents degrés d'édition) au sein de projets qui sont ainsi considérés comme pertinents et en déterminant le nombre de clones détectés parmi ces clones. Cette approche permet ainsi de déterminer plusieurs mesures de rappel selon l'oracle d'évaluation de pertinence utilisé qui pourra se limiter à des clones exacts ou à des clones présentant un degré d'édition plus ou moins important. Une tentative de classification des clones est présentée dans le chapitre 4 selon les opérations d'édition qui les différencient.

Les définitions de précision et de rappel introduisent la notion de volume global de clones qui peut intuitivement être résumé comme une mesure de la quantité d'information portée par l'ensemble des morceaux de code participant à des groupes de clones. Cette mesure peut par exemple être réalisée en découpant la représentation du code en atomes élémentaires, chacun associé à un volume, le volume d'un morceau étant la somme des volumes des atomes le constituant.

2.2.5 Pertinence floue

Une limitation des définitions proposées réside dans l'absence de prise en compte de *pertinence floue*. En effet, un groupe de clones non pertinent pouvant être transformé en groupe pertinent par écourtement ou dépeuplement minime peut dégrever la précision et le rappel. En revanche, aussi bien la précision que le rappel sont insensibles à l'existence de groupes pertinents qui pourraient être étendus ou peuplés.

Ainsi, sur la chaîne $u = abcabdab$, le report des groupes de clones $ab : \{(u[1..2], u[4..5], u[7..8])\}$ (groupe pertinent non-prolongeable et non-extensible selon un oracle considérant uniquement pour pertinents des facteurs exacts) ou des groupes $ab : \{(u[1..2], u[4..5]\}$ et $ab : \{(u[4..5], u[7..8]\}$ (groupes peuplables), ou $a : \{u[1], u[4], u[7]\}$ et $b : \{u[2], u[5], u[8]\}$ (groupes extensibles) engendrent la même précision et le même rappel (ici de 1) car tous les lexèmes appartenant à des clones pertinents, et uniquement ceux-ci sont couverts.

2.2.6 Quantification de la similarité d'exemplaires de clones

Définition 2.9. Une métrique de similarité s sur l'espace des exemplaires de clones est une fonction associant à chaque paire de morceaux de code un vecteur reflétant son degré de similarité.

Les morceaux de code d'un groupe de clones ne sont généralement pas des copies exactes de code : des opérations d'édition plus ou moins importantes peuvent être notées, que ce soit pour l'adaptation du clone à un nouveau contexte ou alors pour masquer une copie illégitime. Il est intéressant d'établir une métrique de similarité entre plusieurs morceaux. Celle-ci peut être de nature vectorielle et concerner plusieurs critères : similarité des noms de variables, distance d'édition du code... En règle générale, le vecteur de similarité entre deux morceaux de code est calculé en utilisant la représentation intermédiaire de code spécifique à l'outil de détection employé. Deux familles de représentations seront plus particulièrement étudiées dans ce document : les chaînes de lexèmes ainsi que les arbres de syntaxe.

Exemples de métriques de similarité

Une approche simple employable lorsque le code source est représenté par des chaînes de lexèmes est de calculer une métrique de similarité basée sur la distance d'édition (ou distance de Levenshtein [14]) entre les deux séquences (ce qui est réalisé par l'utilisation d'une technique d'alignement telle que décrite au chapitre 5). Au préalable cela nécessite de définir une fonction de volume associée aux éléments manipulés dans la représentation intermédiaire (ici des séquences de lexèmes). Nous discuterons plus en détail des métriques de similarité au chapitre 12.

Pour une représentation par arbre de syntaxe, il est possible de définir également une distance d'édition basée sur des opérations élémentaires (ajout/suppression de nœud, transformation de nœud) pour transformer un sous-arbre en un autre. Nous évoquons les techniques de programmation dynamique afin de calculer une telle distance en section 5.5. Calculer une distance d'édition entre deux sous-arbres ne paraît judicieux que si ces deux structures présentent préalablement un certain niveau de similarité. Ainsi lorsque nous nous limitons à des

opérations de transformation sur certains types de nœuds ou de sous-arbres, nous pouvons utiliser des méthodes de hachage de sous-arbres selon différents profils d'abstraction comme abordé dans le chapitre 10. Deux sous-arbres identiques selon un profil d'abstraction général peuvent alors être examinés plus en détails avec des profils plus spécialisés pour en déduire une distance d'édition.

* *
*

Nous avons pu au cours de ce chapitre examiner les causes et motivations sous-jacentes à l'apparition de code dupliqué au sein d'un même projet ou dans des projets différents. Cette duplication peut apparaître à l'échelle locale voire à une échelle macroscopique. À l'issue de l'obtention de portions de code dupliquées par un outil spécifique, une des principales difficultés concerne l'évaluation de la pertinence de ces résultats, notion toujours relative au jugement humain ou à la comparaison avec d'autres outils automatiques de recherche. Au-delà d'une simple fonction booléenne, la notion de pertinence devrait également intégrer la possibilité d'étendre, de réduire, de peupler ou de dépeupler un groupe de clones. L'objectif d'une méthode algorithmique de recherche de similarité consiste alors à proposer l'ensemble exhaustif des groupes de clones complets, non-extensibles et pertinents selon le jugement de la majorité de ses utilisateurs humains. Cet objectif peut être atteint lorsque la pertinence est équivalente à l'égalité exacte d'une représentation sous-jacente par chaîne de lexèmes ; nous utilisons à cet effet des structures d'indexation de suffixes dans le chapitre 6. Toutefois lorsque des similarités approchées sont recherchées, la relation de duplication liant les paires d'exemplaires d'un groupe de clones n'est plus transitive. Nous obtenons alors des groupes de deux exemplaires (2-correspondances) pour lesquels la notion de peuplabilité ne fait plus sens : nous ne nous intéressons alors qu'à l'extensibilité. Nous aborderons ainsi au chapitre 11 une méthode de consolidation de clones exacts sur arbre de syntaxe pour laquelle nous nous questionnerons au sujet de la réductibilité et de l'extensibilité des 2-correspondances trouvées. Ce n'est que dans un second temps que nous nous intéresserons à la possibilité de fusionner des groupes de 2-correspondances proches au sein de groupes plus peuplés au chapitre 13.

3

Représentations intermédiaires du code source

Sommaire

La recherche de similitudes sur du code source nécessite au préalable le choix d'une représentation adaptée pour celui-ci. En effet, s'il existe des méthodes utilisant le code source en tant que texte brut, celles-ci se révèlent assez limitées pour des cas d'éditions, même mineures, du code. Nous présentons ici les représentations intermédiaires les plus couramment employées par des outils de recherche de similitudes. Tout d'abord nous évoquons la représentation sous

la forme de séquence de lexèmes consistant à obtenir, par analyse lexicale du source brut, une suite finie de lexèmes. Cette représentation peut ensuite être utilisée en coordination avec des algorithmes classiques de recherche de motifs inspirés du champ de la bioinformatique. Nous traitons également d'un enrichissement des séquences de lexèmes issues d'un projet par le découpage en unités fonctionnelles avec résolution de liens entre les différentes fonctions. Cette représentation sera utilisée par l'algorithme de factorisation de fonctions présenté au chapitre 7. Une autre représentation, plus riche que les séquences de lexèmes, se présente sous la forme d'arbres de syntaxes. Ceux-ci peuvent présenter un niveau d'abstraction et/ou de normalisation plus ou moins avancé afin de minimiser les effets de certaines éditions du code source, qu'elles soient ou non issues d'une volonté obfuscatoire. Tout comme pour les séquences de lexèmes, nous pouvons enrichir les arbres de syntaxe avec des liens d'appels entre fonctions afin de permettre l'extension de similitudes à travers le graphe d'appels d'un projet comme présenté au chapitre 11. Enfin, nous évoquons brièvement la représentation du code source sous la forme de graphe de dépendances : il devient alors possible de détecter des similitudes par la recherche de sous-graphes de dépendance homomorphes. Si cette représentation est relativement avantageuse face à des obfuscations basées sur l'ajout ou la suppression de code inutile, les algorithmes les utilisant sont généralement assez coûteux et prohibitifs pour la recherche de similitudes sur une base de projets. Les représentations utiles citées et les opérations nécessaires pour les obtenir sont résumées par la figure 3.1.

3.1 Code source

Le code source d'un programme est une représentation textuelle humainement intelligible dont la construction obéit à un ensemble de règles définies par un dictionnaire de lexèmes, une grammaire ainsi qu'une sémantique. Après avoir été élaboré par un développeur, le code source fait généralement l'objet de transformations en représentations intermédiaires successives dans l'optique d'obtenir un code objet binaire directement interprétable par la machine. Un code source est généralement différencié d'un texte en langue naturelle par le fait que la grammaire et la sémantique rattachées au langage sont définies formellement, sans aucune possibilité d'ambiguïté. Cependant, l'existence de dictionnaires, tables de lexique-grammaires et graphes d'étiquetage lexical et sémantique encore incomplets pourrait être mis à profit pour la recherche de similitudes sur des langues naturelles.

Un mot sur les formes compilées intermédiaires Nous nous intéressons à la recherche de similitudes au sein du code source original. Dans certains cas, il pourrait être judicieux de travailler sur des formes compilées intermédiaires disposant d'informations de liaison avec le code source original. Ainsi, par exemple, nous pouvons utiliser le code assembleur dérivé d'un code source C ou alors le bytecode obtenu par la compilation d'une unité de compilation Java. Ces représentations intermédiaires peuvent également utiliser des langages de haut niveau (tel que par exemple un compilateur de langage Eiffel [120] produisant des sorties en langage C ou Java). De telles formes compilées intermédiaires présentent l'avantage d'avoir subi certaines opérations de normalisation par le compilateur et peuvent permettre de supprimer certaines formes d'obfuscation du code source tel que l'ajout de code trivialement inutile. Toutefois, certaines formes d'optimisation du code (inlining, déroulage de boucle) peuvent à l'inverse avoir des effets obfuscatoires. Certains compilateurs ont même pour principal objectif l'obfuscation du code produit afin de compliquer les tentatives de rétro-ingénierie. Il paraît donc

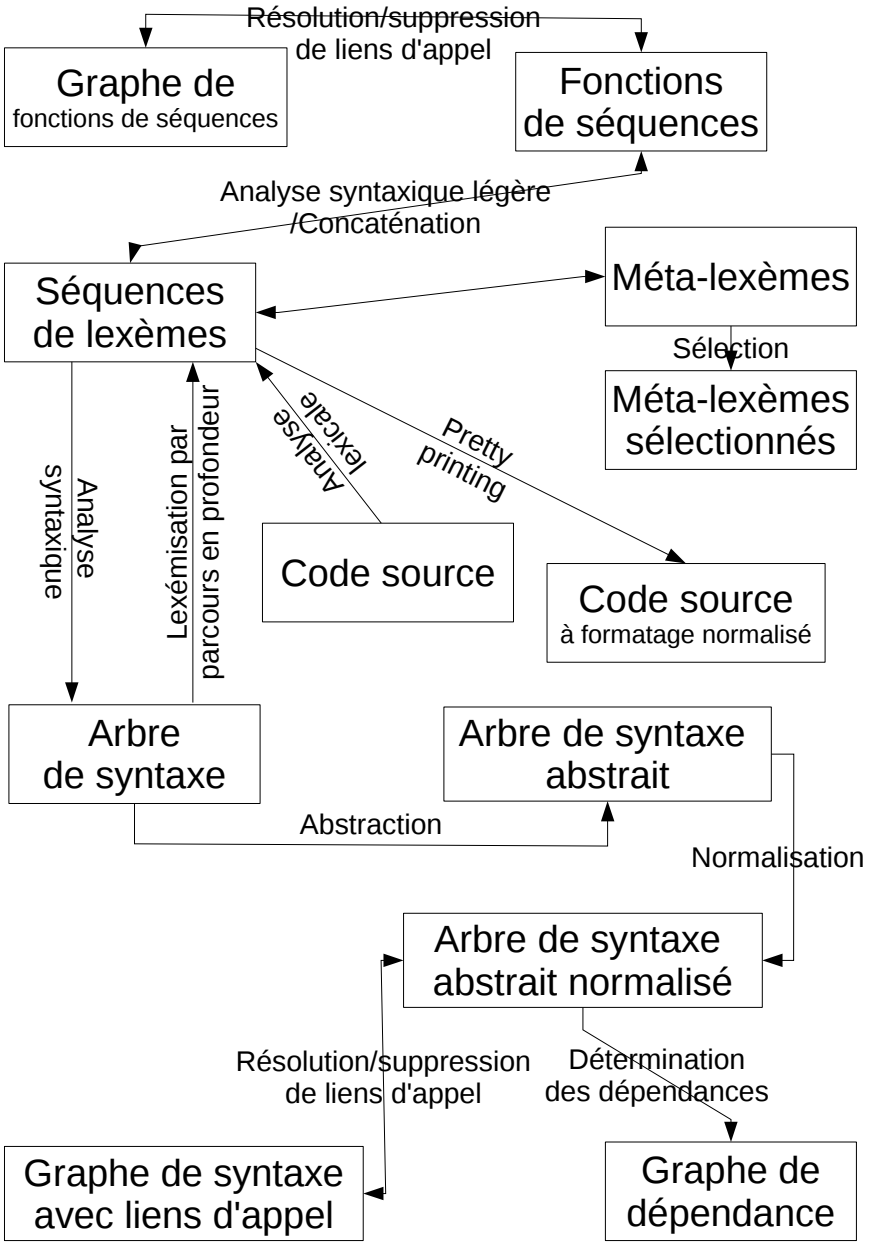


FIG. 3.1 – Représentations et opérations de transformation

/** Calcul de la moyenne */
static int calculeMoyenne(double[] tab) {
double somme = 0.0 ;
for (int i=0 ; i < tab.length ; i++)
 	
somme += tab[i] ;
 	
return somme / tab.length ;
}
	javadoc[Calcul de la moyenne]
	static, int, functionName[calculeMoyenne], lpar, double, lbracket, rbracket, identifier[tab], rpar, lbra
	double, identifier[somme], eq, constant[0.0], semi
	for, lpar, int, identifier[i], eq, constant[0], semi, identifier[i], lt, identifier[tab], dotlength, semi, identifier[i], postinc, rpar
	identifier[somme], assignplus, identifier[tab], lbracket, identifier[i], rbracket, semi
	return, identifier[somme], div, identifier[tab], dotlength, semi
	rbra

Code source	Forme lexemisée

FIG. 3.2 – Un extrait de code source en Java (calcul de la moyenne des termes d'un tableau) et sa forme lexemisée

préférable d'utiliser le code source original et de contrôler les opérations de normalisation sur celui-ci. Lorsque seule une forme compilée intermédiaire est disponible (étude d'un logiciel à source fermée), des opérations de décompilation peuvent être tentées, avec plus ou moins de succès, ou alors des méthodes dynamiques de recherche de similarité sur des flux d'exécutions peuvent être menées. Dans le cadre de notre travail, nous nous intéressons exclusivement à des méthodes de recherche statique de similarité.

3.2 Séquences de lexèmes

3.2.1 Séquence de lexèmes bruts

Étant donnée la définition du lexique d'un langage, l'obtention d'une séquence de lexèmes à partir d'un code source est réalisée par une étape d'analyse lexicale. Généralement les lexèmes d'un langage peuvent être reconnus par des expressions rationnelles et donc par l'usage d'automates finis déterministes (AFD). Les analyseurs lexicaux à base d'AFD peuvent être générés automatiquement à partir d'une description des termes du lexique à l'aide d'outils tels que, par exemple, (F)lex [117] (génération d'analyseurs lexicaux en C) ou J(F)lex [118] (génération en Java). La figure 3.2 présente la forme lexemisée d'une fonction Java.

3.2.2 Abstraction

Les opérations d'abstraction sur les séquences portent principalement sur la suppression des noms d'identificateurs. Il est possible également d'abstraire les noms de types voire également certains types primitifs. Les commentaires peuvent aussi être supprimés. Ces opérations permettent de détecter des similitudes malgré le renommage de variables, le changement de types ou l'édition de commentaires. Toutefois, plutôt que de définir une fonction booléenne de similarité sur des lexèmes, il peut être intéressant de conserver des lexèmes paramétrés par ces informations concrètes afin de définir une distance de similarité plus précise entre lexèmes paramétrés.

fonction	⟶	qualificateur* type identificateur parg (liste-arguments | rien) pard { corps-fonction }
liste-arguments	⟶	argument (, argument)*
corps-fonction	⟶	lexemes-sans-accolades* ({ corps-fonction })* lexemes-sans-accolades*
lexemes-sans-accolades	⟶	tout lexème sauf { et }

FIG. 3.3 – Règles grammaticales de découpage de séquence de lexèmes en unités fonctionnelles

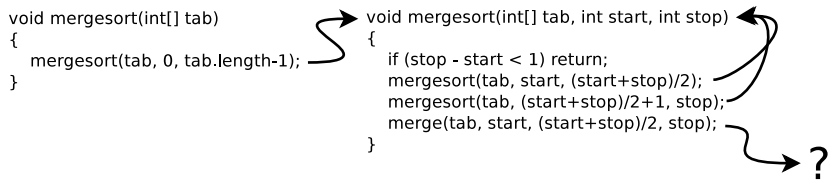


FIG. 3.4 – Un petit exemple de graphe d'appels entre deux fonctions

3.2.3 Fonctions de séquences de lexèmes

Principe

Disposant de la séquence de lexèmes issue d'une unité de compilation, nous introduisons un découpage de cette séquence en unités syntaxiques fonctionnelles. Ce découpage nous permet d'associer à chaque fonction de l'unité de compilation la séquence de lexèmes représentant le code l'implantant. Cette représentation limite la portée des similitudes recherchées à un niveau infra-fonctionnel : nous notons ainsi que des similitudes chevauchant partiellement plusieurs fonctions ne présentent pas d'intérêt tandis que s'il existe des correspondances entre toutes les fonctions de plusieurs classes, celles-ci peuvent ultérieurement être consolidées par une méthode d'extension de clones. En pratique, nous utiliserons cette représentation pour l'exploitation du graphe d'appels du projet analysé : il est donc nécessaire de procéder à la résolution des liens d'appels de fonctions (cf 3.4.2). La factorisation des fonctions (sous la forme de séquences de lexèmes) présentée dans le chapitre 7 permet l'obtention d'un nouveau graphe d'appels de fonctions.

Analyse syntaxique légère

Le découpage d'une séquence de lexèmes en unités fonctionnelles peut aisément être réalisé dans la plupart des langages à l'aide d'une grammaire allégée. Une telle grammaire est indiquée pour les langages C et Java en figure 3.3 : elle peut être utilisée pour traiter le petit exemple indiqué en figure 3.4.

3.3 Arbres de syntaxes

La représentation sous forme de séquence de lexèmes présente des limitations lorsqu'il s'agit d'éviter l'obtention de correspondances chevauchant plusieurs unités syntaxiques. L'arbre de syntaxe d'une unité de compilation est une représentation plus riche représentant sous forme hiérarchique la structure de l'unité telle qu'elle est obtenue par l'application de la grammaire du

langage. Ceci nous permet donc de rechercher des correspondances sur des unités syntaxiques entières : à cet effet nous pouvons utiliser des méthodes d'indexation de sous-arbres basées sur la production de valeur de hachage sur ceux-ci (voir chapitre 9). Il est également possible de quantifier la proximité entre unités syntaxiques de l'arbre afin d'utiliser des méthodes de consolidation et d'extension de correspondances que nous présentons au chapitre 11. La consolidation de correspondances basées sur une représentation par séquences de lexèmes est également réalisable, cependant la proximité de deux correspondances au sein de la séquence de lexèmes ne garantit pas nécessairement la pertinence de leur proximité syntaxique.

3.3.1 Obtention de l'arbre de syntaxe

Arbres de syntaxes concrets et grammaires non-contextuelles L'arbre de syntaxe concret d'une unité de compilation correspond au résultat de l'analyse syntaxique de cette unité en utilisant la grammaire du langage. Les analyses lexicale et syntaxique sur le code source brut peuvent être réalisées en une unique étape (cette approche est préférée par ANTLR [112]) ou plus fréquemment en deux étapes distinctes, le résultat de la lexémisation (la séquence de lexèmes) étant fourni à l'analyseur syntaxique. La plupart des langages de programmation sont conçus pour être syntaxiquement décrits par une grammaire non contextuelle. Une telle grammaire $G = (\Sigma, \Gamma, R, S)$ est décrite par un ensemble de symboles terminaux (Σ) qui représentent les lexèmes du langages, un ensemble de symboles non-terminaux (Γ) qui représentent les différents types de nœuds de l'arbre de syntaxe concret, un ensemble de règles de production R de la forme $U \in \Gamma \longrightarrow V \in (\Sigma \cup \Gamma)^*$ et un symbole non-terminal de départ S de Γ.

Méthodes d'analyse syntaxique L'ensemble des arbres de syntaxe concrets pouvant être obtenus à partir de l'analyse d'une chaîne de n lexèmes peut être calculé avec une complexité temporelle en $O(n^3)$ pour des grammaires non-ambiguës par les méthodes d'analyse de Cocke Younger Kasami [23] et de Earley [7]. Dans la pratique, les langages de programmation peuvent être définis par des grammaires non-contextuelles non-ambiguës afin que chaque séquence de lexèmes ne puisse être interprétée que par un unique arbre de syntaxe concret. Des méthodes d'analyse de complexité temporelle moindre mais ne reconnaissant qu'un sous-ensemble des grammaires non-contextuelles non-ambiguës et nécessitant un pré-traitement par la génération de tables spécifiques sont le plus souvent utilisées pour l'analyse de langages de programmation. Parmi elles, la méthode d'analyse LL(k) permet une analyse descendante (du symbole de départ S aux feuilles terminales de Σ) avec dérivation à gauche avec k lexèmes d'anticipation pour le choix de la règle à développer. La méthode LR(k) et ses variantes (LALR(k)) réalise quant à elle une analyse syntaxique ascendante (des feuilles vers le symbole de départ) avec k lexèmes d'anticipation : elle permet l'analyse des langages non contextuels déterministes. L'analyse LR(k) est généralement privilégiée car reconnaissant une classe plus étendue de langages que l'analyse LL(k) et autorisant une récupération sur erreur plus souple pour les codes sources présentant des erreurs syntaxiques. Une généralisation de l'analyse LR présentant un temps d'exécution plus bas que l'analyse de Earley a été proposée par Tomita [22] pour des grammaires ambiguës.

Quelques générateurs d'analyseurs syntaxiques Si générer manuellement un analyseur syntaxique LL($k \leq 1$) est possible, la complexité des tables d'analyse générées préalablement à une analyse LL(k) et LR(k) pour des valeurs de k élevées nécessite l'utilisation de générateurs

d'analyseurs syntaxiques. Parmi ces générateurs, nous pouvons citer ANTLR [112] créant des analyseurs LL(k) dans de multiples langages et JavaCC [105] générant des analyseurs en Java. Pour l'analyse LALR(1), SableCC [119] (multi-langages), CUP [116] (Java), Tatoo [106] (Java) ou Bison [114] (C) peuvent être utilisés. Ces générateurs proposent chacun leur propre format de définition de grammaire : les règles définies peuvent alors soit être associées à des actions dans le langage cible (Bison, Cup, ANTLR, ...), soit être utilisées pour construire directement un arbre de syntaxe concret qui pourra être ultérieurement manipulé (SableCC, ANTLR, ...).

3.3.2 Abstraction de l'arbre de syntaxe

L'arbre de syntaxe concret issu directement de l'analyse syntaxique par la grammaire du langage présente des nœuds sans réel intérêt sémantique. Par exemple, une liste d'arguments peut être représentée dans l'arbre concret par un arbre binaire de type *peigne*, certains lexèmes ou symboles non-terminaux utilisés pour la désambiguisation de la grammaire peuvent être présents avec des nœuds d'arité sortante unitaire. Par ailleurs, au-delà de cet aspect purement technique, il peut être utile d'introduire une abstraction au niveau de certains nœuds tels que les identificateurs, les commentaires ou certains modificateurs sans intérêt réel pour la recherche de similitudes. L'étape d'abstraction peut être menée directement durant l'analyse en définissant des actions adéquates associées aux règles de la grammaire. Elle peut aussi être réalisée en transformant l'arbre de syntaxe concret par l'utilisation de visiteurs ou la spécification de règles de réécriture dans un langage spécifique (tel que Stratego [121], Tom [122] ou TXL [123]).

Un des objectifs de l'étape d'abstraction pourrait être de définir pour chaque classe de langage de programmation un jeu de nœuds internes et feuilles communs afin de pouvoir exprimer un code source sous une forme abstraite commune. Une telle représentation permettrait de traiter des cas d'obfuscation impliquant une traduction inter-langage du code source.

La figure 3.5 illustre l'analyse lexicale et syntaxique d'une instruction Java simple avec des exemples d'arbre de syntaxe concret et abstrait. On notera notamment la suppression de symboles terminaux et de nœuds d'arité sortante unaire.

3.3.3 Normalisation de l'arbre de syntaxe

L'arbre de syntaxe abstrait obtenu par l'analyse syntaxique du code source peut ensuite subir des opérations de transformations afin d'obtenir une forme normalisée de celui-ci. Une compréhension sémantique plus ou moins avancée du code est alors nécessaire afin de réaliser certaines opérations telles que :

1. La détermination et la suppression des sous-arbres inaccessibles à l'exécution (représentant du code *mort*).
2. La normalisation des expressions.
3. Le découpage de structures composées en structures élémentaires équivalentes.

Par exemple pour l'instruction Java `for (int i=0 ; i < tab.length ; i++) somme += tab[i] ;`, nous pouvons réaliser des opérations de normalisation afin d'obtenir l'arbre de syntaxe abstrait suivant (exprimé sous forme de pseudo-code développé) : Nous notons qu'une instruction analogue utilisant une boucle de type *tant que* (*while*) serait normalisée sous la même forme.

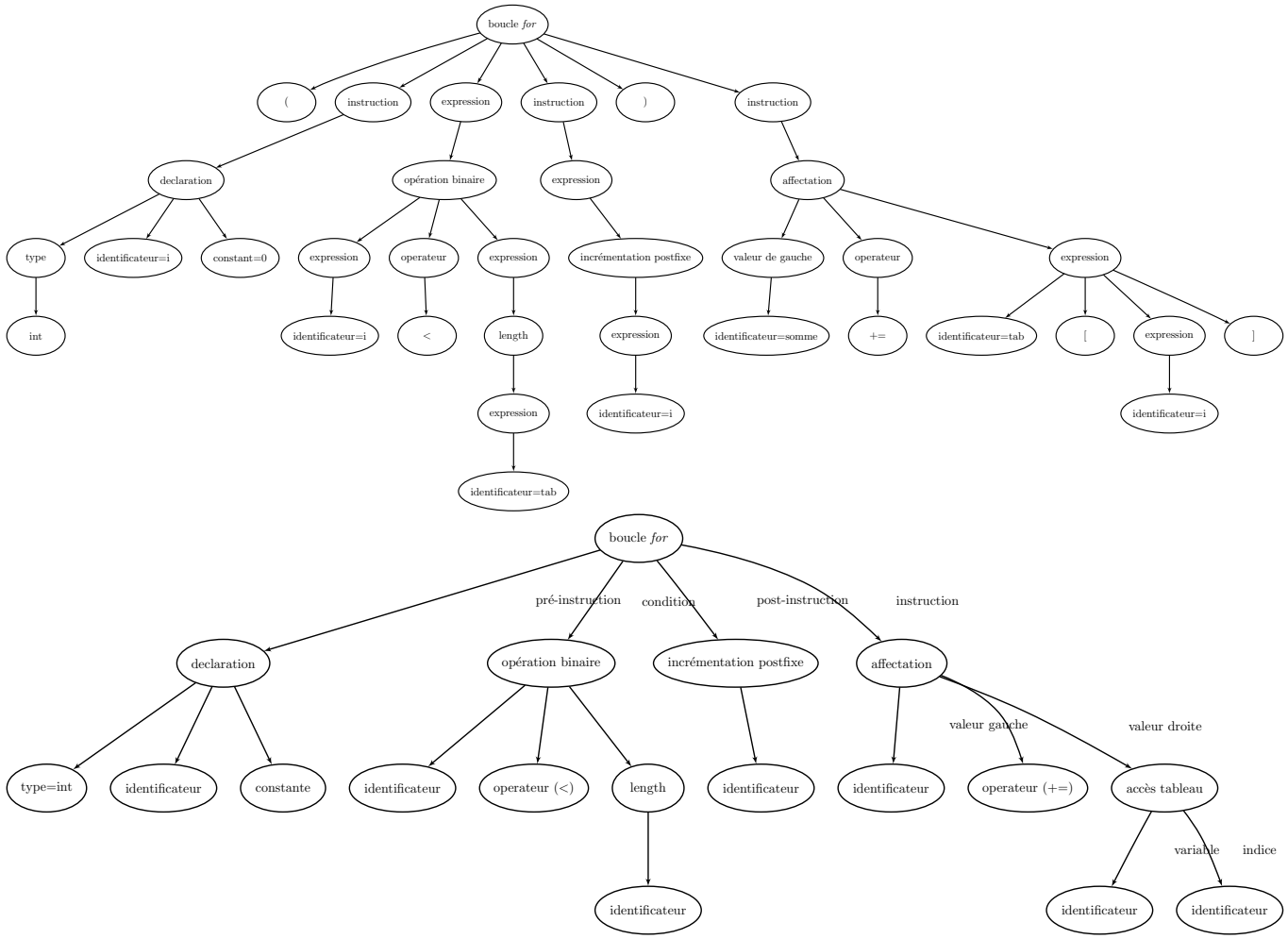


FIG. 3.5 – Un exemple d'arbre de syntaxe concret et abstrait de l'instruction Java `for (int i=0 ; i < tab.length ; i++) somme += tab[i] ;`

```
declaration(entier, i); affectation(i, 0);
boucle(condition = (i < tab.length)) {
affectation(somme, somme + tab[i]); affectation(i, i+1);
}
```

FIG. 3.6 – Pseudo-code développé

3.3.4 Parcours de l'arbre de syntaxe

Le parcours en profondeur de l'arbre de syntaxe concret ou abstrait permet d'obtenir une séquence de lexèmes associée à une structure syntaxique. Le résultat obtenu diffère d'une analyse lexicale directe du code source en fonction du degré d'abstraction et de normalisation préalablement appliqué à l'arbre de syntaxe concret. Ce procédé permet d'intégrer aux séquences de lexèmes des informations de limite syntaxique afin de borner les correspondances. Il devient également possible de limiter la profondeur de parcours de l'arbre de syntaxe afin d'obtenir des séquences de lexèmes dont l'alphabet intègre des symboles non-terminaux abstraits. Ainsi, par exemple, l'instruction Java dont les arbres de syntaxes sont présentés en figure 3.5 pourrait être lexemisée, en se limitant à une profondeur 2 de l'arbre abstrait, en la séquence suivante exprimée en figure 3.7. Certaines modifications telles que la réécriture de l'expression conditionnelle ou de l'affectation sont neutres pour cette représentation.

```
début_for
declaration opération_binaire incrémentation_postfixe affectation
fin_for
```

FIG. 3.7 – Résultat de la lexémisation par parcours en profondeur de l'arbre de syntaxe

3.4 Graphe d'appels

3.4.1 Introduction

L'arbre syntaxique ne comporte que les relations hiérarchiques entre unités structurelles du code source. Afin de pouvoir envisager le support des opérations d'obfuscation par développement et factorisation de fonctions, nous introduisons des liens entre sites d'appels et fonctions appelées par ces sites grâce à la connaissance du graphe d'appels du projet. Les liens d'appels issus du graphe d'appels peuvent enrichir la forêt d'arbres de syntaxe représentant un projet ou bien les fonctions de séquences de lexèmes par l'utilisation de lexèmes d'appel résolus pour la modélisation d'un lien d'appel.

Formellement, le graphe d'appels $\mathcal{G}(p)$ d'un projet p est le graphe ayant pour nœuds les fonctions de p. Il existe une arête $f \rightarrow g$ entre deux fonctions f et g ssi il existe au moins un site d'appel de f appelant g. Nous ajoutons pour chacune des arêtes les lieux des sites d'appel de g (de cardinalité unitaire ou multiple) dans la fonction f.

3.4.2 Résolution des appels de fonctions

Pour les langages procéduraux, fonctionnels ou orientés objet, une fonction peut être généralement définie de façon univoque par la donnée de son nom ainsi que des types de ses paramètres. Certains langages n'autorisent pas le développeur à spécifier le type des paramètres de fonctions : les paramètres peuvent être statiquement non-typés (cas de la majorité des langages de scripts tels que Python, Ruby ou Perl) ou inférés statiquement (cas des langages de programmation fonctionnelle tels que Caml ou Haskell). Lorsqu'une fonction est abritée par un objet ou imbriquée dans une autre fonction, nous pouvons considérer qu'une structure contenant l'état des membres de l'objet ou des variables locales de son environnement est passée en argument supplémentaire à la fonction.

Pour chaque site d'appel d'une fonction, nous cherchons à déterminer la fonction appelée par ce site. Deux possibilités sont alors envisageables : soit la fonction appelée est implantée dans le projet étudié, soit celle-ci en est absente. Cette deuxième possibilité survient pour des appels à des fonctions de bibliothèque externe. Nous nous intéressons exclusivement au premier cas. D'autre part, nous avons pour objectif de réaliser la résolution des appels de fonction uniquement pour les appels dont le nom de la fonction est spécifié statiquement. Ceci nous limite donc aux appels statiquement résolvables. En ce qui concerne les fonctions appelées via des pointeurs, dans le cas général, seule une analyse dynamique permettrait de déterminer un sous-ensemble des fonctions appelées pour chaque site.

Dans le cas le plus simple, le nom de la fonction appelée permet univoquement de déterminer la fonction appelée. C'est le cas notamment des langages procéduraux tel que C ou Pascal où ne peut exister dans un espace de nommage qu'une unique fonction portant un nom donné. Les langages orientés objet autorisent l'existence de fonctions homonymes au sein d'objets distincts. Il devient nécessaire, pour la résolution d'un appel vers une fonction disposant de versions homonymes, de déterminer le type de l'objet concerné : celui-ci, dans le cas général, n'est connu qu'à l'exécution, un même site pouvant appeler des fonctions différentes au cours de celle-ci. Une analyse statique s'avère donc insuffisante. Les langages à multi-méthodes (tels que Common Lisp, C# 4.0 ou Perl 6.0) ne se limitent eux pas uniquement à la détermination du type dynamique de l'objet concerné pour résoudre l'appel : les types dynamiques de tous les paramètres sont considérés.

3.4.3 Graphe d'appels probabiliste

Comme explicité plus tard en 4.9.1, des processus obfuscatoires peuvent ainsi rendre plus délicate voire impossible la résolution des liens d'appel par spécification dynamique de la méthode appelée ou alors par exploitation des mécanismes de surcharge homonyme à types d'arguments différents. Lorsque la résolution statique des liens d'appels est ambiguë, nous pouvons introduire l'usage d'un graphe d'appels probabiliste. Contrairement au graphe d'appels classique, chaque site d'appel est susceptible d'appeler plusieurs fonctions : à chaque lien nous associons une probabilité d'appel $p(\phi \to g)$ (ϕ étant un site d'appel de f). Pour chaque site d'appel appelant les fonctions $g_1, g_2, \cdots, g_k$, nous avons $\sum_{i\in[1..k]} p(\phi \to g_i) = 1$. La somme des probabilités des liens d'appels $p(f \to \cdots)$ depuis f est égale à la cardinalité des sites d'appel. S'il existe au moins un site d'appel depuis f appelant g, une arête $f \to g$ est présente associée à la probabilité normalisée $p(f \to g) = \frac{1}{p(f\to\cdots)} \sum_{i\in[1..l]} p(\phi_i \to g)$ où $\{\phi_i\}_{[1..l]}$ est l'ensemble

des sites d'appels de f appelant g. Un exemple de graphe d'appels probabiliste est présenté en figure 3.8 avec redéfinition d'une méthode.

```
1  public class A {
     public int do() { ... }
3    public int do(int p) { ...}
   }
   public class B extends A {
     public int do() { ... } /* Méthode redéfinie */
   }
8  public class Doer {
     public int action(int p) {
       A doer = DoerFactory.createDoer(p);
       return (p==0)?doer.do():doer.do(p);
     }
13 }
```

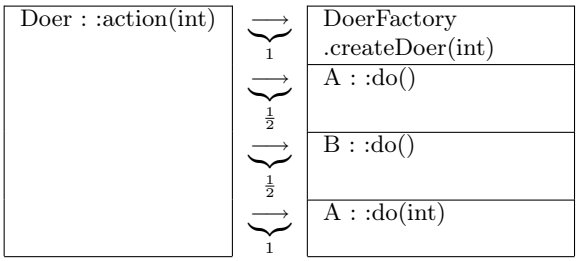


FIG. 3.8 – Code source en Java et graphe d'appels probabiliste associé

3.4.4 DAG d'appels

Utilité

Le graphe d'appels $\mathcal{G}(P)$ d'un projet p n'est pas nécessairement un graphe acyclique (Directed Acyclic Graph : DAG). À des fins de simplification, nous transformons ce graphe en DAG. Cette transformation est réalisée, dans un premier temps, en déterminant les composantes fortement connexes de ce graphe. Pour tout couple de fonctions (f, g) d'une composante fortement connexe, il existe statiquement au moins un chemin d'appel de f vers g ainsi que de g vers f. Chaque composante fortement connexe est considérée comme une méta-fonction représentant l'ensemble des fonctions concrètes qu'elle contient. Une composante fortement connexe de fonctions réalise globalement une tâche algorithmique spécifique : il est rare que plus d'une fonction de cette composante soit appelée depuis une fonction extérieure à la composante. À ce titre, une composante connexe de fonctions peut être, par obfuscation, remplacée par une unique fonction (celle appelée extérieurement) avec l'usage d'une pile pour simuler la pile d'appels.

Le DAG d'appels de projet sera utilisé ultérieurement par la méthode de factorisation de séquences de lexèmes (cf chapitre 7) afin de déterminer une métrique entre deux fonctions d'un même projet ou de projets différents. Il sera également employé afin de réaliser l'extension de correspondances infra-fonctionnelles (cf chapitre 11) par le suivi des liens d'appels entre composantes connexes de fonction.

Volume de couverture des fonctions

Chaque fonction f dispose d'un volume propre $\mathcal{V}(f)$ mesurant la quantité de code qu'elle implante directement (volume de son bloc d'implantation). En première approche, une métrique simple de volume consiste à considérer la longueur de la séquence de lexèmes représentant la fonction ou alors le nombre de nœuds de l'arbre de syntaxe dérivé. Nous définissons, pour chaque fonction f du projet p considéré, l'ensemble des fonctions $\mathcal{R}(f)$ appelées directement

depuis f. Nous en déduisons la clôture transitive de cet ensemble notée $\mathcal{R}^+(f)$: il s'agit de l'ensemble des fonctions qui sont statiquement couvertes par une chaîne de liens d'appel en provenance de f (par convention, $f \in \mathcal{R}^+(f)$). Le volume de couverture de f est alors défini comme la somme des volumes propres de la clôture transitive $\mathcal{R}^+(f)$ des fonctions appelées depuis f (dont f elle-même) :

$$\mathcal{V}^+(f) = \sum_{\phi \in \mathcal{R}^+(f)} \mathcal{V}(\phi)$$

Détermination du DAG d'appels

Deux fonctions f et g appartiennent à la même composante fortement connexe du graphe d'appel ssi g est atteignable depuis f et f depuis g : cela signifie que $g \in \mathcal{R}^+(f)$ et $f \in \mathcal{R}^+(g)$. Comme chaque fonction appartient à sa propre clôture transitive, toutes les fonctions partageant une même clôture transitive d'appels, et uniquement celles-ci, appartiennent à une même composante fortement connexe du graphe. Nous utilisons l'algorithme de Tarjan [21] afin de déterminer, par un parcours en profondeur, les composantes fortement connexes, leurs membres ainsi que le volume de couverture de chacune des composantes. Cela requiert une complexité linéaire en nombre de fonctions et liens d'appel du graphe avec l'usage d'une pile de taille linéaire en nombre de fonctions.

Les fonctions d'une même composante fortement connexe sont représentées par une unique méta-fonction dans le nouveau DAG d'appels. c_i et c_j étant deux composantes fortement connexes, une arête $c_i \rightarrow c_j$ est présente dans le DAG ssi il existe au moins un lien d'appel depuis une fonction de c_i vers une fonction de c_j. Cela signifie également que $\mathcal{R}^+(c_i) \supset \mathcal{R}^+(c_j)$ ($\mathcal{V}(c_i) > \mathcal{V}(c_j)$).

Exemple Un exemple de graphe d'appels comportant 12 fonctions est présenté en figure 3.9. Le volume propre de chacune des fonction est fixé à 1. Nous déterminons le DAG des composantes fortement connexes de ce graphe ainsi que leur volume de couverture.

3.5 Graphes de dépendances

Au-delà des relations purement structurelles modélisées par l'arbre de syntaxe, les relations sémantiques entre différents éléments structurels peuvent être représentées par un graphe de dépendances (*Program Dependency Graph*, PDG). Les relations sémantiques principalement étudiées concernent les dépendances de variables entre instructions : une instruction J dépendant d'une variable v modifiée directement précédemment (dans au moins un chemin d'exécution) par l'instruction I est modélisée par une arête entre J et I. Lorsque des clones creux sont recherchés, il peut être utile d'étudier les relations de dépendances entre macro-structures (dépendances entre unités de compilation, paquetages...).

L'intérêt principal des graphes de dépendances dans la recherche de clones locaux réside dans leur relative insensibilité à des opérations d'édition courantes consistant à ajouter ou supprimer du code source inaccessible ou à manipuler des structures de contrôle. Le problème de recherche de portions de code source similaires peut être traduit dans l'espace des graphes de dépendance en la recherche de sous-graphes homomorphes. Le graphe de dépendances

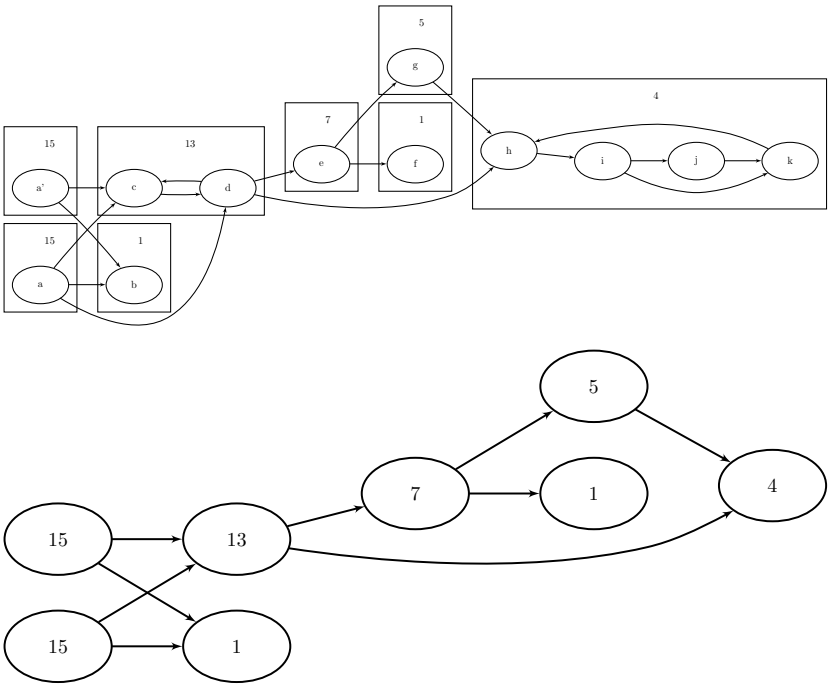


FIG. 3.9 – Un exemple de graphe d'appels et son DAG de composantes fortement connexes étiqueté par leur volume de couverture

peut également être utilisé pour la normalisation d'une séquence de lexèmes ou d'un arbre de syntaxe par la suppression du code trivialement inutile : si l'on considère une fonction retournant une valeur (sans instructions avec effet de bord), seules les instructions couvertes par la clôture transitive du graphe de dépendances à partir des instructions de retour de valeur (considérées comme racines) sont utiles.

Nous présentons un exemple de graphe de dépendances en figure 3.10 sur la fonction de calcul de moyenne précédemment présentée à laquelle nous avons ajouté une instruction inutile de déclaration de variable non utilisée ainsi qu'une boucle réalisant une unique itération. Les nœuds non couverts par la clôture transitive à partir de l'instruction de retour sont représentés par des parallélogrammes non carrés.

```
1 static int calculeMoyenne(double[] tab) {
2  double somme = 0.0;
   for (int k=0; k < 1; k++) { int inutile = 0; for (int i=0; i < tab.length; i++) somme += tab[i];
        }
   return somme / tab.length;
  }
```

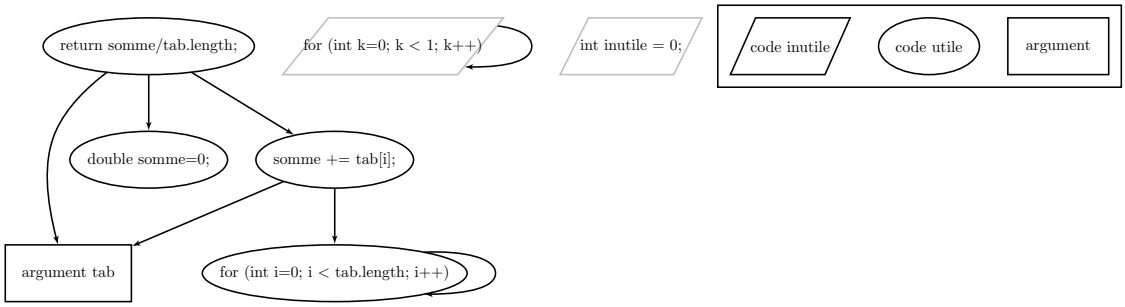


FIG. 3.10 – Fonction et son graphe de dépendances

Originality is the art of concealing your source.

Myself

4

Correspondances approchées et obfuscation

Sommaire

Si les clones exacts sont les plus aisés à mettre en évidence, dans la pratique des opérations d'édition sont généralement réalisées entre plusieurs exemplaires de clones. Ces éditions interviennent aussi bien dans un cadre de copie légitime afin d'adapter le code copié à son nouveau contexte que dans un cadre de copie illégitime où les modifications sont potentiellement plus importantes afin de limiter l'efficacité d'outils de détection de similarité. Nous nous intéressons ici principalement aux modifications liées à une volonté d'obfuscation. Nous présentons les opérations de modification de code source les plus fréquentes accompagnées de quelques idées et pistes pouvant les contrecarrer. Un tableau récapitulatif de ces différentes opérations et leurs caractéristiques principales est présenté en figure 4.3 ; des exemples de code obfusqué sont réunis en figure 4.4 en fin de chapitre.

Obfuscation et... obfuscation Il existe deux types principaux de processus obfuscatoire selon l'objectif recherché. Le premier consiste à introduire des transformations dans le code source afin de rendre celui-ci inintelligible pour un humain : il s'agit de rendre les opérations de rétro-ingénierie plus difficiles. Ce type d'obfuscation peut également aller de pair avec une volonté d'optimiser le code. Il peut se caractériser notamment par la suppression des commentaires, le renommage de variables en noms non signifiants, un usage spécifique d'opérations de préprocesseur (pour les langages en disposant) ou le développement de fonctions ou structures de contrôle permettant une optimisation [107, 109, 113]. La détection d'une telle obfuscation est triviale pour un juge humain ; à première vue l'utilisation de certaines métriques [111] pourrait permettre une détection automatisée. Le second type d'obfuscation, auquel nous nous intéressons, concerne le camouflage d'une opération de copie illégitime de code. L'objectif est alors non pas de rendre le code inintelligible, mais de limiter l'efficacité d'une recherche de similitudes par un humain ou un outil automatisé.

Filigranage de code Le filigranage du code (en anglais *watermarking*) est un procédé de modification du code source afin d'y introduire des marqueurs témoignant de l'origine de celui-ci. Le filigranage permet ainsi de retrouver un morceau de code copié suffisamment significatif sans avoir à disposer d'une base de codes de référence. Un procédé de filigranage idéal devrait permettre la reconnaissance de ces marqueurs malgré des opérations d'obfuscation ultérieure sur le code ne dénaturant pas sa sémantique, avec la connaissance publique de l'algorithme de filigranage et un couple clé privé/clé publique permettant respectivement d'ajouter le filigranage et de vérifier son existence. De nombreux travaux ont été réalisés pour le filigranage de fichiers multimédias [126, 133]. Cependant ces fichiers sont finalement destinés à une interprétation humaine et supportent ainsi des procédés destructifs. Dans le cadre de code sources, la sémantique doit être conservée afin que l'exécution du programme ne soit pas affectée. Ainsi, un filigranage automatisé ne peut être réalisé que par des modifications de formatage ou par des modifications structurelles sémantiquement neutres. Une méthode envisageable serait alors le camouflage d'une chaîne de caractères indiquant l'origine par un procédé stéganographique. Un copieur de code attentif peut toutefois systématiser des opérations de normalisation de code afin d'effacer tout filigranage.

4.1 Modifications lexicalement et syntaxiquement neutres

4.1.1 Modifications de formatage

Les opérations d'édition concernant le formatage du code peuvent intervenir aussi bien dans le cadre de copie légitime qu'illégitime. Elles consistent principalement à ajouter ou supprimer des caractères non-significatifs pour l'analyse lexicale tels que, pour la plupart des langages[1] des retours à la ligne, des tabulations ou des espaces. Si certaines métriques se référant au style de programmation peuvent considérer le formatage (style d'indentation, de retours à la ligne, nombre moyen de caractères par ligne...), aucun outil de recherche de similitude ne les utilise. Cependant les méthodes de recherche de similitudes par recherche purement textuelle sans analyse lexicale préalable sont sensibles à des modifications de formatage.

4.1.2 Édition de commentaires

L'édition de commentaires accompagne généralement des copies illégitimes. Il s'agit soit d'ajouter de nouveaux commentaires, soit (pratique plus fréquente) de supprimer des commentaires ou de les modifier. La suppression peut être réalisée systématiquement ou partiellement à l'aide d'une analyse lexicale. La réécriture de commentaires peut être réalisée manuellement par un humain ou alors automatiquement, par exemple en remplaçant certains termes par des synonymes à l'aide d'un corpus approprié. Aucun outil de recherche de similitudes intègre à notre connaissance une recherche de similarité en langue naturelle sur des commentaires : seul le code source utile pour la compilation est considéré. La suppression de commentaires étant une opération simple, il est possible d'attendre d'un plagiaire qu'elle soit réalisée : cependant si celle-ci n'est pas menée, une similarité sur un commentaire est un indice important sur la similarité du code environnant.

4.1.3 Renommage d'identificateurs

Obfuscation Le renommage d'identificateurs est peu fréquent dans un cadre légitime. Un juge humain chargé de rechercher des similitudes possédant généralement une mémoire assez sensible aux noms de variables utilisées, un plagiaire cherchera souvent à les modifier. Le renommage d'identificateurs est alors réalisable manuellement ou systématisable automatiquement. Si le renommage systématique de variables est sans risque à une échelle locale, cette opération est plus délicate pour des membres (variables, méthodes...) d'accessibilité large d'unité de compilation ou de classes, ceux-ci pouvant être évoqués en interne par des mécanismes dynamiques de réflexion ou par du code extérieur non connu. D'autre part, un obfuscateur automatique devra prendre soin de réaliser le renommage avec des termes humainement réalistes (lettres uniques, conversion des noms entre un style *under_score* et *dromaDaire*...).

Parade Une solution pour la détection de similitudes malgré le renommage consiste à abstraire totalement le nom des identificateurs. Un tel procédé peut toutefois augmenter les risques de report de faux-positifs sur des petites portions de code. Une attitude intermédiaire consiste à attribuer des noms normalisés aux variables identiques d'une expression, instruction ou unité

[1] Pour certains (rares) langages, le formatage du code source possède cependant une signification syntaxique. Appartenant à cette catégorie de langage, nous pouvons citer Ruby, Python ainsi que le (plus exotique) langage Whitespace (`http://compsoc.dur.ac.uk/whitespace/`).

syntaxique plus vaste. Par exemple l'instruction `i = i + j + 1` pourra être considérée comme `ID = ID + ID + const` ou plus précisément `ID1 = ID1 + ID2 + const`.

4.2 Transposition de code

4.2.1 Ordre de sous-unités structurelles

Étant donnée une unité structurelle du code source composée d'une séquence de sous-unités $E = e_1, e_2, \cdots, e_n$ (par exemple le corps d'une fonction composé de n instructions), nous déterminons l'ordre imposé sur chaque paire de sous-éléments (e_i, e_j) : soit aucun ordre n'est imposé (les sous-éléments sont indépendants), soit e_j dépend de e_i et doit donc être placé après e_i, soit il s'agit du contraire. La dépendance entre sous-unités peut être obtenue à l'aide d'un graphe de dépendances tel que décrit en 3.5. Il peut donc exister des possibilités de déplacer des sous-unités de l'unité structurelle tout en conservant les relations d'ordre imposées par les dépendances.

4.2.2 Obfuscation par transposition

Obfuscation Le code source peut être obfusqué par transposition en déterminant les relations d'ordre entre sous-unités et en réalisant des déplacements valides selon ces relations. Cela permet donc de conserver la sémantique du code tout en désorganisant sa structure. Ainsi pour la plupart des langages modernes n'accordant pas d'importance à la position des fonctions déclarées ou implantées, il est possible de modifier l'ordre des fonctions. À l'échelle locale, un graphe de dépendances de variables permet de déterminer les déplacements possibles.

Résistance à l'obfuscation Le graphe de dépendances d'une unité étant insensible aux opérations de déplacement de code sémantiquement neutres, il peut être utilisé pour rechercher des similitudes malgré ce type d'obfuscation par détermination de sous-graphes homomorphes.

Pour les méthodes de recherche de similarité basées sur d'autres représentations, il est nécessaire de considérer le volume de code transposé ainsi que la fréquence de transposition. Dans des cas classiques, le volume d'un morceau de code transposé est assez conséquent pour pouvoir être individuellement détecté, en dehors de son avant et après contexte d'origine, eux aussi assez volumineux pour être détectés. Ainsi, des correspondances sont relevées individuellement : celles-ci peuvent être consolidées, malgré leur désordre en utilisant par exemple une méthode d'extension (cf chapitre 11). Lorsque le morceau dupliqué n'est pas suffisamment volumineux pour être détecté, l'avant et après contexte peuvent être reportés individuellement, ou être consolidés en une unique correspondance en utilisant un algorithme d'alignement local (qui signalera le morceau déplacé en tant que morceau supprimé). Quant aux méthodes d'alignement globales, celles-ci doivent être proscrites si des opérations d'édition par transposition de code volumineux sont attendues.

4.3 Insertion et suppression de code inutile

L'insertion et la suppression de code inutile au sein de code copié limite la détection de similarité par des algorithmes de recherches d'arbres ou séquences de lexèmes exactement identiques. Ainsi des portions intactes sont détectées mais séparées par des portions de code sans correspondance que ce soit pour l'exemplaire original (code supprimé) ou pour l'exemplaire

copié (code ajouté). Néanmoins, des algorithmes d'alignement local de séquences peuvent être utilisés car tolérant des zones non concordantes dans les correspondances. Concernant les méthodes d'indexation de code source par génération et sélection d'empreintes (décrites au chapitre 8), elles imposent un seuil minimal de lexèmes identiques t pour le report possible de similitudes : en automatisant l'insertion de code inutile tous les $t-1$ lexèmes, aucune séquence d'au moins t lexèmes identiques ne peut être détectée. Cependant une telle insertion extensive de code inutile est rapidement remarquable par une analyse humaine du code et induit une inflation conséquente du volume du code.

4.3.1 Ajout de code inutile

Une opération d'obfuscation par ajout de code nécessite d'être neutre pour le résultat de l'exécution du code. Cette neutralité peut être atteinte par deux moyens :

1. L'ajout de code non accessible. Dans ce cas, le code n'est jamais exécuté. Ceci peut être obtenu par l'insertion de code quelconque à l'intérieur d'une structure de contrôle garantissant sa non-exécution (bloc *alors* d'une structure conditionnelle de condition invariablement fausse — ou bloc *sinon* d'une structure de condition vraie —, bloc de gestion d'exception jamais levée, ajout après une instruction de retour de fonction, implantation d'une fonction jamais utilisée...).
2. L'ajout de code accessible mais neutre à l'exécution. Il s'agit alors d'insérer du code ne présentant aucune conséquence sur le résultat de l'exécution mais pouvant potentiellement avoir un impact sur les performances du programme. On pourra par exemple définir des variables non utilisées ou pseudo-modifier le contenu de variables par des opérations neutres.

Dans ces deux cas, l'inutilité du code peut être plus ou moins simple à mettre en évidence. Des méthodes d'analyse statiques des chemins d'exécution de programmes peuvent permettre de détecter des morceaux de code trivialement inaccessibles[2]. De la même façon des instructions neutres à l'exécution peuvent être détectées dans des cas triviaux. En supposant l'inexistence d'effets de bord (par exemple avec un style de programmation fonctionnelle), ceci est réalisable par l'utilisation de graphe de dépendances. Il est donc envisageable pour ces cas de normaliser le code source manipulé par suppression du code détecté inutile puis d'utiliser une méthode de détection arbitraire ou alors d'utiliser directement des techniques de recherche de similitudes basées sur des représentations du code en graphe de dépendances de variables. Il demeure cependant que dans le cas général d'un langage Turing-complet, il est impossible de déterminer statiquement si une fonction retourne un résultat constant quels que soient les paramètres d'exécution. Une technique d'obfuscation pourrait ainsi utiliser une bibliothèque de routines algorithmiques de résultat invariant (fonctions retournant 0 dont le résultat est ajouté à la valeur d'une variable, anti-tautologies pour la condition de structures conditionnelles non exécutées...).

Quelques schémas d'insertion de code Nous définissons des schémas d'insertion de code par trois critères principaux :

[2]Nous pouvons remarquer que certains langages tels que Java imposent une discipline au programmeur en interdisant des cas triviaux de code inaccessible (de tels cas génèrent des erreurs à la compilation) ce qui limite les perspectives d'obfuscation par ajout de code inaccessible.

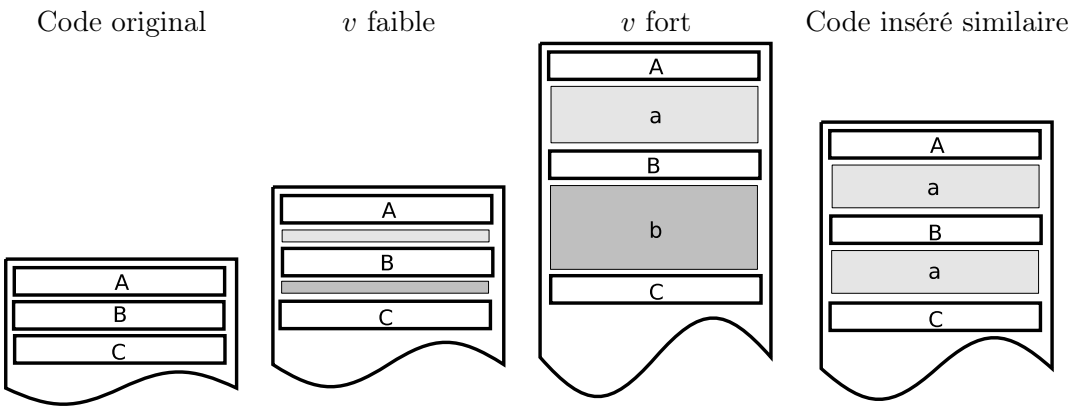


FIG. 4.1 – Quelques schémas singuliers d'insertion de code

1. Le ratio ρ de volume[3] de code inséré par rapport au volume de code original.
2. Le volume unitaire v de chaque morceau de code inséré.
3. Les relations de similarité entre morceaux de code insérés.

En fonction de ces critères, différentes stratégies sont employables pour la recherche de similitudes. Si ρ est important, une grande dissimilarité existe entre la taille du code original et du code copié : une métrique de similarité adaptée quantifiant le volume de l'intersection de code similaire sur le volume de code original doit alors être employée (cf chapitre 12). Si v est petit, les fossés de zones non correspondantes sont faibles et un algorithme d'alignement de séquences (cf chapitre 5) ou de consolidation de correspondances proches (cf chapitre 11) peut être utilisé. En revanche si ρ est élevé et v faible, les systèmes d'indexation et de sélection d'empreintes de séquences peuvent être *aveuglés*. Pour le troisième critère, si les morceaux de codes insérés sont similaires, ils peuvent être détectés comme tels par une recherche de similarité intra-projet : ces morceaux de code peuvent alors se voir attribuer un poids plus faible voire être totalement ignorés dans un processus de recherche de similarité inter-projet.

4.3.2 Suppression de code inutile

La suppression de code inutile présuppose l'existence de tel code au sein de morceaux copiés. Tout comme pour la normalisation de code avec du code inséré, des outils de recherche de couverture de code [115] peuvent être employés. La recherche de code inutile n'est généralement pas menée dans une optique obfuscatoire mais plutôt pour améliorer la qualité du code en recherchant des anomalies potentielles pouvant entraîner des bogues.

4.4 Réécriture d'expressions

La réécriture d'expression peut être utilisée dans un but d'obfuscation. Il s'agit de remplacer une expression par une expression dérivée de valeur identique à l'exécution. Parmi les méthodes de réécriture d'effet neutre, nous pouvons citer non exhaustivement :

[3]Nous définissons le volume d'une portion de code c par une métrique permettant de quantifier la quantité d'information apporté par ce code. Celui-ci peut être exprimé par une mesure théorique telle que la complexité de Kolmogorov du code, ou alors en pratique par le nombre de lignes (code brut), de lexèmes (séquence de lexèmes) ou de nœuds (arbre de syntaxe) utilisés pour la représentation du code.

$$
\begin{array}{lcl}
e & \longrightarrow & e+0 \mid e*1 \mid e-0 \mid e>>0 \mid e<<0 \\
e+f+g & \longrightarrow & (e+f)+g \\
e-f-g & \longrightarrow & e-(f+g) \\
e*(f+g) & \longrightarrow & e*f+e*g \\
 & \dots &
\end{array}
$$

FIG. 4.2 – Quelques règles de réécriture neutres d'arbres d'expressions arithmétiques

- l'usage d'une opération avec une opérande neutre (addition ou soustraction de 0, produit par 1...). L'opérande neutre pourra être masquée en utilisant une fonction de calcul telle que décrite en 4.3.1 ;
- l'usage de la commutativité d'opérateurs en modifiant l'ordre des opérandes ;
- l'usage de propriétés d'associativité et de distributivité de certains opérateurs.

Ces opérations peuvent être systématisées par l'utilisation de règles de réécritures simples[4] sur les arbres syntaxiques d'expressions (quelques règles sont présentées en figure 4.2). Symétriquement, il est possible d'établir une normalisation des arbres syntaxiques d'expressions avant traitement par un détecteur de similitudes afin d'éliminer les effets de cette obfuscation.

4.5 Changements de type

Les changements de types au sein du code copié peuvent intervenir dans une optique de réadaptation de code à un nouveau contexte ou dans un but d'obfuscation. Les types primitifs d'un langage sont souvent hiérarchisés : ainsi en Java un élément de type *byte* peut être représenté par un élément *short*, lui-même représentable par un *int*, représentable par un *long*. Nous pouvons ainsi remplacer le type d'une variable par un *supertype* pouvant le représenter. Concernant des types structurés (structures, classes...), il est possible de réaliser un remplacement en dupliquant une déclaration de type avec un nouveau nom ou alors, pour les langages à classes avec héritage, en remplaçant un type par un type ancêtre (supertype) ou descendant (sous-type). Cette dernière opération peut être accompagnée d'opérations de coercition si nécessaire.

Une solution envisageable afin de résister à ce type d'obfuscation consiste soit à normaliser avec un supertype suffisamment générique, soit à abstraire totalement des types (ce qui revient à la sélection du type racine).

4.6 Modifications de structures de contrôle

La manipulation de structures de contrôle peut être utilisée afin de modifier structurellement le code source avec un impact neutre sur l'exécution. Si les premiers langages de programmation ne comportaient pas de structures de contrôle mais plutôt des étiquettes et instructions de

[4]Il faut toutefois porter une attention particulière aux langages autorisant la surcharge d'un même opérateur pour différents types. Par exemple, l'opérateur * qui serait surchargé en C++ pour implanter la multiplication matricielle ne serait pas commutatif. En Java, l'opérateur + peut être utilisé pour l'addition de nombres mais également pour la concaténation de chaînes où il n'est pas commutatif.

saut conditionnel ou inconditionnel, les langages impératifs récents implantent généralement trois types de structures de contrôle :

1. Des structures conditionnelles permettant d'assortir l'exécution du code à une condition.
2. Des structures de boucle autorisant l'exécution d'une même portion de code un nombre variable de fois.
3. Des structures de capture d'exception permettant de gérer les erreurs inattendues survenant lors de l'exécution d'un bloc de code.

Entourage par une structure d'exécution unitaire Il est aisé d'entourer arbitrairement des groupes d'instructions d'un code source par une structure de contrôle arbitraire d'exécution unitaire. À cet effet, nous pouvons entourer les instructions d'une structure conditionnelle de condition invariablement vraie, d'une structure de boucle exécutée une unique fois via une condition d'arrêt ad-hoc ou alors d'une structure de capture d'exception (non nécessairement levable dans le contexte). Inversement ce procédé d'obfuscation peut également être intéressant afin d'introduire des volumes importants de code inaccessibles avec une condition invariablement fausse de la structure conditionnelle, une boucle exécutée 0 fois ou du code inutile inséré dans le code de capture d'une exception qui n'est jamais levée. Lorsque le code est représenté par un arbre de syntaxe, une étape de normalisation peut être réalisée afin de supprimer certaines de ces structures d'entourage inutiles lorsqu'elles sont statiquement détectables. Enfin, une méthode d'extension (cf chapitre 11) peut permettre la propagation de correspondances au-delà de ces structures d'entourage.

Modification de structures conditionnelles Nous examinons quelques opérations impliquant la modification de structures conditionnelles. Elles se caractérisent par la transposition de code, la création de nouvelles structures ou la transformation en structure de boucle :

- Dissociation/réassociation de structures conditionnelles. Les structures conditionnelles abritant un bloc *alors*, un bloc *sinon* ainsi qu'éventuellement plusieurs autres blocs *sinon* associés à une condition peuvent être éclatées en plusieurs structures conditionnelles individuelles. La condition d'entrée dans les blocs *sinon* transformés en structure *si...alors* doit vérifier si le bloc conditionnel précédent n'a pas été exécuté. Des opérations d'obfuscation peuvent également porter sur des structures de *pattern matching* complexes ou plus basiques (blocs *switch*) par conversion d'une suite de structures conditionnelles dont la condition porte sur les mêmes variables en une structure de *pattern matching* ou vice-versa.
- Inversion de blocs *alors-sinon*. Les blocs *alors* et *sinon* d'une structure conditionnelle peuvent être inversés en négativant la condition initiale. Cette modification est équivalente à une transposition de blocs de code.
- Transformation de structures conditionnelles en boucle ou structure de gestion d'exception. Une structure conditionnelle classique peut être réécrite sous la forme d'une boucle assortie d'une condition permettant une unique itération mais également par l'usage d'une structure de gestion d'exception en levant conditionnellement l'exception dans le bloc de capture et en insérant dans le bloc de gestion d'exception le code exécuté dans la structure conditionnelle originale.

Modification de type de boucle Afin d'offrir des possibilités syntaxiques plus étendues, la plupart des langages proposent dans leur grammaire plusieurs types de boucle (boucle de

type *pour* pour l'exécution itérée sur un ensemble, boucle de type *tant que* avec condition de continuation initiale ou finale, boucle de type *jusqu'à* avec condition de sortie finale...). La modification d'une boucle de type *pour* en boucle *tant que* nécessite l'extraction de la condition d'initialisation de la boucle et l'ajout en fin de boucle de l'instruction d'itération. Les boucles avec condition initiale de continuation peuvent être transformées en boucle avec condition finale de continuation en insérant une structure conditionnelle d'entrée au début de boucle pour la première itération. Préalablement à l'utilisation d'un outil de recherche de similitudes, il est donc possible d'utiliser une normalisation en un seul type de boucle.

Utilisation d'instructions de saut Il est généralement convenu que l'utilisation d'instructions de saut à la place de structures de contrôle adaptées peut dégrader la bonne compréhension du code [127]. Certains langages modernes (tel C++ ou D) conservent cependant la possibilité d'employer des instructions de saut (*goto*) et des étiquettes de branchement. Ainsi, tout code peut être linéarisé avec suppression de ses structures de contrôle et introduction de sauts et étiquettes de branchement. Des modifications moins importantes peuvent concerner également l'utilisation d'instructions de sortie de boucle (*break*) et de saut direct à la prochaine itération (*continue*) à la place d'une structure conditionnelle entourant les instructions de fin de boucle ainsi qu'une modification de la condition de début ou fin de boucle afin de permettre la sortie prématurée.

Déroulage de boucle Le déroulage de boucle consistant à dupliquer un bloc d'instructions au sein d'une boucle afin de limiter le nombre d'itérations de la boucle est généralement utilisé pour l'optimisation du code par les compilateurs afin de réduire les sauts sur le code assembleur. Dans un objectif d'obfuscation sur du code source, une telle méthode est d'effet limité dans la mesure où elle introduit des exemplaires supplémentaires de clones intra-projet facilement détectables.

4.7 Factorisation et développement de fonctions

4.7.1 Factorisation

La factorisation de fonctions est l'objectif principal de la recherche de similarité intra-projet afin de réduire le volume global de code. Elle peut être utilisée au sein d'un code copié dans un but d'obfuscation. Elle consiste alors à externaliser un morceau de code d'une fonction dans une nouvelle fonction indépendante[5]. En notant $\alpha\beta\gamma$ la portion de code originale et $\alpha'\beta'\gamma'$ l'exemplaire copié, l'externalisation de β' conduit un outil de recherche de similitudes au report des trois correspondances $(\alpha, \alpha'), (\beta, \beta'), (\gamma, \gamma')$ au lieu de l'unique correspondance $(\alpha\beta\gamma, \alpha'\beta'\gamma')$. Si un des morceaux α, β ou γ a un volume trop faible, ils peuvent ne pas être reportés pour correspondance par un outil à seuil de détection trop élevé. Un procédé extrême d'obfuscation serait alors d'externaliser individuellement chaque instruction afin d'aveugler l'outil de recherche de similitudes. Ce procédé peut être contré en réintégrant les fonctions dont le corps est de faible volume dans le corps de leurs fonctions appelantes. Cette opération devrait être réalisée en priorité sur les fonctions d'arité entrante (nombre de fonctions appelant cette fonction) basse.

[5]Le morceau de code peut également être externalisé dans une nouvelle macro pour des morceaux de faibles volumes dans des langages offrant un préprocesseur.

Nous remarquons que ce type d'obfuscation conduit à l'obtention de fonctions dont l'arité entrante est unitaire (en excluant des appels récursifs). Dans un souci de normalisation, nous pouvons redévelopper les appels en les remplaçant par le corps de la fonction appelée. Il reste néanmoins envisageable pour un plagiaire de consolider des fonctions externalisées en une unique fonction comportant une structure conditionnelle et un paramètre supplémentaire afin de déterminer le code à exécuter.

4.7.2 Développement

Le développement est l'opération inverse de la factorisation : elle consiste à remplacer un appel de fonction par son implantation complète, moyennant quelques modifications pour son adaptation au contexte local par déclaration et affectation des variables correspondant aux paramètres. Chaque occurrence de paramètre peut également être remplacée par l'expression communiquée comme argument à la fonction. Une obfuscation par développement (sauf à supprimer une fonction d'arité entrante unitaire) alourdit le code et fragmente des correspondances de taille importante comprenant des appels de fonction en leur sein en correspondances plus petites sans appel de fonction. Le remplacement des identificateurs de paramètres par des expressions différentes peut handicaper un outil de recherche de similarités exactes. Lorsqu'une fonction est développée en différents exemplaires analogues, ceux-ci peuvent être considérés comme des clones.

Une approche traitant ce type d'obfuscation (ainsi que la factorisation) est proposée au chapitre 7 sur des fonctions de séquences de lexèmes. Le principe consiste alors à ne considérer que des fonctions ne comprenant pas d'appels de fonctions qui sont ensuite factorisées en itérations successives. Nous obtenons alors finalement un nouveau graphe d'appels de fonctions factorisées commun à plusieurs projets pouvant être utilisé afin de proposer une métrique de similarité sur les fonctions originales.

Une autre approche utilisant des arbres de syntaxe est exposée au chapitre 11. Elle repose sur la recherche de correspondances exactes de sous-arbres. Ces *germes* sont ensuite consolidés et étendus afin de trouver des correspondances sur des entités structurelles plus importantes en suivant à la fois l'arbre de syntaxe mais aussi les liens d'appels de fonctions.

4.7.3 Fonctions récursives et boucles

Afin de s'affranchir de la gestion manuelle d'une pile d'appels, les algorithmes récursifs sont plus facilement codés par une fonction récursive ou un groupe de fonctions mutuellement récursives : le graphe d'appels contient donc une boucle. Un procédé d'obfuscation consiste à supprimer la récursivité d'une fonction (ou d'un groupe) par le remplacement par une fonction gérant manuellement une pile d'appels pour simuler la récursion.

Certaines fonctions récursives présentent dans les faits une récursivité terminale : un seul appel récursif à la fonction est réalisé à son terme. L'utilisation de ce type de fonctions est équivalent à l'usage d'une boucle itérative avec évolution d'une ou plusieurs variables locales. Dans les langages procéduraux, l'utilisation de fonctions récursives terminales est assez rare. Cependant, l'usage de boucles itératives est très fréquent : un obfuscateur peut les externaliser dans des fonctions à récursivité terminale.

4.8 Traduction vers un autre langage

Les langages de programmation généralistes étant dans leur majorité Turing-complets, en faisant abstraction des bibliothèques utilisées, tout algorithme implémenté dans un langage peut théoriquement être traduit dans un autre. Cependant cette traduction peut demander plus ou moins d'efforts en fonction des notions implantées par le langage. Ainsi, par exemple, la traduction d'un programme utilisant un paradigme fonctionnel (écrit par exemple en Haskell ou Caml) vers un langage procédural tel que le C est possible au prix de certaines difficultés. Pour des langages offrant les mêmes notions, en faisant abstraction des bibliothèques standards accompagnant ces langages, la traduction peut être plus simple. Par exemple, traduire un programme en langage Java vers le langage C# peut être réalisé à partir de règles de réécriture de l'arbre de syntaxe du programme en Java. Une méthode d'obfuscation par traduction dans un langage offrant des fonctionnalités proches est particulièrement intéressante car peu d'outils de recherche de similarité réalisent une comparaison inter-langages de projets à partir d'une forme normalisée de séquence de lexèmes ou d'arbre de syntaxe.

4.9 Obfuscation dynamique

Certains langages offrent des possibilités de contrôle dynamique de fonctions ou portions de code exécutées. Il s'agit alors de les exploiter afin de réaliser des opérations d'obfuscation impossibles à déceler via des méthodes d'analyse statique. Nous distinguons ici deux techniques d'obfuscation dynamique. La première se base sur le camouflage des liens d'appels entre fonction, ou alors sur l'extériorisation du code copié à exécuter. Quant à la seconde, elle utilise des possibilités d'introspection de certains langages afin soit de générer du code à l'exécution, soit de modifier dynamiquement du code existant.

4.9.1 Obfuscation des liens d'appel et chargement dynamique

Obfuscation des liens d'appel

L'utilisation de méthodes de recherche et d'extension de zones de similarité utilisant le graphe d'appels du projet est favorisée par une résolution non-ambiguë des liens entre sites d'appel et fonctions appelées. Comme évoqué en 3.4.2, la résolution des liens peut être problématique dans certaines situations telle que l'utilisation de pointeurs de fonctions ou l'existence de fonctions homonymes. Des procédés d'obfuscation consistant à rendre difficile la détermination du graphe d'appels peuvent exploiter des ambiguïtés de liens ne pouvant être levées que lors de l'exécution.

Par exemple en Java, nous pouvons systématiser l'obfuscation d'un lien d'appel vers une fonction appelée m depuis un type statiquement défini C (le type réel à l'exécution dérivant de C est alors inconnu) en créant une superclasse (classe ancêtre) A pour C qui sera utilisée pour la déclaration de l'objet et qui implantera toutes les fonctions de C (dont la fonction m) par du code aléatoire compilable. Dans le cas général, il est alors statiquement impossible de déterminer si la fonction appelée est $A.m$, $C.m$ ou une autre méthode m implantée par un descendant de C. Il est également possible de camoufler le nom de la méthode appelée en l'indiquant dans une chaîne de caractères de contenu statiquement indéterminable : la

méthode *Class.invoke(Object instance, String nomMethode)* est alors utilisée pour l'appeler. Dans le même esprit, un pointeur peut être employé en C pour appeler une fonction.

Chargement dynamique

Le chargement dynamique du code par récupération de code source ou code compilé localement ou à distance permet de masquer ce code source pour un détecteur de similarité tout en permettant son utilisation.

4.9.2 Modification dynamique de code

Certains langages possèdent des possibilités avancées de réflexivité tel que Smalltalk (ou des langages interprétés) permettant de modifier (et compiler) à l'exécution du code source. Un plagieur peut ainsi obfusquer l'utilisation du code en réalisant dans un premier temps des modifications pouvant avoir une portée sémantique sur le code. Ce code modifié, ayant un comportement à l'exécution différent du code original, subit ensuite des modifications symétriques lors de l'exécution afin de retrouver le comportement du code original.

4.10 Modifications sémantiques non-triviales

Plutôt que de copier directement un morceau de code réalisant une tâche spécifique et de réaliser ensuite des opérations d'obfuscation structurelles peu résistantes, un plagieur peut être tenté de réaliser des modifications sémantiques sur le code. Il est ainsi possible de modifier le fonctionnement même de l'algorithme implanté tout en conservant un résultat identique à l'exécution. Ces modifications peuvent être plus ou moins neutres sur la complexité mémorielle et temporelle du code. Toutefois ces opérations, même si elles peuvent s'inspirer du code original, nécessitent un investissement non-négligeable du plagiaire, parfois même supérieur à l'écriture du code *ex-nihilo*. D'autre part, la détermination d'une similarité algorithmique entre deux codes sources est, en règle générale, indécidable car étant un problème plus général que celui de l'arrêt d'un algorithme qui lui-même est indécidable. Si une similarité algorithmique est néanmoins recherchée, il demeure possible de tester les valeurs de sortie des programmes sur un sous-ensemble de l'espace des valeurs d'entrée possibles.

4.11 Récapitulatif

Les opérations d'édition et obfuscation présentées dans ce chapitre sont résumées par la figure 4.3. Nous les classons selon trois critères principaux que sont la facilité de l'automatisation des opérations d'obfuscation, les représentations ainsi que les méthodes adaptées pour la recherche de similitude en présence de ces obfuscation. Enfin, nous exposons pour chaque opération d'obfuscation sa classification selon Bellon [86] et Roy [87]. Bellon, pour son comparatif quantitatif de différents outils de recherche de clones classe les clones en trois grandes catégories : les clones de type 1 comportant pour seules opérations d'édition des modifications de formatage insensibles par analyse lexicale, les clones de type 2 avec des modifications d'identificateurs et de type et les clones de type 3 avec des modifications syntaxiques. Roy et al. quant à eux introduisent pour leur comparatif qualitatif un classement à quatre niveaux dont les deux premiers correspondent à ceux de Bellon. Le troisième niveau concerne les clones avec opérations d'ajout, suppression et modification de code ; le quatrième niveau introduit des

clones algorithmiquement équivalents avec transposition d'instructions, ajout de structures de contrôle ainsi que d'autres opérations syntaxiquement non-neutres.

Opération	Automatisation de l'obfuscation	Représentations adaptées	Méthodes adaptées	Taxonomie [86]/[87]
Modifications de formatage	Possible	Toutes sauf brute		1/1c
Édition de commentaires	Possible	Toutes sauf brute		1/1b
Renommage d'identificateurs	Possible (syntaxe)	Toutes sauf brute, lexicale non abstraite		2/2a
Transposition de code	Possible (PDG)	PDG	PDG homomorphes, alignement local de séquences, extension sur germes exacts	3/4abc
Ins./supp. de code trivialement inutile	Possible	syntaxique normalisée, PDG	PDG homomorphes, alignement de séquences, extension sur germes exacts	3/3d
Ins./supp. de code non-trivialement inutile	Difficile	Trace d'exécution	PDG homomorphes, alignement de séquences, extension sur germes exacts	3
Réécriture triviale d'expressions	Possible (syntaxe)	Syntaxique normalisée	alignement de séquences, hachage dégradé de sous-arbres	3/2bd,3ab
Réécriture non-triviale d'expressions	Difficile	Aucune	Alignement de séquences, hachage dégradé de sous-arbre	3
Changements de types	Possible (syntaxe+sémantique)	Syntaxique abstraite		2/2c
Modifications de structures de contrôle	Possible (syntaxe)	Syntaxique normalisée	Alignement de séquences, extension sur germes exacts	3/3ce,4d
Factorisation/développement de fonctions	Possible	Graphe d'appel	Factorisation, extension sur graphe d'appel	3/4
Traduction vers un autre langage	Possible	Traduction inverse, arbres de syntaxe avec abstraction de langage		3
Obfuscation dynamique	Possible	Trace d'exécution		NA
Modifications sémantiques non-triviales	Impossible (manuel)	Aucune	Aucune	NA

FIG. 4.3 – Opérations d'éditions, représentations et méthodes adaptées pour la recherche de similitudes

Fonction originale

```
int fib(int n)
{
 /* Initialisation des variables */
 int k = 1; int l = 1; int m = 0;
 /* Traitement des cas où n <= 2 */
 if (n == 1) return k;
 if (n == 2) return l;
 /* m contient la somme des deux termes
       précédents */
 for (int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 return m;
}
```

Obfuscation syntaxiquement neutre

```
int fib(n)
{
int a = 1; /* Renommage de variables */
int b = 1; int c = 0;
if (n == 1) return k;
if (n == 2) return l;
for (int i = 3; i < n; i++) { c = a + b; a
      = b; b = c; }
return m;
}
```

Transposition de code

```
int fib(int n)
{
int l = 1; int k = 1;
int m = 0;
for (int i = 3; i < n; i++)
 { m = k + l; k = l; l = m; }
if (n == 2) return l;
if (n == 1) return k;
return m;
}
```

Insertion de code inutile

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 int inutile = 1; /* Déclaration inutile */
 if (n == 1) return k;
 if (n == 2) return l;
 for (int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 for (int i = 0; i < n; i++)
  inutile++; /* Boucle inutile insérée */
 return m;
}
```

Réécriture d'expression

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 if (n == 1) return k*1;
 if (n == 2) return l+0;
 for (int i = 3; i < n; i += 1) {
  m = k + 2*l - l;
  k = l/1 + 0;
  l = m * m / (m * 1);
 }
 return pgcd(m,m);
}
```

Changement de types

```
int fib(short n)
{
 long k = 1; long l = 1; long m = 0;
 if (n == 1) return k;
 if (n == 2) return l;
 for (unsigned int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 return (int)m;
}
```

FIG. 4.4 – Fonction de calcul de nombres de Fibonacci en Java obfusquée par différentes méthodes

Modification de structures de contrôle

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 if (! (n == 1 || n == 2))
 {
  for (int j = 0; j < 1; j++)
  for (int i = 3; i < n; i++)
   { m = k + l; k = l; l = m; }
  return m;
 } else if (n == 1) return k;
 else if (n == 2) return l;
}
```

Externalisation de fonction

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 int m = fib12(n);
 if (m != -1) return m;
 else
 {
  for (int i = 3; i < n; i++)
   { m = k + l; k = l; l = m; }
  return m;
 }
}

int fib12(int n)
{
 if (n == 1) return 1;
 if (n == 2) return 2;
 return -1;
}
```

Traduction en JavaScript et obfuscation dynamique

```
/* Fonction JavaScript avec chargement
     dynamique */
function fib(n)
{
 var k = 1; var l = 1; var m = 0;
 if (n == 1) return k;
 if (n == 2) return l;
 for (var i = 3; i < n; i++)
  eval("c = a + b; a = b; b = c;".
       replace("a","k").replace("b","l").
       replace("c","m"));
 return m;
}
```

Modification sémantique non-triviale

```
int fib(int n)
{
 /* Réécriture récursive */
 if (n == 1) return 1;
 else if (n == 2) return 2;
 else return fib(n-1) + fib(n-2);
}
```

Deuxième partie

Recherche de correspondances sur des chaînes de lexèmes

Considérer le code source sous la forme d'une chaîne de lexèmes permet l'utilisation d'algorithmes de recherche de motifs classiquement utilisés sur les séquences. Nous pouvons dans un premier temps réaliser des alignements entre chaînes de lexèmes afin de déduire des sous-séquences similaires avec l'existence éventuelle de fossés : des correspondances approchées peuvent alors être mises en évidence. À cet effet, nous employons des techniques de programmation dynamique. Une digression est alors réalisée pour étendre ces méthodes aux structures bidimensionnelles que sont les arbres de syntaxe : il est alors possible de quantifier la distance entre deux sous-arbres.

Après avoir rappelé les méthodes d'alignement dans le chapitre 5, nous nous intéressons à des approches plus dimensionnables mais se limitant à la recherche de facteurs répétés exactement similaires. Les structures d'indexation de suffixes que sont la table et l'arbre de suffixes sont introduites avec d'autres structures utiles annexes. Elles seront utilisées afin de proposer la méthode de factorisation exposée dans le chapitre 7. Celle-ci permet de modéliser les similarités sur un jeu fixe de projets par le calcul d'un graphe d'appel synthétique commun à ces projets, le code dupliqué commun étant représenté par des fonctions créées.

Ensuite nous nous concentrons sur les techniques de méta-lexémisation travaillant sur des k-uplets de lexèmes consécutifs (k-grams) plutôt que sur des lexèmes uniques. L'alphabet manipulé est donc de cardinalité plus importante mais la fréquence d'apparition de chacun de ses membres est plus faible. Ces k-grams peuvent être représentés par des empreintes et indexés dans une base pour un ensemble incrémental de projets.

La figure 4.4 présente une vue synthétique des différents processus décrits au cours de cette partie avec leur rôle et coopération pour la recherche de similarité dans du code source.

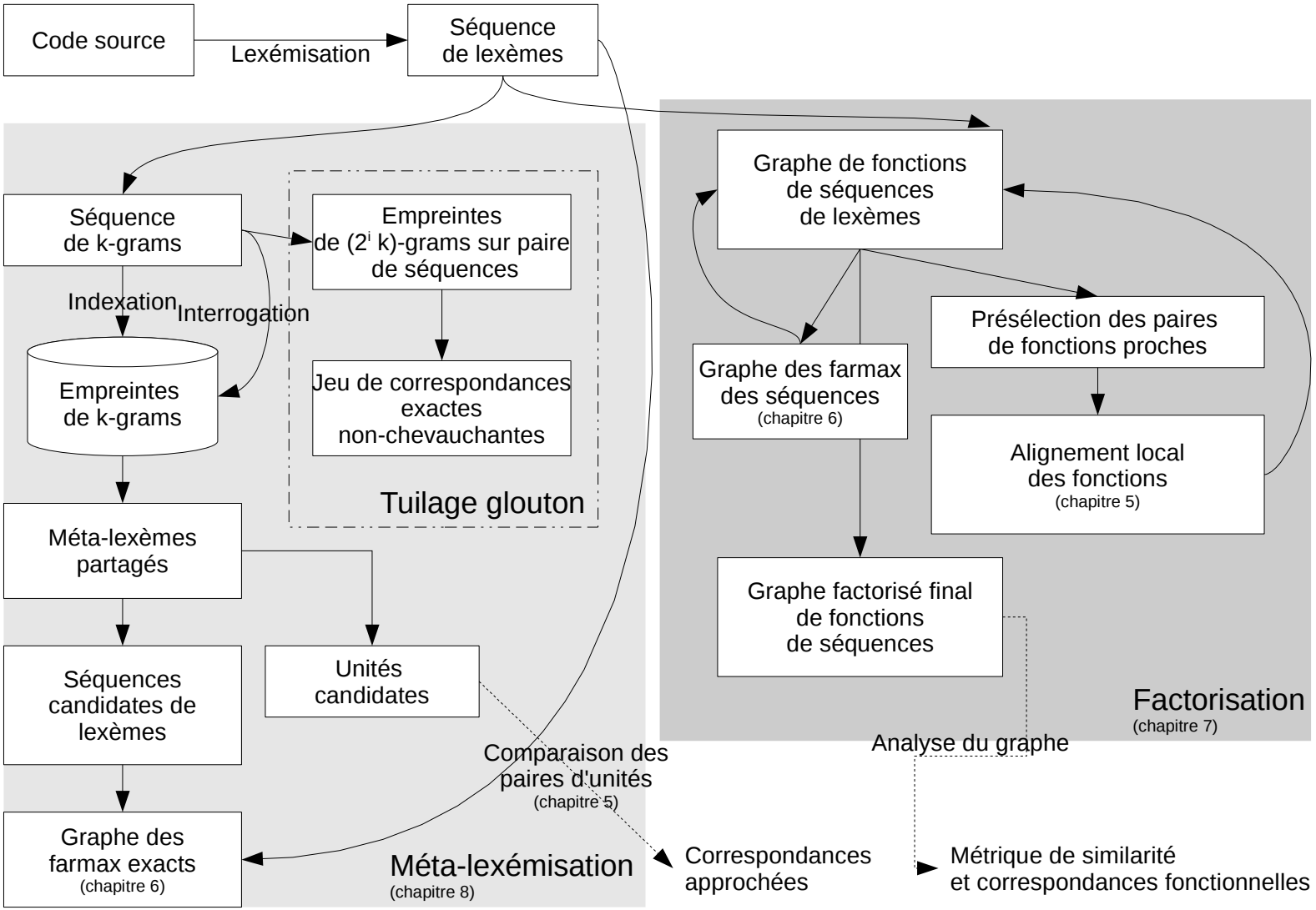


FIG. 4.4 – Vue schématique des processus de recherche de similarité sur séquences de lexèmes

5

Comparaison de paires par alignement de séquences

Sommaire

Insérer ou supprimer du code entre la version originale et la copie d'un morceau de code source induit un impact sur la représentation de celui-ci, que ce soit sous une forme lexemisée ou par un arbre de syntaxe. Ainsi la recherche de facteurs exacts ou sous-arbres exacts, que nous allons aborder ultérieurement, se révèle inadéquate car découpant des correspondances pour cause de fossés non correspondants. Nous rappelons ainsi dans ce chapitre des techniques d'alignement de chaînes et d'arbres par programmation dynamique permettant de récupérer de telles correspondances approchées. Si celles-ci sont de complexité générale prohibitive pour la comparaison globale de projets, elles peuvent s'avérer intéressantes afin de raffiner la nature de zones de forte similarité pressenties par d'autres méthodes de filtrage de complexité temporelle plus avantageuse.

5.1 Introduction à l'alignement de séquences de lexèmes

Présentation La recherche de similarité entre deux projets exprimés sous la forme de chaînes de lexèmes peut bénéficier de méthodes d'alignement de séquences. Ces méthodes utilisant des techniques de programmation dynamique sont couramment utilisées en bioinformatique afin de déterminer des zones de similarité entre séquences biologiques (nucléotides d'ADN, ARN ou acides aminés) et quantifier leur niveau de différence ou de ressemblance afin de déterminer des phylogénies. Il s'agit ainsi schématiquement de déterminer les chaînes ajoutées, supprimées ou modifiées entre les deux séquences comparées. Dans cet objectif soit une métrique de similarité, soit une métrique de distance entre deux séquences t_1 et t_2 d'éléments d'un alphabet Σ est calculée à l'aide d'opérations élémentaires que sont :

1. La correspondance entre deux éléments (*match*).
2. La suppression d'un élément (*del*).
3. L'ajout d'un élément (*add*).
4. Le remplacement d'un élément (*sub*).

À chacune de ces opérations est associée un coût $\mathcal{C}$: l'objectif étant de trouver la séquence d'opérations d'édition maximisant la métrique de similarité ou minimisant la métrique de distance. Dans le premier cas les coûts des opérations d'édition ($\mathcal{C}_{\text{del}}$, $\mathcal{C}_{\text{add}}$, $\mathcal{C}_{\text{sub}}$) sont strictement inférieurs au coût de correspondance ($\mathcal{C}_{\text{match}}$), dans le second cas il s'agit du contraire. On note que le jeu d'opérations élémentaires présenté n'est pas minimal (dans la mesure où une substitution peut être remplacée par une suppression suivie d'un ajout) ; il pourrait également être étendu avec de nouvelles opérations si cela présente un intérêt pratique (substitutions d'une chaîne de longueur non-unitaire d'éléments, coûts personnalisés utilisant des matrices de coût...).

Plus longue sous-séquence commune (LCS : *Longest Common Subsequence*) et suite d'opérations d'édition à deux chaînes d'éléments En cherchant à minimiser la distance d'édition avec $\mathcal{C}_{\text{match}} = 0 < \mathcal{C}_{\text{del}} = \mathcal{C}_{\text{add}} = \mathcal{C}_{\text{sub}}$ et en collectant la suite d'opérations élémentaires *match*, nous pouvons déterminer une plus longue sous-séquence commune à deux séquences de lexèmes u et v. Nous rappelons (voir page 11) qu'une sous-séquence u' de la chaîne u est une suite extraite de u et se présente sous la forme $u' = u'_1 u'_2 \cdots u'_{n'}$ avec pour tout $i \in [1..n'-1]$ l'existence d'indices $\alpha < \beta$ tels que $u'_i = u_\alpha$ et $u'_{i+1} = u_\beta$. La détermination de la LCS de deux suites finies de lexèmes u et v dérivés d'unités de code source nous permet de relever des chaînes discontinues de lexèmes similaires entre u et v liées à la présence de lexèmes insérés, supprimés ou modifiés par des opérations d'édition de code. Alors que la connaissance des opérations de *match* permet l'établissement de la LCS, celle complémentaire des opérations d'édition permet d'établir la séquence d'opérations élémentaires d'édition de coût minimal transformant u en v. Ainsi par exemple, le couple de chaînes $u = abc$ et $v = acd$ possède pour LCS la sous-séquence ac (obtenue par les opérations de *match* sur les lexèmes concordants a et c) et pour séquence d'opérations d'édition la suppression de b puis l'ajout de d.

Facteurs correspondants L'objectif d'une méthode d'alignement de lexèmes ne consiste pas à déterminer la LCS de deux unités lexemisées mais plutôt un ensemble de couples de facteurs exactement ou approximativement correspondants. Un couple de facteurs exactement

correspondants présente une égalité lexème par lexème contrairement aux couples approximativement correspondants possédant une distance d'édition non nulle. Le seul critère d'une distance d'édition seuil apparaît néanmoins limitatif pour juger de la pertinence d'un couple de facteurs correspondants. Celle-ci doit être mise en relation avec la longueur (voire le volume) des facteurs. La présence d'une zone de non-correspondance (fossé) importante peut également motiver la réductibilité d'un couple de correspondances, i.e. son découpage en plusieurs couples de plus petites longueurs n'intégrant pas le fossé.

5.2 Programmation dynamique

Le calcul de la métrique de similarité (ou de distance) $s(u, v)$ basée sur une des séquences d'opérations élémentaires optimales $e(u, v)$ sur une paire de séquences de lexèmes ($u = u_1 \cdots u_{m-1}u_m, v = v_1 \cdots v_{n-1}v_n$) est un problème fondamentalement récursif[1] nécessitant la connaissance de :

$\{s, e\}(u_1 \cdots u_{m-1}, v_1 \cdots v_{n-1})$	$\{s, e\}(u_1 \cdots u_{m-1}, v)$
$\{s, e\}(u, v_1 \cdots v_{n-1})$	

Il s'agit ensuite de sélectionner l'opération d'édition optimisant s à partir d'une des trois paires de sous-chaînes de u et v. Le coût de cette opération est alors ajouté pour s et celle-ci est ajoutée à la séquence d'opération e. Nous constatons que le calcul récursif des valeurs de s et e pour les trois paires de sous-chaînes occasionne des calculs redondants avec jusqu'à $3^{\max(m,n)}$ appels récursifs. Cela explique l'utilisation d'une méthode de programmation dynamique par stockage des valeurs de métrique dans une matrice où la cellule (i, j) contient $s(u_1 \cdots u_i, v_1 \cdots v_j)$: la matrice est alors calculée par balayage par ligne, colonne, diagonale ou anti-diagonale. Finalement la séquence d'opérations d'édition optimale peut être obtenue par la détermination du chemin de la matrice partant de la cellule (m, n) (chaînes (u, v)) jusqu'à la cellule $(0, 0)$ (chaînes (ϵ, ϵ)) en déduisant pour chaque cellule (i, j) l'opération d'édition optimale ainsi que la cellule prédécesseur parmi les trois cellules $(i - 1, j)$, $(i, j - 1)$ et $(i - 1, j - 1)$. Le chemin représentant la séquence d'opérations sur la matrice n'est pas nécessairement unique.

Nous constatons ainsi que la détermination de la métrique de similarité $s(u, v)$ nécessite mn opérations pour le remplissage de la matrice avec la nécessité de conserver en mémoire uniquement une ligne (calcul par balayage par ligne) ou une colonne (calcul par balayage par colonne) de la matrice et la cellule précédemment calculée. La reconstitution de la séquence d'opérations élémentaires d'édition optimale nécessite en revanche la conservation complète de la matrice, soit une mémoire de $mn \log_2(|E|)$ bits (avec E le jeu d'opérations élémentaires d'édition), les opérations pouvant être conservées à la place des valeurs de similarité.

On notera que lorsque le calcul de la matrice est réalisé par balayage par anti-diagonale, seules les deux précédentes anti-diagonales sont nécessaires au calcul des valeurs de l'anti-diagonale courante : l'absence de dépendance à une cellule de l'anti-diagonale courante autorise le calcul simultané en parallèle de toutes les valeurs des cellules de cette anti-diagonale ce qui est un moyen avantageux d'exploiter certaines capacités d'unités de traitement graphiques (GPU) [46, 73], dans la limite de la taille de leurs tampons de calcul.

[1] Pour l'initialisation, nous convenons de $s(\epsilon, \epsilon) = 0$ et $e(\epsilon, \epsilon) = \emptyset$.

Méthode mémoriellement linéaire de Hirschberg Lorsque les ressources mémorielles sont restreintes et que la séquence des opérations élémentaires d'édition est requise, la méthode de Hirschberg [44] par division récursive de l'espace matriciel peut être utilisée. Elle consiste à déterminer la cellule $(x_0, \lfloor n/2 \rfloor)$ du chemin suivi par les opérations élémentaires : x_0 peut être calculé par la conservation de pointeurs vers les cellules de la colonne $\lfloor n/2 \rfloor$ pour la colonne précédente de calcul et la cellule précédemment calculée. Finalement, le procédé est répété sur les sous-matrices correspondant aux chaînes $(u_1 \cdots u_{x_0}, v_1 \cdots v_{\lfloor n/2 \rfloor})$ et $(u_{x_0+1} \cdots u_m, v_{\lfloor n/2 \rfloor} \cdots v_n)$ pour déterminer x_{11} et x_{12} et ainsi de suite récursivement jusqu'à l'obtention de matrices de dimensions directement manipulables en mémoire centrale, voire de matrices de taille unitaire dans le cas extrême, le traitement de ces matrices pouvant être réparti sur plusieurs machines. Ce procédé ne nécessite que la conservation d'une colonne et une cellule de pointeurs et de valeur de similarité, soit $m+1$ cellules mais augmente le temps d'exécution car nécessitant le calcul d'une matrice de dimensions (m, n) à l'itération 0 (on suppose que n est une puissance de 2), de 2 matrices de dimensions $(x_{11}, n/2)$ et $(x_{12}, n/2)$ à l'itération 1, ..., de 2^i matrices de dimensions $(x_{i1}, n/2^i), \cdots, (x_{i2^i}, n/2^i)$, ... Par ailleurs, $\sum_{1 \leq k \leq 2^i} x_{ik} = m$ d'où $\frac{mn}{2^i}$ cellules à calculer à la récursion i, soit $2mn + o(mn)$ cellules pour l'ensemble des récursions : le temps d'exécution est donc multiplié par un facteur 2.

5.3 Alignement global

Les méthodes d'alignement global sur deux chaînes de lexèmes ont pour objectif de calculer une métrique de similarité ou de distance entre la *globalité* de ces chaînes ainsi que les opérations élémentaires d'édition associées. Ces méthodes sont particulièrement adaptées à la recherche de différences (séquence d'opérations d'édition) entre codes sources relativement similaires plutôt qu'à la recherche de petites similitudes enfouies dans des codes sources différents. Elles sont donc généralement utilisées par les gestionnaires de version afin de déterminer une séquence d'opérations élémentaires d'édition ainsi que la plus longue sous-séquence commune entre deux versions successives de fichier (les lexèmes élémentaires considérés étant souvent les lignes du fichier).

5.3.1 Alignement global avec code non-transposable

À titre d'exemple, nous cherchons à déterminer la plus longue sous-séquence commune et les opérations élémentaires d'édition entre les deux implantations de fonction de calcul du n^{e} nombre de Fibonacci (cf figure 5.1). Pour cela, dans un premier temps, nous identifions les instructions similaires. Ensuite nous utilisons une méthode d'alignement global sur la séquence linéaire des instructions des deux implantations :

$$f_1 = aaabbcdeec'd \quad f_2 = aaaefgcdeec'dh$$

La matrice de programmation dynamique obtenue par l'alignement global des deux chaînes est représentée en figure 5.1c avec les valeurs de métrique de distance et les flèches symbolisant les chemins de retour nécessaires afin de déterminer la séquence des opérations d'édition. Nous constatons que la distance d'édition est de 4 et que trois séquences d'opérations d'édition permettent d'obtenir cette distance minimale :

– la première comprend l'insertion de e (déclaration inutile) puis la substitution de bb par fg (réécriture des instructions *return*),

```
1  int f1(int n)
   {
3   int k = 1; /* a */
    int l = 1; /* a */
    int m = 0; /* a */
    if (n == 1) return k; /* b */
    if (n == 2) return l; /* b */
8   for (int i = 3; i < n; i++) { /* c */
     m = k + l; /* d */
     k = l; /* e */
     l = m; /* e */
    } /* c' */
13  return m; /* d */
   }
```

(a) Fonction originale f_1

```
1  int f2(int n)
   {
    int k = 1; /* a */
    int l = 1; /* a */
    int m = 0; /* a */
6   byte inutile = 0; /* e */
    if (n == 1 || n == 2) return k; /* f */
    if (n < 1) return 0; /* g */
    for (int i = 3; i < n; i++) /* c */
    {
11   m = k + l; /* d */
     k = l; /* e */
     l = m; /* e */
    } /* c' */
    return m; /* d */
16  assert(false); /* h */
   }
```

(b) Fonction copiée f_2

	ϵ	a	a	a	e	f	g	c	d	e	e	c'	d	h
ϵ	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	1	↖ 0	1	2	3	4	5	6	7	8	9	10	11	12
a	2	1	↖ 0	1	2	3	4	5	6	7	8	9	10	11
a	3	2	1	↖ 0	← 1	2	3	4	5	6	7	8	9	10
b	4	3	2	1	↖ 1	←↖ 2	3	4	5	6	7	8	9	10
b	5	4	3	2	2	↖ 2	↖← 3	4	5	6	7	8	9	10
c	6	5	4	3	3	3	3	↖ 3	4	5	6	7	8	9
d	7	6	5	4	4	4	4	4	↖ 3	4	5	6	7	8
e	8	7	6	5	4	5	5	5	4	↖ 3	4	5	6	7
e	9	8	7	6	5	5	6	6	5	4	↖ 3	4	5	6
c'	10	9	8	7	6	6	6	7	6	5	4	↖ 3	4	5
d	11	10	9	8	7	7	7	7	7	6	5	4	↖ 3	← 4

(c) Matrice de distance d'édition avec information de retour

FIG. 5.1 – Copie de fonction avec instructions insérées, supprimées et remplacées avec la matrice de distance d'édition (par lignes) correspondante

– la seconde considère le remplacement de b par e, l'insertion de f et le remplacement de b par g,
– la troisième alternative consiste à remplacer bb par ef puis à insérer g.

Dans une optique de recherche de similarité, la première alternative est préférable. Il aurait été possible de supprimer la seconde et la troisième alternative en utilisant des coûts de remplacement personnalisés selon la similarité des instructions (avec ici $\mathcal{C}_{\text{sub}}(b,f), \mathcal{C}_{\text{sub}}(b,g) < \mathcal{C}_{\text{sub}}(b,e)$).

5.3.2 Alignement global avec code transposable

Les méthodes d'alignement global montrent leur limitation lorsque des opérations de transposition de volume important de code sont réalisées. Nous étudions ici deux cas caractéristiques. Le premier concerne une unité composée de deux zones de code qui auraient été échangées (telles que des fonctions) : seule la plus grande zone (en termes de lexèmes) est reportée comme LCS. Pour le second cas, des petits facteurs communs conduisent au report d'une LCS rendant invisible l'existence d'un facteur commun plus grand.

La plus longue sous-séquence commune est un facteur (ou un quasi-facteur)

Nous considérons deux fonctions f et g implantées dans l'ordre fg pour l'unité de compilation 1 et gf pour l'unité de compilation 2. f et g sont représentées par leurs séquences d'instructions. La détermination de la plus longue sous-séquence commune par alignement global retourne la sous-séquence correspondant à f si $|f| > |g|$: aucune similitude n'est alors relevée pour g. Plus généralement si une paire de séquence de lexèmes u et v possèdent les facteurs communs maximaux (i.e. extensibles ni sur la gauche, ni sur la droite) $\alpha_1, \alpha_2, \cdots, \alpha_k$, les facteurs étant deux à deux distincts, alors la plus longue sous-séquence commune de lexèmes est la plus longue sous-séquence commune de facteurs communs maximaux. Ainsi si u est de la forme $\alpha_1 \cdots \alpha_2 \cdots\cdots \alpha_k$ et v de la forme $\alpha_k \cdots \alpha_{k-1} \cdots\cdots \alpha_1$, la plus longue sous-séquence commune de lexèmes sera le plus grand facteur commun parmi l'ensemble des facteurs communs $\{\alpha_i\}_{1 \leq i \leq k}$.

La remarque précédente peut également être étendue pour des couples de facteurs approximativement correspondants, extension floue de la notion de facteur commun intégrant des fossés de non-correspondance.

La sous-séquence commune la plus longue étant une sous-chaîne, une solution pourrait alors être de réexécuter un alignement global sur les séquences u et v après avoir supprimé le plus grand facteur commun sélectionné. La complexité temporelle globale du procédé itéré est alors cubique en la longueur $\max(|u|, |v|)$.

La plus longue sous-séquence commune n'est pas un facteur

Nous envisageons maintenant le cas où la plus longue sous-séquence commune n'est pas un facteur (ou un quasi-facteur). C'est notamment le cas si $u = \cdots\alpha\cdots\beta_1\cdots\beta_2\cdots\cdots\beta_k$ et $v = \cdots\beta_1\cdots\beta_2\cdots\cdots\beta_k\cdots\alpha\cdots$ avec α le plus grand facteur commun et $\beta_1, \beta_2, \cdots, \beta_k$ des petits facteurs communs tels que $\sum_{1 \leq i \leq k} |\beta_i| > |\alpha|$: la plus longue sous-séquence commune est $\beta_1\beta_2\cdots\beta_k$. L'existence du facteur commun α n'est donc pas reportée.

5.4 Alignement local

Comme vu précédemment, les méthodes d'alignement global s'avèrent difficilement utilisables lorsque du code transposé est présent entre les deux unités comparées. Une idée est donc de privilégier non pas la recherche de la plus longue sous-séquence commune mais des quasi-facteurs communs les plus intéressants entre les deux sous-séquences de lexèmes.

5.4.1 Algorithme de Smith-Waterman

Nous souhaitons introduire une métrique de similarité pour les sous-séquences locales intéressantes. Dans cette optique, nous utilisons la méthode de Smith-Waterman [47]. Nous utilisons un coût positif ($\mathcal{C}_{\text{match}}$) pour l'opération de correspondance (extension de correspondance) tandis que des coûts négatifs sont adoptés pour l'ajout, la suppression ($\mathcal{C}_{\text{add}} = \mathcal{C}_{\text{del}}$) et le remplacement ($\mathcal{C}_{\text{sub}}$).

Afin de ne pas prolonger inutilement des sous-séquences qui présenteraient une similarité inintéressante, $\omega \leq 0$ est fixé comme borne minimale de la métrique de similarité. Nous obtenons alors les relations de récurrences suivantes pour le calcul de la matrice de programmation dynamique :

$$s[i][j] = s(u[1..i], v[1..j]) = \begin{cases} \omega & \text{si } (i,j) = (0,0) \\ s(u[1..i-1], v[1..j-1]) + \mathcal{C}_{\text{match}} & \text{si } u[i] = u[j] \\ \max \left(\begin{array}{l} \omega \\ s(u[1..i-1], v[1..j]) + \mathcal{C}_{\text{add}} \\ s(u[1..i], v[1..j-1]) + \mathcal{C}_{\text{del}} \\ s(u[1..i-1], v[1..j-1]) + \mathcal{C}_{\text{sub}} \end{array} \right) & \text{sinon} \end{cases}$$

La recherche de sous-séquences locales utiles est ensuite réalisée en localisant les cellules (i, j) présentant la métrique de similarité la plus élevée non-nulle : une sous-séquence correspondante peut alors être trouvée par détermination du chemin des opérations d'édition par remontée dans la matrice, jusqu'à l'atteinte d'une cellule de similarité ω (ω correspondant au coût d'ouverture d'une sous-séquence).

La principale difficulté réside alors dans la détermination des valeurs de similarité localement maximales $s(u[1..i], v[1..j])$ dans la matrice telles que ces cellules ne soient pas dominées. Il y a domination d'une cellule lorsque celle-ci appartient à un chemin d'opérations élémentaires comportant une cellule de valeur de similarité supérieure. Deux types de domination sont à distinguer :

- La post-domination intervient lorsque la cellule (i, j) est suivie par une cellule (i', j') avec $i' > i$ et $j' > j$: dans ce cas, la sous-séquence peut être étendue jusqu'à (i', j') pour obtenir une meilleure valeur de similarité.
- La pré-domination survient lorsque la cellule (i, j) est précédée par une cellule (i', j') avec $i' < i$ et $j' < j$: la sous-séquence doit alors être réduite à (i', j').

Une méthode peut alors consister à récupérer les chemins d'opérations élémentaires des cellules non-dominées de plus forte similarité à la cellule de plus faible similarité. Ceci ne nous garantit cependant pas que les sous-séquences trouvées ne se chevauchent pas.

Réitération Un moyen de localiser les sous-séquences de similarité maximale non-chevauchantes peut consister à réitérer à chaque fois l'application de l'algorithme de recherche sur les zones ne participant pas à la sous-séquence trouvée. Ainsi si nous recherchons les sous-séquences de $u[1..m]$ et $v[1..n]$, nous déterminons en premier lieu la cellule (i, j) de similarité maximale M : ceci peut être réalisé en espace linéaire. Ensuite avec la méthode de Hirschberg, nous déterminons un chemin d'opération d'éditions (parmi tous les existants) menant de (i, j) à une cellule de similarité ω (notée (i_0, j_0)) : ce chemin ne doit pas comporter de similarité M ; si cela est le cas, la sous-séquence est close à cette cellule. Nous obtenons finalement une sous-séquence extraite des facteurs $u[i_0..i]$ et $v[j_0..j]$. Il reste alors à réitérer la recherche de la sous-séquence de similarité maximale sur les facteurs $u[1..i_0 - 1], u[i + 1..m]$ et $v[1..j_0 - 1]$, $v[j + 1..n]$, soit 4 paires à comparer. Globalement, cette méthode se révèle de complexité temporelle en $O(\max(m, n)^3)$. Une méthode [49], de coût temporel plus avantageux mais nécessitant le stockage intégral de la matrice, consiste à remplacer les valeurs des cellules du chemin de la sous-séquence trouvée par ω et à recalculer uniquement les $O((i - i_0)^2)$ cellules — en bas et à droite du début du chemin (i_0, j_0) — affectées par ce changement. Pour K alignements locaux intéressants de longueur moyenne L relevés, la complexité temporelle s'élève alors en $O(nm + KL^2)$.

5.4.2 Algorithme amélioré avec coupure

Les sous-séquences obtenues par la méthode précédemment présentée sont certes de similarité maximale, mais pour des besoins pratiques de recherche de similarité sur des séquences de lexèmes, nous tolérons la présence d'ajout, de suppression et de remplacement de lexèmes isolés mais pas de longues séquences. Pour répondre à cette problématique, [67] propose de casser le suivi d'une sous-séquence dans la matrice de similarité lorsque nous constatons que la cellule en cours est largement pré-dominée par une autre cellule ; la sous-séquence est alors close par cette cellule pré-dominante. En pratique, il s'agit de déterminer pour chaque cellule (i, j) la similarité de la cellule pré-dominante en utilisant une matrice auxiliaire m définie ainsi :

$$m[i][j] = \begin{cases} \omega & \text{si } s[i][j] = 0 \\ \max(m[i-1][j-1], s[i-1][j-1]) & \text{si } u[i] = u[j] \\ \max_{(p,q)\in\mathcal{P}((i,j))}(m[p][q], s[p][q]) & \text{sinon} \end{cases}$$

avec $\mathcal{P}((i, j))$ les cellules prédécesseures de (i, j). Cette matrice est utilisée pour calculer une valeur de similarité révisée s' réinitialisée à ω pour couper la sous-séquence suivie lorsque la cellule courante est trop largement pré-dominée, i.e. $m - s \geq t_c$ où t_c est le seuil de coupure fixé. Ce seuil peut éventuellement être variable en étant une fonction croissante de s : nous sommes alors plus tolérants pour la coupure lorsque la sous-séquence suivie est déjà de similarité importante. Une coupure systématique peut également être imposée lors de la fin d'une unité syntaxique (telle qu'une fonction) afin d'éviter l'obtention de sous-séquences chevauchantes entre unités.

5.4.3 Raccordement de facteurs par matrice *dotplot*

La matrice *dotplot* d'une paire de séquences de lexèmes (u, v) de longueurs respectives m et n est définie comme la matrice de booléens de dimensions m et n dont la cellule de coordonnées

(i, j) spécifie si $u[i] = v[j]$. Elle est couramment utilisée pour la représentation visuelle de similarités (comme décrit en 14.1.2).

Plutôt que de calculer la globalité de la matrice de similarité (dans la plupart des cas assez creuse) pour déterminer ensuite les sous-séquences en correspondance les plus intéressantes, il peut être plus judicieux de déterminer dans un premier temps l'ensemble des facteurs répétés maximaux entre les deux séquences de lexèmes analysées u et v puis de rechercher ensuite des raccordements (représentant des *fossés* de non-correspondance). Cette approche utilisée par [79] et [81] se révèle moins coûteuse en présence d'un petit nombre de facteurs correspondants. La recherche des k occurrences de facteurs répétés maximaux peut être menée en temps $O(m + n + k)$ en utilisant une structure d'indexation de suffixes[2] (voir chapitre 6).

Il est nécessaire ensuite de raccorder des paires de clones exacts (visualisés par une diagonale sur la matrice *dotplot*) qui sont proches entre-eux. À cet effet, nous pouvons décider d'imposer un critère de distance d'édition pour le fossé de raccordement inférieur à un seuil t_g fixé. On détermine ensuite pour chaque paire de clones exacts (c_1, c_2) les paires de clones exacts les plus proches (dont la tête est à une distance inférieure à t_g de la queue de (c_1, c_2)) : ceci peut être réalisé par l'utilisation d'un index sur les têtes de clones. Nous obtenons ainsi finalement un graphe acyclique de clones exacts reliés entre-eux par des fossés matérialisés par des arêtes. Les sous-séquences communes peuvent ensuite être extraites par le parcours de ce graphe avec filtrage de sous-séquences de similarité trop faible (comportant des fossés trop importants) et éventuelle détermination de sous-séquences non-chevauchantes.

Nous présentons en figure 5.2 un exemple de raccordement de clones exacts : ici les facteurs a, b, c et d représentent des correspondances exactes sur u et v. Alors que $u = abcd$, v présente une transposition entre b et c. Deux possibilités de raccordement sont proposées : en pratique, puisque $|c| < |b|$, le raccordement permettant l'obtention de la correspondance approchée utilisant la sous-séquence abd sera utilisée car minimisant les distances de raccordement.

5.5 Extension aux alignements sur les arbres

5.5.1 Approches algorithmiques

Les méthodes d'alignement étudiées précédemment peuvent être généralisables pour des séquences à n dimensions, i.e. comportant n types de relations. En particulier, nous nous intéressons aux séquences à 2 dimensions que sont les arbres à nœuds étiquetés et ordonnés (dotés de deux types de relation que sont les relations de fratrie et de parenté). Nous pouvons ainsi définir des coûts pour l'ajout, la suppression et le remplacement d'un frère ou alors d'un enfant : nous recherchons ensuite une métrique de similarité ou une métrique de distance quantifiant l'ensemble des opérations d'édition. En conservant la matrice de programmation dynamique (de dimension 4 pour la comparaison d'arbres), il est possible de reconstituer une séquence optimale d'opérations d'édition pour transformer un arbre en l'autre. La figure 5.3 présente sur une petit exemple une séquence d'édition optimale pour transformer un arbre en un autre.

[2] [81] propose une méthode moins efficace où les paires de clones sont déterminés par les diagonales de la matrice *dotplot*, ce qui nécessite mn comparaisons de lexèmes. [79] utilise CCFinder [71] pour la recherche de clones exacts par arbre de suffixes.

$$u = \cdots (a_1 a_2 \cdots a_w)(b_1 b_2 \cdots b_x)(c_1 c_2 \cdots c_y)(d_1 d_2 \cdots d_z) \cdots$$
$$v = \cdots (a_1 a_2 \cdots a_w)(c_1 c_2 \cdots c_y)(b_1 b_2 \cdots b_x)(d_1 d_2 \cdots d_z) \cdots$$

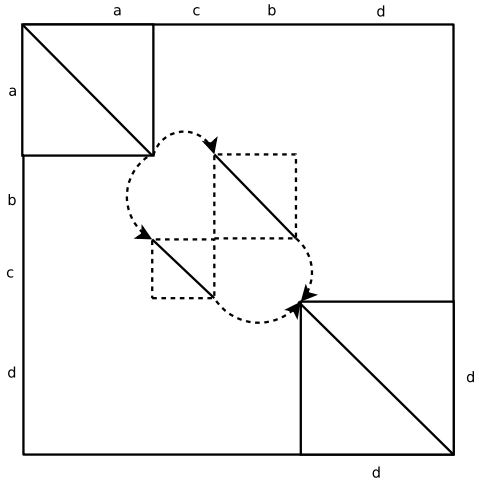
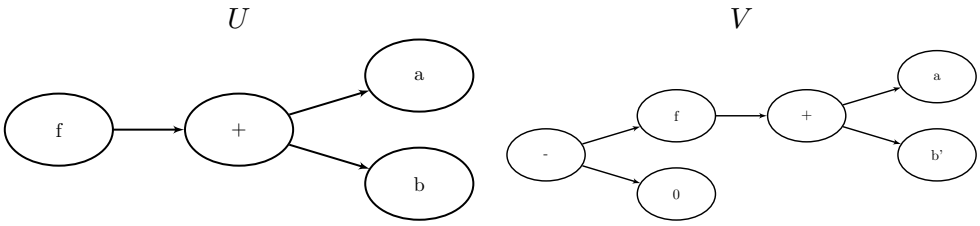


FIG. 5.2 – Raccordement de clones exacts proches sur matrice *dotplot* avec transposition d'un petit morceau de code



Distance d'édition : $s(U, V) = \mathcal{C}_{\text{add}}(-) + \mathcal{C}_{\text{add}}(0) + \mathcal{C}_{\text{sub}}(b, b')$

FIG. 5.3 – Distance d'édition sur deux arbres syntaxiques d'expression

Nous notons que la comparaison d'arbres dont les enfants d'un nœud ne sont pas ordonnés — ce qui peut être utile pour des structures à enfants commutatifs (opérateurs commutatifs, classes avec fonctions d'ordre indifférent) — est un problème NP-complet dans le cas général nécessitant la comparaison avec tous les ordonnancements possibles.

Si nous souhaitons comparer deux forêts d'arbres (chaînes d'arbres ordonnés) $\mathcal{F}_1$ et $\mathcal{F}_2$, nous pouvons décomposer ces forêts de deux manières en isolant l'arbre de gauche ou l'arbre de droite :

$$\mathcal{F}_i = \underbrace{\gamma_i(A_i) \cdot F_i}_{\text{décomposition gauche}} \quad \text{ou} \quad \underbrace{F_i \cdot \gamma_i(A_i)}_{\text{décomposition droite}}$$

À la suite du choix d'une décomposition, la formule de récurrence suivante est utilisée pour calculer la valeur de similarité entre les forêts $\mathcal{F}_1$ et $\mathcal{F}_2$ (nous notons $F_i - \beta$ la forêt F_i dont le nœud β a été supprimé et remplacé par ses enfants directs et $F_i - \beta(A)$ la forêt dont l'arbre $\beta(A)$ a été supprimé) :

$$s(\mathcal{F}_1, \mathcal{F}_2) = \min \begin{cases} s(\gamma_1(A_1), \gamma_2(A_2)) + s(\mathcal{F}_1 - \gamma_1(A_1), \mathcal{F}_2 - \gamma_2(A_2)) \\ s(\mathcal{F}_1 - \gamma_1, \mathcal{F}_2) + \mathcal{C}_{add} \\ s(\mathcal{F}_1, \mathcal{F}_2 - \gamma_2) + \mathcal{C}_{del} \end{cases}$$

Lorsque les deux forêts comparés se limitent à deux arbres (à une racine) $\gamma_1(A_1)$ et $\gamma_2(A_2)$, nous utilisons la récurrence suivante :

$$s\left(\gamma_1(A_1), \gamma_2(A_2)\right) = \min \begin{cases} s(A_1, A_2) + \mathcal{C}_{match/sub}(\gamma_1, \gamma_2) \\ s(\gamma_1(A_1), A_2) + \mathcal{C}_{del}(\gamma_1) \\ s(A_1, \gamma_2(A_2)) + \mathcal{C}_{add}(\gamma_2) \end{cases}$$

Décomposer les forêts sur la gauche ou la droite entraîne un nombre d'appels récursifs variable pour le calcul des valeurs de similarité. Ainsi en utilisant une décomposition systématique sur la gauche ou sur la droite, nous obtenons l'algorithme utilisé par Zang et Shasha [50] dont la complexité temporelle est en :
$\Theta\left(\prod_{i\in\{1,2\}} |F_i| \min\left(\text{hauteur}(F_i), \text{feuilles}(F_i)\right)\right)$.
Une amélioration introduite par Klein [45] consiste à comparer la taille des arbres à gauche (L_1) et à droite (R_1) afin de réaliser une décomposition sur la gauche si $|L_1| < |R_1|$, sur la droite dans le cas contraire. Cette stratégie de décomposition permet de réduire la complexité temporelle en $\Theta(|F_1| \log |F_1| + |F_2|^2)$ avec une complexité mémorielle en $\Theta(|F_1||F_2|)$. Des stratégies de décomposition encore plus efficaces [48, 43] peuvent être employées afin d'améliorer la complexité temporelle.

5.5.2 Applications

La détermination de la distance d'édition entre deux arbres, de par son importante complexité, ne peut raisonnablement pas être utilisée directement pour la comparaison systématisée de grands arbres de syntaxe (comportant typiquement plusieurs milliers de nœuds). En revanche, pour la comparaison de structures syntaxiques de taille limitée, celle-ci peut comporter un intérêt.

Le chapitre 9 est consacré aux méthodes de hachage sur des arbres de syntaxes. Nous y examinons différentes vues abstraites des arbres de syntaxe conduisant à l'obtention de valeurs de hachage dégradées. Parmi les abstractions envisageables figure le remplacement de tous les petits sous-arbres par un nœud unique. Cette abstraction permet d'attribuer une valeur de hachage unique à des sous-arbres différant par le contenu de leurs petits sous-arbres. Des couples de sous-arbres de même abstraction peuvent ensuite être comparés de façon plus approfondie par la détermination de la séquence d'éditions élémentaires et de la similarité afférente permettant de transformer l'un en l'autre. Cette technique est notamment utilisée par l'outil CloneDr [63, 91].

6

Utilisation de structures d'indexation de suffixes

Sommaire

Nous présentons dans ce chapitre des structures d'indexation de suffixes telles que l'arbre ou la table de suffixes ainsi que des structures annexes nous permettant de localiser les facteurs répétés de projets lexemisés. Ces structures ainsi que des algorithmes introduits dans ce chapitre serviront de base à la méthode de factorisation explicitée au chapitre 7.

Après quelques définitions préliminaires, nous introduisons l'arbre de suffixes avec la description rapide de quelques algorithmes de construction existants en temps linéaire. Cette structure nous permet de trouver les facteurs répétés d'un ensemble de séquences de lexèmes. Elle est cependant très consommatrice en mémoire centrale, c'est pourquoi nous nous orientons ensuite vers une approche plus légère utilisant une table de suffixes. La table de suffixes peut être construite en temps linéaire et grâce à une table annexe des plus longs préfixes communs, peut servir de base à l'établissement d'un arbre des intervalles, sorte d'arbre de suffixes dont les feuilles auraient été supprimées. À l'aide de l'arbre des intervalles, nous introduisons une structure nouvelle, le graphe des farmax, modélisant l'ensemble des groupes de facteurs répétés existants sur les séquences avec relations de chevauchement sur la gauche (préfixe) et sur la droite (suffixe). Ce graphe généralise l'arbre des intervalles où seule la relation préfixe est considérée entre les intervalles représentant les groupes de facteurs répétés. Il s'agit ensuite d'utiliser le graphe des farmax afin de décomposer une chaîne provenant d'un ensemble de chaînes en facteurs extraits des chaînes plus petites de cet ensemble. Enfin le terme du chapitre est consacré à une discussion des avantages et inconvénients respectifs des arbres et tables de suffixes.

6.1 Introduction

Étant donnée une représentation de code source sous la forme de chaînes de lexèmes, une première approche à la recherche de similarité réside dans la recherche de sous-chaînes exactes répétées. Les facteurs répétés en multiples occurrences au sein d'une unique chaîne (représentation d'un unique projet ou unité de compilation) ou de multiples chaînes (représentation de plusieurs projets ou unités de compilation) témoignent de la présence d'un groupe de clones. La validité du clone ainsi détecté est dépendante du niveau d'abstraction choisi pour la représentation en lexèmes ainsi que de la longueur du facteur répété. Pour la suite, nous confondrons les facteurs répétés avec leurs occurrences associées sur les chaînes traitées.

6.1.1 Facteurs répétés : définitions

Nous considérons ainsi k projets chacun représenté par une chaîne de lexèmes (de longueurs respectives $\ell_1, \ell_2, \ldots, \ell_n$) $\mathcal{T} = \{T_1 = t_{1,1} \cdots t_{1,\ell_1}, \cdots, T_n = t_{n,1} \cdots t_{n,\ell_n}\}$. Nous notons la longueur cumulée $L = \ell_1 + \ell_2 + \cdots + \ell_n$. L'alphabet de lexèmes manipulé est doté d'une opération transitive d'égalité $=$ sur les lexèmes. Nous introduisons quelques définitions pour nous mener à la recherche de facteurs répétés maximaux (farmax). Ceux-ci peuvent être associés à l'ensemble exhaustif de leurs occurrences qui ne peuvent être étendus par l'incorporation de lexèmes contigus (maximalité). Ainsi par exemple, l'instruction lexemisée $x = (a+1) * (a-1)$ avec une opération d'égalité confondant les opérateurs conduit à l'obtention d'un famax de longueur d'au moins trois lexèmes : $(a \,\mathrm{op}\, 1)$.

Définition 6.1. *(Facteur répété)* R est un facteur répété (far) de T ssi il existe $k \geq 2$ occurrences de R (que nous noterons $\{r_1 = T_{i_1}[\alpha_1|\ell], \cdots, r_k = T_{i_k}[\alpha_k|\ell]\}$). Nous avons alors

égalité lexème par lexème pour les occurrences deux à deux ($\forall 1 \leq p \leq q \leq k, 1 \leq t \leq \ell, r_p[\alpha_p + t] = r_q[\alpha_q + t]$).

Définition 6.2. *(Facteur répété maximal — farmax —)* Un facteur répété R d'occurrences $\{r_1, \cdots, r_k\}$ est maximal (farmax) ssi ses occurrences ne peuvent être étendues ni sur la gauche, ni sur la droite, i.e. il n'existe pas d'occurrences $r'_1 = T_{i_1}[\alpha_1 - u|l + v]$, ..., $r'_k = T_{i_k}[\alpha_k - u|\alpha_k + v]$ avec $u \geq 0$, $v \geq 0$ et $u + v > 0$ tels que $R' : \{r'_1, \cdots, r'_k\}$ soit un facteur répété.

Définition 6.3. *(Occurrence propre)* Une occurrence d'un facteur répété est dite *occurrence propre* si celle-ci n'est pas comprise dans un autre facteur répété. Lorsqu'un facteur répété ne comprend aucune occurrence propre, celui-ci est dit *facteur répété unificateur.*

Ainsi, déterminer les facteurs répétés maximaux farmax(T) et leurs ensembles d'occurrences associés équivaut à trouver l'ensemble des groupes de clones pertinents, complets et maximaux comme explicité en section 2.2. L'oracle de pertinence utilisé considère comme pertinents un groupes de clones ssi leurs représentations lexemisées (occurrences de facteurs de lexèmes) sont égales deux à deux, lexème par lexème.

Exemple Nous présentons ici un petit exemple illustrant les précédentes définitions de facteurs répétés sur trois chaînes de lexèmes $T = \{\alpha, \beta, \gamma\}$ ainsi définies :

$$\begin{aligned} \alpha &= abcd \\ \beta &= cdada \\ \gamma &= eabcdae \end{aligned}$$

Nous obtenons l'ensemble des facteurs maximaux suivants avec leurs occurrences associées (les occurrences propres sont exprimées en gras) :

$$\text{farmax}(T) = \left\{ \begin{array}{l} \{abcd : \boldsymbol{\alpha[1..4]}, \boldsymbol{\gamma[2..5]}\} \\ \{cd : \alpha[3..4], \beta[1..2], \gamma[4..5]\} \\ \{d : \alpha[4], \beta[2], \beta[4], \gamma[5]\} \\ \{cda : \boldsymbol{\beta[1..3]}, \boldsymbol{\gamma[4..6]}\} \\ \{da : \beta[2..3], \boldsymbol{\beta[4..5]}, \gamma[5..6]\} \\ \{a : \alpha[1], \beta[3], \beta[5]\} \\ \{e : \boldsymbol{\gamma[1]}, \boldsymbol{\gamma[7]}\} \end{array} \right\}$$

Après avoir vérifié l'exhaustivité de chacun des occurrences associées à chacun des facteurs, nous examinons la maximalité de chaque facteur répété. Si nous considérons par exemple le facteur *cd*, les occurrences $\beta[1..2]$ et $\gamma[4..5]$ peuvent être étendues sur la droite pour obtenir le facteur répété *cda* ; l'extension sur la gauche est possible, elle, pour les occurrences $\alpha[3..4]$ et $\gamma[4..5]$ en le facteur répété *bcd* ($\beta[1..2]$ ne pouvant être étendu sur la gauche). En revanche, il est impossible d'étendre simultanément les trois occurrences de *cd* d'un même lexème sur la gauche ou sur la droite : *cd* est donc bien maximal. Nous constatons cependant que, contrairement à *cda* qui est maximal, *bcd* ne l'est pas et ne figure pas dans farmax(T) ($\{bcd\} \subset (\text{far}(T) - \text{farmax}(T))$). Les deux occurrences de *bcd* sont en effet extensible sur la gauche en *abcd* qui lui est un facteur maximal.

Indexation de suffixes Une structure d'indexation de suffixes permet de calculer aisément l'ensemble des facteurs répétés complets d'un ensemble de chaînes de lexèmes. Elle autorise un accès rapide à tous les suffixes des sous-chaînes débutant par un préfixe donné. Nous introduisons dans le reste de ce chapitre les structures d'arbre (section 6.2) et de table de suffixes (section 6.3) permettant l'indexation de suffixes en temps linéaire (sous-sections 6.2.2 et 6.3.2) de la taille cumulée des sous-chaînes manipulées. Nous ne nous intéresserons pas à la structure d'automate de suffixes (ou de facteurs) plus adaptée à la recherche de facteurs sur un corpus de chaînes fixes avec une contrainte mémorielle forte qu'à l'énumération exhaustive des facteurs répétés. Nous nous concentrerons spécifiquement sur la structure de table de suffixes et étudierons des structures annexes telle que l'arbre des intervalles (section 6.4) afin de déterminer les facteurs répétés (dont ceux maximaux) d'un ensemble de séquences de lexèmes.

6.1.2 Implantations

Les structures d'indexation de suffixes ont pour principal intérêt la recherche rapide de chaînes requêtes sur un ensemble de chaînes indexées. Elles présentent également une utilité pour la recherche de chaînes répétées. À l'origine celles-ci ont été utilisées pour la manipulation de séquences biologiques ou pour la recherche de facteurs requêtes sur du texte en langue naturelle. Elles ont ensuite également été employées par certaines implantations d'outil de recherche de similarité sur du code source (ou sur du langage naturel [132]). Nous pouvons ainsi citer Phoenix [78] et CCFinder [71, 89] qui réalisent une recherche de facteurs répétés par table ou arbre de suffixes sur des chaînes issues de la lexémisation de code source. Dans [72], Koschke et al. proposent une approche équivalente en travaillant sur une représentation par arbre de syntaxe abstrait qui est ensuite transformé en chaîne de lexèmes par parcours de l'arbre. Une recherche de facteurs répétés est ensuite réalisée après construction d'un arbre de suffixes.

Nous décrivons dans le chapitre 7 une méthode de factorisation de fonctions représentées par des chaînes de lexèmes utilisant une table de suffixes pour la détermination des facteurs à factoriser et externaliser. Enfin dans le chapitre 10, une méthode d'extension de correspondances sur empreintes de sous-arbres de syntaxe abstraits utilisant une structure de table de suffixes est développée. Celle-ci permet de retrouver des chaînes d'arbres frères similaires au sein d'arbres de syntaxe en évitant la lexémisation complète de ces arbres telle qu'elle est effectuée dans [72].

6.2 Arbre de suffixes

6.2.1 Définition

Définition 6.4. *(Arbre de suffixes)* L'arbre des suffixes $\mathrm{ST}(T)$ des chaînes de lexèmes T est l'arbre de racine ϵ dont les nœuds sont étiquetés par des facteurs de T et dont :

1. L'ensemble des chemins (concaténation des étiquettes des nœuds) est homomorphe à l'ensemble des facteurs de T.
2. L'ensemble des chemins terminés par un nœud terminal est homomorphe à l'ensemble des suffixes de T.

3. Chaque facteur correspond à un chemin unique.

L'arbre de suffixes de l'ensemble des chaînes de lexèmes $T = \{T_1, \ldots, T_n\}$ est l'arbre lexicographique construit à partir de l'ensemble des suffixes de T. Un tel arbre représentant L suffixes est composé dans le pire des cas de $\frac{L(L+1)}{2} + 1$ nœuds si tous les facteurs sont distincts. En pratique, seuls les nœuds internes d'arité sortante d'au moins 2 ne représentant pas une occurrence de suffixe sont conservés en fusionnant les chaînes de nœuds non-suffixes successifs d'arité sortante unitaire. Nous obtenons ainsi l'arbre compact de suffixes de $\mathcal{T}$ comportant au maximum $2L-1$ nœuds [1]. Par la suite tous les arbres de suffixes manipulés sont présumés compacts. Chacune des feuilles de l'arbre représente au moins un suffixe (plusieurs si les chaînes comportent plusieurs suffixes égaux) : elles sont étiquetées par les chaînes d'appartenance et les positions des suffixes.

Nous notons que pour chaque nœud $a \cdot u$ de l'arbre de suffixes ($a \in \Sigma, u \in \Sigma^*$) il existe son suffixe u : $a \cdot u$ est connecté à u par un lien suffixe. L'arbre de suffixes de l'exemple introduit en 6.1.1 est présenté en figure 6.1 avec les liens suffixes des seuls nœuds internes.

6.2.2 Construction

Quelques algorithmes de construction

Différents algorithmes ont été proposés pour la construction d'un arbre de suffixes d'un ensemble de chaînes T de taille cumulée L en $O(L)$. Une première approche naïve consiste à insérer les suffixes de chaque chaîne de T dans l'ordre inverse de leur position dans un arbre lexicographique. Chaque suffixe devant être entièrement parcouru pour son insertion, cela nécessite dans le pire des cas $\sum_{k=1}^{|T|} \frac{1}{2} l_k(l_k + 1)$ recherches de caractères dans l'arbre lexicographique pour des chaînes de puissance d'un même lexème. Cette première approche en temps quadratique est écartée au profit de méthodes de complexité plus favorables exploitant l'existence de liens suffixes entre les nœuds [30]. En effet s'il existe un nœud étiqueté $a \cdot u$ ($a \in \Sigma, u \in \Sigma^*$) dans $\mathrm{ST}(T)$ (ce qui signifie l'existence du facteur au dans T), il existe également un nœud étiqueté u pour son suffixe direct : la forêt des nœuds enfants de u est l'union des forêts des nœuds enfants des nœuds $(x \cdot u)_{x \in \Sigma \cup \{\epsilon\}}$.

Nous décrivons ci-après succinctement un algorithme en temps linéaire proposé par McCreight [36]. Parmi d'autres algorithmes de construction, nous pouvons citer l'algorithme historique de Weiner [41] ainsi que celui plus récent d'Ukkonen [40] réalisant une construction incrémentale de l'arbre des suffixes. Un facteur logarithmique en la taille de l'alphabet manipulé doit être introduit dans la complexité temporelle de construction afin de trouver pour chaque nœud interne, le nœud enfant d'étiquette commençant par un lexème donné. Si l'alphabet manipulé est énumérable, l'algorithme de Farach [28] permet de s'abstraire de ce facteur logarithmique en construisant et fusionnant deux arbres de suffixes (suffixes de position paire et impaire) à l'aide d'un tri par bacs des lexèmes.

[1] L'arbre de suffixes non-compact comporte au maximum L feuilles pour les L suffixes (des chaînes partageant des suffixes communs, certaines occurrences de suffixes pouvant représenter des facteurs égaux). Dans le pire des cas, un arbre compact de $L = 2^k$ feuilles dont tous les nœuds sont d'arité sortante 2 comporte $2L - 1$ nœuds.

Algorithme de construction de McCreight

L'algorithme de McCreight [36] procède à la construction de l'arbre des suffixes compact en temps linéaire par ajout itératif des suffixes de la première à la dernière position. L'ajout du suffixe de position j d'une chaîne t_i ne requiert que la seule connaissance des caractères $t_i\,[j|k_j]$ avec k_j un nombre fini de caractères. On notera en fait que k_j est défini de telle sorte que le nœud $t_i\,[j|k_j-1]$ existe déjà dans l'arbre[2]. $t_i\,[j|k_j-1]$ existant déjà et ayant été créé lors de l'ajout d'un suffixe précédent (suffixe $t_i[l..]$ avec $t_i\,[l|k_j-1]=t_i\,[j|k_j-1]$), le nœud de chemin $t_i\,[j+1|k_j-2]$ a également été créé par l'insertion d'un suffixe précédent, $t_i[l+1..]$ en l'occurrence. Ainsi l'ajout du suffixe $t_i[j+1..]$ peut être réalisé en se rendant d'abord au nœud de chemin $t_i\,[j+1|k_j-2]$ par le suivi du lien suffixe de $t_i\,[l|k_j-1]$ puis en y ajoutant autant de lexèmes que nécessaire $c_{j+1}=k_{j+1}-(k_j-1)$ afin de créer effectivement un nouveau nœud dans l'arbre compact. La première étape est réalisée rapidement puisqu'il s'agit de suivre un lien suffixe en temps constant. Nous avons ainsi un nombre total de comparaisons linéaire $(\sum_{j\in[1..n]} c_j = n)$.

6.2.3 Recherche de facteurs répétés

Étant donné l'arbre de suffixes $\mathrm{ST}\,(T)$, nous déterminons pour chacun des nœuds p :

1. Le nombre d'occurrences de suffixes p pour lequel p est terminal, que nous notons $|t(p)|$.
2. Le nombre de nœuds enfants mono-feuille de p noté $|C_{\text{mono}}(p)|$, étant précisé qu'une mono-feuille q est définie par $|t(q)|=1$ et ne possède pas d'enfant.
3. Le nombre total de nœuds enfants de p noté $|C(p)|$.

Nous en déduisons une taxonomie des nœuds p de l'arbre de suffixes selon leurs valeurs $|t(p)|, |C_{\text{mono}}(p)|, |C(p)|$:

1. $|C(p)|=0$: p est une feuille
   (a) $|t(p)|=1$: p est une *mono-feuille* ne représentant qu'une occurrence de suffixe.
   (b) $|t(p)|>1$: p est une *multi-feuille* représentant plusieurs occurrences $t(p)$ du suffixe p.
2. $|C(p)|>0$: p est un nœud interne
   (a) $|C_{\text{mono}}(p)|=|C(p)|$: p est un nœud *faiblement interne* qui est le plus grand préfixe commun de ses mono-feuilles enfants $C_{\text{mono}}(p)$.
   (b) $|C_{\text{mono}}(p)|<|C(p)|$: p est un nœud *fortement interne*.

Cette taxonomie est appliquée sur l'arbre de suffixes présenté en figure 6.1.

L'ensemble des occurrences de suffixes atteignables depuis le sous-arbre de racine p est noté $\mathrm{suff}(p)$: il s'agit de tous les suffixes de T partageant le préfixe commun p.

[2]Nous désignons le nœud u comme le nœud atteint en suivant le chemin u depuis la racine. S'agissant d'un arbre compact, il est possible que le suivi de u aboutisse au « milieu d'une étiquette » d'un nœud uv feuille. Il est alors nécessaire de transformer le nœud uv en nœud u et de lui greffer un nœud de chemin relatif v : les nœuds u et uv coexistent alors.

Nous notons que les nœuds n'étant pas des mono-feuilles représentent des facteurs répétés non-extensibles sur la droite. Si un facteur p représenté par un nœud interne ou multi-feuille était extensible sur la droite avec le lexème a, celui-ci ne comporterait qu'un seul enfant d'étiquette pa ce qui contredit sa compacité. L'ensemble des nœuds internes et multi-feuilles de l'arbre de suffixes est donc un sur-ensemble de farmax(T). Il reste maintenant à vérifier que chacun des nœuds internes ne puisse être étendu sur la gauche.

Définition 6.5. *(Liens suffixes inverses)* Soit ST(T) l'arbre de suffixes de T muni de liens suffixes. Soit p un nœud (chemin) dans cet arbre. Nous notons $\mathcal{S}^{-1}(p)$ l'ensemble des nœuds connectés par un lien suffixe vers p. $\mathcal{S}^{-1}(p) = \{l_1 p, l_2 p, \cdots, l_k p\}$ avec $l_1, l_2, \cdots, l_k$ des lexèmes deux à deux distincts.

Ainsi le facteur répété p est extensible sur la gauche en ap ssi il n'existe qu'un unique nœud connecté par un lien suffixe à p : $\mathcal{S}^{-1}(p) = \{ap\}$ et que le sous-arbre de racine p ne comporte aucune feuille qui soit une occurrence de début de chaîne (une telle feuille ne pouvant admettre un lien suffixe pointant vers elle). L'existence d'un lexème a tel que $|\text{suff}(ap)| = |\text{suff}(p)|$ est équivalente à l'extensibilité de p sur la gauche.

Nous proposons la méthode suivante pour récupérer tous les farmax de T. Nous partons de chaque nœud p qui ne soit pas une mono-feuille de profondeur maximale en posant $l = \left|\mathcal{S}^{-1}(p)\right|$. Si $l = 1$ et que p ne comporte pas de feuille de début de chaîne, nous suivons la source du lien suffixe et ceci itérativement jusqu'à parvenir à un nœud p' ne respectant plus ces conditions : le chemin de ce nœud représente un facteur répété non-extensible à gauche. Il s'agit donc d'un farmax.

6.3 Table des suffixes et des plus long préfixes communs

6.3.1 Définitions

Définition 6.6. *(Ordre lexicographique)* Soit Σ l'alphabet des lexèmes manipulés muni d'un ordre total $<$. Nous définissons l'ordre lexicographique $<_{lex}$ dérivé de $<$ sur les chaînes de Σ^* ainsi : pour tout couple (u, v) de Σ^*, $u <_{lex} v$ ssi il existe $(x, y, z) \in \Sigma^{*3}$ avec la décomposition $u = x \cdot y$ et $v = x \cdot z$ telle que $y[1] < z[1]$ (par convention $\epsilon[1] = \epsilon$ et pour tout $a \in \Sigma$, $\epsilon \leq a$). x est le plus long préfixe commun de u et v.

Définition 6.7. *(Table des suffixes)* La table des suffixes SA(T) (en anglais *Suffix Array*) de l'ensemble des chaînes de lexèmes T est la suite d'occurrences de suffixes de T triées par ordre lexicographique (chaque occurrence de suffixe étant représentée par le couple (i, j), i étant l'indice de la chaîne t_i, j l'indice de début du suffixe dans la chaîne).

Définition 6.8. *(Table inverse des suffixes)* La table inverse des suffixes rSA(T) de l'ensemble des chaînes de lexèmes T est la table faisant correspondre chaque occurrence de suffixe $T_i[j]$ avec sa position dans la table de suffixes SA(T).

Il découle de la définition de la table des suffixes que si l'on considère un arbre de suffixes (dont les enfants de chaque nœud sont triées par premier lexème du nœud), la table des suffixes est obtenue par un parcours en profondeur de l'arbre en ne conservant que ses nœuds p avec $|t(p)| > 0$. Il est donc immédiat d'obtenir la table de suffixes à partir de l'arbre de

suffixes en temps linéaire du nombre de suffixes. Réciproquement, il est également possible de construire l'arbre des suffixes à partir de la table des suffixes (cf section 6.4). Nous étudions des algorithmes de construction directe de la table de suffixes, mémoriellement plus intéressants.

6.3.2 Construction de la table de suffixes

Algorithme original de doublement Manber et Myers introduisirent [35] la structure de table de suffixes avec un algorithme de construction de complexité temporelle en $O(L \log l_{\max})$ pour des chaînes de longueur cumulée L avec une chaîne de longueur maximale $l_{\max}$. Le principe de la construction repose sur le constat suivant : s'il est possible de trier tous les préfixes de suffixes de longueur k, nous pouvons considérer tous les préfixes de suffixes de longueur $2k$ comme les concaténations de deux préfixes consécutifs de suffixes de longueur k. Ainsi nous pouvons écrire la chaîne t comme la concaténation de chaînes de longueur k : $t = c_1 c_2 \cdots c_{|c|/k}$ ($|c_1| = |c_2| = \cdots = |c_{|c|/k}|$)[3]. Connaissant le classement lexicographique des facteurs $c_1, c_2, \cdots, c_{|c|/k}$, nous en déduisons en temps $O(|c|/k)$ le classement lexicographique des couples $c_1c_2, c_2c_3, \cdots, c_{|c|/k-1}c_{|c|/k}$ par un tri par bacs. En conséquence, nous obtenons le classement des préfixes de suffixes de longueur $2k$. Ce processus de doublement de la taille des préfixes de suffixes triés débute par le classement des lexèmes unitaires et se termine à l'itération f lorsque les préfixes de suffixes de taille 2^f sont tous distincts. La complexité temporelle est donc dépendante de la taille r des plus grands facteurs répétés ($O(L \log r), r \leq l_{\max}$).

Algorithmes en temps linéaire La table de suffixes d'un ensemble de chaînes (de longueurs cumulées) peut être construite simplement en temps $O(L)$ à partir du parcours des feuilles de l'arbre de suffixes correspondant construit lui-même en temps linéaire. La première méthode de construction directe en temps linéaire, dite *skew*, a été proposée par Kärkkäinen et Sanders [31]. Celle-ci procède en trois étapes principales : la première consiste à trier les deux tiers des suffixes de position $1 \mod 3$ et $2 \mod 3$ (on suppose le premier caractère d'indice 0) tandis que lors de la deuxième étape, les suffixes de position $0 \mod 3$ sont comparés avec l'ensemble des suffixes précédemment triés. Enfin, la troisième étape permet de trier par fusion les suffixes de position $\{1, 2\} \mod 3$ et les suffixes de position $0 \mod 3$. Le tri des suffixes est appliqué récursivement sur les suffixes de position $\{1, 2\} \mod 3$ en remplaçant ces suffixes par les 3-grams débutant à ces positions et en triant les suffixes de la chaîne ainsi obtenue. La connaissance de l'ordre des suffixes de positions $\{1, 2\}$ permet un tri et la fusion en temps $O(L)$ des suffixes de position $0 \mod 3$. Finalement l'algorithme peut être mené en temps $T(L) = O(L) + T(\frac{2}{3}L) = O(L)$ avec un tri par bacs des k-uplets de lexèmes ou k-grams. Cela implique l'utilisation d'un alphabet énumérable. Dans le cas contraire (qui ne nous intéresse cependant pas pour nos applications), le tri des suffixes nécessite une complexité temporelle minimale de $O(L \log L)$.

Comparaison des algorithmes D'autres algorithmes ont été proposés chacun avec leurs spécificités : privilégiant les chaînes avec des faibles répétitions, de longues répétitions ou tentant de réduire l'espace mémoire nécessaire pour la construction. L'algorithme *skew* de Kärkkäinen et Sanders [31] bien qu'étant de complexité linéaire dans le pire des cas ne permet pas les implantations les plus rapides, comme le montrent les tests présentés par [39]. Des

[3] Si $|t|$ n'est pas multiple de k, on pourra ajouter $\lceil |t|/k \rceil \times k - |t|$ lexèmes ϵ

algorithmes de complexité temporelle dans le pire des cas équivalente à un tri naïf des suffixes ($\Theta(n^2 logn)$) peuvent présenter des résultats expérimentaux temporellement et mémoriellement avantageux [37].

6.3.3 Table des plus longs préfixes communs

Afin de pouvoir localiser les suffixes présentant un préfixe commun de longueur importante (facteurs répétés), nous nous intéressons à la longueur des préfixes communs de suffixes consécutifs de la table des suffixes.

Définition 6.9. Soient deux chaînes u et v. w est le plus long préfixe commun de u et v ssi il existe des factorisations $u = wu'$ et $v = wv'$ avec soit $u' = \epsilon$, $v' = \epsilon$ ou $u'[1] \neq v'[1]$. On note $w = \mathrm{lcp}\,(u, v)$.

Définition 6.10. La table des plus long préfixes communs $\mathrm{LCP}(T)$ est la suite finie de taille $|\mathrm{SA}\,(T)|$ dont l'élément d'indice $i > 0$ est la longueur de $\mathrm{lcp}\,(\mathrm{SA}\,(T)\,[i-1], \mathrm{SA}\,(T)\,[i])$. Par convention $\mathrm{LCP}(T)[1] = 0$.

Une méthode de construction rapide de la table des LCP (introduite par Kasai et al. [32] et présentée en algorithme 1) nécessite la connaissance de la table de suffixes ainsi que de la table inverse de suffixes. L'idée consiste à non pas calculer la table par indice croissant mais par position croissante des suffixes dans leurs chaînes. Sans perte de généralité, nous considérons une unique chaîne de lexèmes u à partir de laquelle est construite la table (directe et inverse) des suffixes. $p = \mathrm{rSA}\,u[i]$ désigne la position du suffixe $u[i..]$ dans la table de suffixes : nous calculons ainsi $\mathrm{LCP}[p]$ par comparaison lexème par lexème de $\mathrm{SA}[p] = u[i..]$ ($p \geq 2$) avec $\mathrm{SA}[p-1] = u[h..]$ ($k+1$ comparaisons pour un plus long préfixe commun de taille k). Avec $p' = \mathrm{rSA}\,u[i+1..]$ (position du suffixe $u[i+1..]$), nous avons $\mathrm{LCP}[p'] \geq \mathrm{LCP}[p] - 1$. En effet $\mathrm{LCP}[p']$ est minimal lorsque le prédécesseur lexicographique de $u[i+1..]$ dans la table des suffixes est $u[h+1..]$: alors $\mathrm{LCP}[p'] = \mathrm{LCP}[p] - 1$. Nous pouvons ainsi calculer avec $2|u| - 1$ comparaisons de lexèmes au maximum toutes les valeurs de la table $\mathrm{LCP}[p]$ selon le même raisonnement que celui utilisé pour l'algorithme de construction d'arbre de suffixes de McCreight (cf sous-section 6.2.2).

La table des LCP peut être utilisée pour localiser les couples d'occurrences de plus longs facteurs répétés et donc mettre rapidement en évidence les clones les plus substantiels. La recherche exhaustive des facteurs répétés requiert elle le groupement de toutes les occurrences d'un facteur répété pour garantir la complétude : celle-ci peut être menée à partir de la table de suffixes aidée de la table des LCP afin de construire l'arbre des intervalles. Chacun des intervalles de cet arbre correspond à un intervalle de suffixes consécutifs dans la table de suffixes et ainsi à un nœud interne de l'arbre des suffixes.

6.4 Arbre des intervalles et structures associées

6.4.1 Définitions

Définition 6.11. *(Généralisation du LCP)* Nous généralisons le LCP (plus long préfixe commun) à un ensemble de chaînes de caractères $U = \{u_1, u_2, \cdots, u_n\}$. $l = \mathrm{LCP}(U)$ est le (plus

```
   Données : SA (T) , rSA (T)
   Résultat : LCP (T)
 1 début
 2 |   pour 1 ≤ i ≤ |T| faire
 3 |   |   k = 0 ;
 4 |   |   pour 1 ≤ j ≤ |t_i| faire
 5 |   |   |   p = rSA (T) [(i, j)] ;
 6 |   |   |   tant que p ≥ 2 ∧ SA (T) [p] = SA (T) [p − 1] faire
 7 |   |   |   |   k = k + 1 ;
 8 |   |   |   LCP(T)[p] = k ;
 9 |   |   |   k = max(0, k − 1) ;
10 fin
11 retourne LCP (T)
```

Algorithme 1 : Calcul linéaire de la table des LCP pour les chaînes $T = \{t_1, t_2, \cdots, t_n\}$

long) préfixe commun à $u_1, u_2, \cdots, u_n$ tel qu'il n'existe pas de caractère $a \in \Sigma$ avec $l \cdot a$ préfixe commun de $u_1, u_2, \cdots, u_n$.

Définition 6.12. *(Arbre des intervalles)* L'arbre des intervalles IT (T) (en anglais, *Interval Tree*) de l'ensemble de chaînes de lexèmes T est un arbre dont les nœuds représentent tous les intervalles $[i..j]$ de SA (T) ayant pour propriété de partager un préfixe commun de suffixe (LCP). Nous associons à chaque nœud son LCP $\mathcal{L}$ correspondant à LCP($\{$SA (T) $[i]$, SA (T) $[i+1], \cdots,$ SA (T) $[j]\}$). L'intervalle (i, j) ne doit pas être extensible, i.e. LCP(SA (T) $[i-1]$, SA (T) $[i]$) $\neq \mathcal{L}$ et LCP(SA (T) $[j]$, SA (T) $[j+1]$) $\neq \mathcal{L}$. $[i'..j'], \mathcal{L}'$ est enfant direct de $[i, j], \mathcal{L}$ ssi $[i'..j'] \subset [i..j]$ (condition de filiation) sans qu'il existe un nœud $[i''..j''], \mathcal{L}''$ tel que $[i''..j''] \subset [i..j]$ et $[i'..j'] \subset [i''..j'']$ (condition de filiation directe). Nous avons $\mathcal{L} \in$ pref $(\mathcal{L}')$.

L'arbre des intervalles d'un ensemble de chaînes de T est construit à partir de la table de suffixes [25] et peut être vu comme homomorphe à l'arbre de suffixes de T auquel on aurait supprimé ses mono-feuilles. En effet, chaque nœud interne ou multi-feuille de l'arbre des suffixes désigne un préfixe commun $\mathcal{L}$ d'au moins deux suffixes de T : ce préfixe commun peut être désigné dans la table de suffixes correspondante comme un intervalle de LCP $\mathcal{L}$. La longueur de cet intervalle représente le nombre d'occurrences de suffixes partageant ce même LCP.

Définition 6.13. La table des intervalles les plus profonds dIT (T) (*Deepest Interval Table*) de l'ensemble des chaînes de lexèmes T est la table associant à chaque occurrence de suffixe de T (représentée par son rang k dans la table des suffixes SA (T)) le nœud $([i..j], \mathcal{L})$ de l'arbre des intervalles tel que $k \in [i..j]$ et qu'il n'existe pas d'autre nœud $([i'..j'], \mathcal{L}')$ tel que $k \in [i'..j']$ avec $\mathcal{L}$ préfixe propre de $\mathcal{L}'$.

dIT(k) désigne l'intervalle de plus forte profondeur contenant le suffixe de rang k. Cette structure peut être utilisée, avec l'aide de la table inverse de suffixes, pour rechercher les occurrences d'un facteur répété débutant à une certaine position d'une chaîne indexée. En pratique, nous l'utiliserons pour l'établissement des liens suffixes de l'arbre des intervalles.

6.4.2 Construction de l'arbre des intervalles

Nous construisons l'arbre des intervalles par parcours itératif de la table des LCP à l'aide d'une pile (cf algorithme 2) : cette pile S contient la branche de l'arbre des intervalles en cours d'examen. Ne sont alors connus que la valeur de LCP des intervalles de la pile ainsi que l'indice de départ. Lorsque nous examinons LCP[i], nous comparons LCP[i] avec la longueur du LCP de l'intervalle $S.\,\text{sommet}$. Différents cas peuvent alors être rencontrés :

1. LCP[i] $< S.\,\text{sommet}\,.\,\text{lcp}$. L'intervalle de plus forte profondeur courant est alors de plus fort LCP que le LCP[i] : l'intervalle courant est donc clos à l'indice précédent $i-1$. On dépile l'intervalle et on répète l'opération jusqu'à ce que le LCP de l'intervalle de haut de pile soit inférieur ou égal à LCP[i]. Il est ensuite nécessaire de créer un nouvel intervalle de LCP LCP[i] contenant les suffixes de rang i et $i-1$. Deux sous-cas sont alors à envisager :
   (a) LCP[i] $= S.\,\text{sommet}\,.\,\text{lcp}$. Un intervalle existant possède le même LCP : celui-ci continue donc au rang i et l'on met à jour son indice de fin en conséquence.
   (b) LCP[i] $> S.\,\text{sommet}\,.\,\text{lcp}$. Un nouvel intervalle doit être créé de LCP égal à LCP[i]. Celui-ci possède pour parent $S.\,\text{sommet}$ et pour enfant le dernier intervalle retiré lors de la phase de dépilement (noté l). Le nouvel intervalle débute avec son premier intervalle enfant l.
2. LCP[i] $= S.\,\text{sommet}\,.\,\text{lcp}$. Un intervalle existant se prolonge sans dépilement préalable.
3. LCP[i] $> S.\,\text{sommet}\,.\,\text{lcp}$. Un nouvel intervalle débutant au rang $i-1$ et sans enfant est créé. Il possède pour parent $S.\,\text{sommet}$.

La table des suffixes avec les valeurs de LCP associées ainsi que l'arbre des intervalles correspondant avec la table des intervalles les plus profonds pour les chaînes exemples α, β et γ sont présentés en figure 6.1.

6.4.3 Ajout de liens suffixes à l'arbre des intervalles

Intérêt des liens suffixes L'arbre des intervalles IT(T) étant homomorphe à l'arbre des suffixes ST(T) privé de ses mono-feuilles (occurrences de facteurs), la recherche des facteurs répétés y est similaire, les intervalles étant des facteurs répétés non-extensibles à droite. Il est cependant important de munir l'arbre des intervalles de liens suffixes afin de savoir, lors de l'examen de chaque intervalle, si le facteur répété candidat qu'il représente présente une non-extensibilité sur la gauche : nous rappelons que le facteur répété u d'intervalle $[a..b]$ est extensible sur la gauche au facteur répété au d'intervalle $[a'..b']$ ssi un seul lien suffixe pointe sur u (au) et que leur cardinalité est identique ($b-a+1 = b'-a'+1$).

Détermination du lien suffixe d'un intervalle Pour l'intervalle $I = ([a..b], l)$ (avec $l \geq 2$) représentant le facteur répété au qui inclut les occurrences de positions $\{(i,j) \in \text{SA}[k]\}_{a \leq k \leq b}\}$, nous notons que l'intervalle suffixe de I contient au moins tous les suffixes directs des occurrences de I, à savoir l'ensemble $\{(i, j+1)/(i,j) \in \text{SA}[k]\}_{a \leq k \leq b}\}$. L'intervalle suffixe direct u de au est représenté par l'intervalle $I' = ([a'..b'], l-1)$. Pour localiser cet intervalle, nous cherchons d'abord deux occurrences de suffixe dont le préfixe commun est de longueur l. Ceci peut être réalisé en choisissant deux suffixes de rangs $k, k' \in [a..b]$ tels que $|\text{LCP}(\text{SA}[k], \text{SA}[k'])| = l$. k et k' peuvent prendre pour valeur les bornes a et b de l'intervalle.

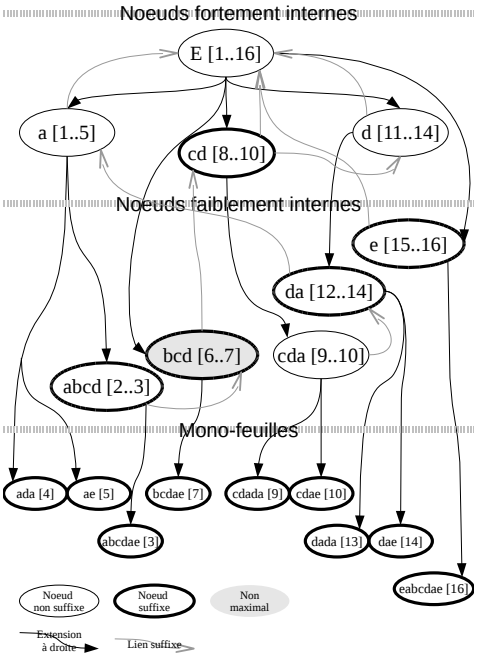
Données : Table LCP
Résultat : Arbre des intervalles IT, table des plus profonds intervalles dIT

```
1  début
2      S ← pile vide ;
3      r ← r = ([1..L]) ;
4      Empiler r (intervalle racine) sur S ;
5      pour 2 ≤ i ≤ L + 1 faire
6          lcp ← LCP[i] (LCP[L + 1] = −1);
7          l ← ∅ ;
8          tant que |S| > 0 ∧ lcp < (S.sommet).lcp faire
9              Nous descendons vers la racine de la branche d'intervalles ;
10             (S.sommet).fin ← i − 1 ;
11             si l non-défini alors
12                 l ← S.sommet ;
13                 Dépiler S.sommet ;
14             l.parent ← S.sommet ;
15             l ← S.sommet ;
16             Dépiler S.sommet ;
17         si |S| > 0 ∧ lcp = (S.sommet).lcp alors
18             L'intervalle de haut de pile continue ;
19             (S.sommet).fin ← i ;
20         si |S| > 0 ∧ lcp > (S.sommet).lcp alors
21             si l défini alors
22                 b ← l.début ;
23             sinon
24                 b ← i − 1 ;
25             Empiler ([b..i], LCP[i]) sur S ;
26             si l défini alors
27                 l.parent ← S.sommet ;
28         dIT[i − 1] ← S.sommet ;
29     retourne r, dIT ;
30 fin
```

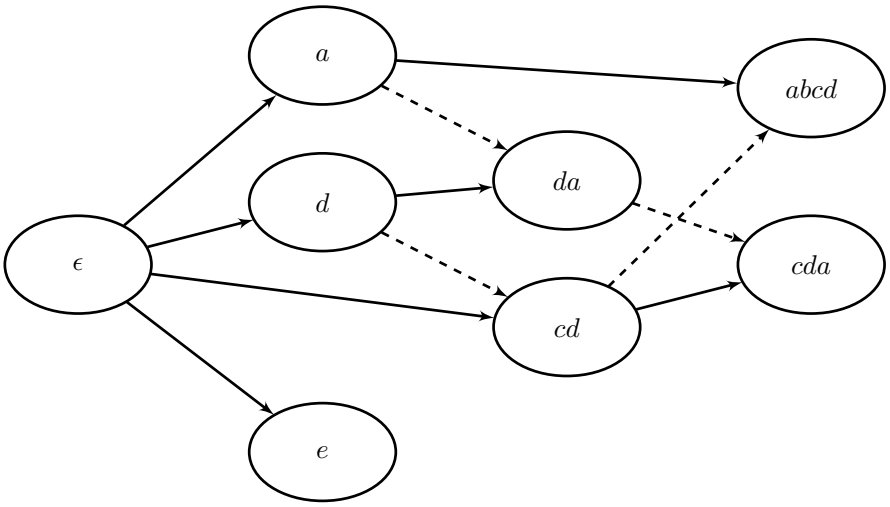
Algorithme 2 : Génération de l'arbre des intervalles et la table des plus profonds intervalles à partir de la table des LCP

Rang	Suffixe	LCP	dIT
1	$\beta[5..] = a$		[1..5]
2	$\alpha[1..] = abcd$	1	[2..3]
3	$\alpha[2..] = abcdae$	4	
4	$\beta[3..] = ada$	1	[1..5]
5	$\gamma[6..] = ae$	1	
6	$\alpha[2..] = bcd$	0	[6..7]
7	$\gamma[3..] = bcdae$	3	[6..7]
8	$\alpha[3..] = cd$	0	[8..10]
9	$\beta[1..] = cdada$	2	[9..10]
10	$\gamma[4..] = cdae$	3	
11	$\alpha[4..] = d$	0	[11..14]
12	$\beta[4..] = da$	1	[12..14]
13	$\beta[2..] = dada$	2	
14	$\gamma[5..] = dae$	2	
15	$\gamma[7..] = e$	0	[15..16]
16	$\gamma[1..] = eabcdae$	1	

(a) Tables des suffixes, table des LCP et table des intervalles les plus profonds



(b) Arbre de suffixes de T (l'arbre des intervalles est obtenu en supprimant les mono-feuilles). Seuls les liens suffixes entre nœuds de l'arbre des intervalles sont indiqués.



(c) Graphe des farmax de T. Les arcs d'extension à droite sont pleines, celles à gauches sont en tirets.

FIG. 6.1 – Table et arbre des suffixes, LCPT, dIT et graphe de farmax pour $T = \{abcd, cdada, eabcdae\}$

Les suffixes directs de SA[k] et de SA[k'] sont ensuite obtenus par la table de suffixes : il s'agit des suffixes de positions $(i, j+1)$ avec $(i, j) = \text{SA}[k]$ et $(i', j'+1)$ avec $(i', j') = \text{SA}[k']$. Les rangs des suffixes α et β des suffixes $T_i[j+1..]$ et $T_{i'}[j'+1..]$ sont ensuite récupérés par la table inverse des suffixes. Il existe un unique intervalle contenant les suffixes de rang α et β et de LCP $l-1$: cet intervalle est le plus profond contenant ces suffixes car leur LCP est de $l-1$. Il peut être obtenu par détermination de plus profond ancêtre commun aux intervalles les plus profonds contenant les rangs α et β, à savoir dIT(α) et dIT(β). Cet intervalle recherché I' est calculable en utilisant une des méthodes de détermination de *Lowest Common Ancestor* (LCA) telles que décrites ultérieurement en 11.2.3, certaines permettant la récupération du LCA en temps constant après une étape de pré-calcul linéaire. Finalement la détermination de tous les intervalles suffixes est réalisable en temps $\Theta(|\text{IT}|)$.

6.4.4 Graphe des farmax

Le graphe des farmax est une structure modélisant les relations d'extension des facteurs répétés maximaux et complets. Pour l'arbre des intervalles, un intervalle I parent de l'intervalle J modélise un facteur répété complet et son extension sur la droite (réduisant le nombre d'occurrences incluses : $|I| > |J|$). D'autre part, les liens suffixes inverses explicitent des extensions d'un lexème à gauche des facteurs répétés. Cependant demeurent des facteurs répétés qui sont extensibles sur la gauche : ainsi pour l'exemple présenté précédemment *bcd* est extensible sur la gauche en *abcd* et n'est donc pas un farmax.

Définition 6.14. *(Graphe des farmax)* Le graphe des farmax d'un ensemble de chaînes de lexèmes T est composé pour sommets de tous les facteurs répétés maximaux (farmax) de T, augmenté de deux types d'arcs : les arcs d'extension droite et gauche. Le farmax u est lié au farmax v ($u \neq v$) par un arc droit (resp. gauche) ssi u est le préfixe (resp. suffixe) le plus direct de v, i.e. il n'existe pas w tel que u soit préfixe (resp. suffixe) de w avec $|w| < |v|$.

Le graphe des farmax se déduit simplement de l'arbre des intervalles muni de liens suffixes. Dans un premier temps, nous supprimons les facteurs répétés n'étant pas maximaux : nous agglomérons les facteurs répétés liés par une chaîne de liens suffixes inverses en un nœud représentant le plus long facteur répété de cette chaîne. Par exemple la chaîne de $k+1$ facteurs répétés ($a_1, a_2 \cdots a_k$ étant des lexèmes) $u : [a_0..b_0] \longrightarrow a_1u : [a_1..b_1] \longrightarrow \cdots \longrightarrow a_k \cdots a_1u : [a_k..b_k]$ avec $b_0 - a_0 + 1 = b_1 - a_1 + 1 = \cdots = b_k - a_k + 1$ est remplacée par un unique farmax $a_k \cdots a_1u : [a_k..b_k]$. Les liens parent-enfant de l'arbre des intervalles représentent les arcs d'extension droite tandis que les liens suffixes inverses restants sont les arcs d'extension gauche.

On remarque que le graphe des farmax peut également être construit à partir de l'arbre des intervalles des chaînes de T ainsi que l'arbre des intervalles de l'ensemble de leurs inverses $\tilde{T}$. Les facteurs répétés de T sont confondus avec leur inverses respectifs : les arcs de $\text{IT}(T)$ sont les arcs d'extension droite tandis que celles de $\text{IT}\left(\tilde{T}\right)$ représentent les arcs d'extension gauche.

Nous présentons en figure 6.1 le graphe des farmax de l'exemple introduit en début de chapitre.

6.5 Décomposition des chaînes t_k en facteurs maximaux de $T_{\leq k}$

Nous introduisons dans le chapitre 7 une méthode de factorisation de fonctions représentées sous la forme de chaînes de lexèmes. Concrètement, nous souhaitons, pour chaque fonction t_i d'un ensemble de fonctions T triées par longueur croissante, trouver tous les facteurs répétés comprenant au moins un facteur de t_i et un facteur de $T_{\leq i}$ où $T_{\leq i}$ est l'ensemble des fonctions dont la longueur est inférieure (ou égale avec un indice inférieur ou égal à i) à celle de t_i. Ceci nous permet de factoriser chaque fonction t_k par des fragments non-extensibles de fonctions de $T_{\leq i}$. Lorsqu'un facteur f_i est utilisé pour factoriser t_k, celui-ci doit comporter deux occurrences non-chevauchantes dans t_k.

Définition 6.15. La décomposition de la chaîne t_k en facteurs maximaux de $T_{\leq k}$ est l'ensemble des occurrences de tous les facteurs f_i de t_k respectant les deux conditions suivantes :

1. f_i est un facteur d'une chaîne de $T_{<k}$ ou f_i comporte deux occurrences non-chevauchantes dans t_k.
2. pour chaque f_i il n'existe aucun f_j respectant la première condition avec f_i facteur de f_j (condition de non-extensibilité).

6.5.1 Méthode de décomposition

Plutôt que de nous intéresser à la recherche de l'ensemble des occurrences non-extensibles facteurs de $T_{\leq k}$, nous souhaitons ne récupérer que l'occurrence provenant de la chaîne d'indice minimal. En cas d'égalité d'indice de chaîne, les occurrences sont départagées par la sélection de l'occurrence de position minimale dans la chaîne. Nous notons $I.\min$ cette occurrence pour le farmax I (confondu avec son intervalle).

Pour chaque intervalle $I = ([a..b], l)$ du graphe des farmax avec $I.min = \Lambda_{\min}[\lambda_{\min}|l]$, nous souhaitons déterminer s'il existe des occurrences de facteurs dans des chaînes d'indice Λ_k supérieure ou égale à $\Lambda_{\min}$ pouvant être utilisées pour la décomposition de t_{Λ_k}. Nous associons l'occurrence de facteur $\mathrm{SA}[k] = \Lambda_k[\lambda_k|l]$ ($k \in [a..b]$) à $I.\min$ ssi :

1. $\Lambda_k > \Lambda_{\min}$ ou $\Lambda_k = \Lambda_{\min}$ avec $\beta_{\min} + l - 1 < \beta_k$ (ce qui signifie que l'occurrence de rang k et l'occurrence minimale ne s'intersectent pas) ;
2. et $\Lambda_k[\lambda_k|l]$ n'a pas déjà été associé à une occurrence de facteur de longueur plus importante (i.e. pour un farmax de LCP plus grand, étendu sur la gauche et/ou la droite).

Ainsi pour chaque intervalle, toutes les occurrences propres (non incluses dans un farmax plus long) sont associées à l'occurrence minimale à l'exception de l'occurrence minimale elle-même qui reste non-associée. Celle-ci fera l'objet d'une association dans un farmax préfixe ou suffixe plus court. En cas de chevauchement, nous notons v ($|v| = \lambda_{\min} + l - \lambda_k > 0$) la portion intersectant l'occurrence minimale ainsi que l'occurrence distincte de même chaîne : le farmax est donc la forme $vuvuv$. Deux associations non-chevauchantes peuvent alors être réalisées : soit sur le préfixe vu du farmax, soit sur son suffixe uv.

Nous en déduisons l'algorithme 3 qui requiert un temps linéaire en la taille de la table de suffixes.

Données : $\mathcal{F}(T)$: graphe des farmax de T
Résultat : Factorisation de t_k dans $T_{<k}$: $\{e_1, e_2, \cdots, e_\zeta\}$

début
  pour $I = ([a..b], l) \in \{$ *ensemble des farmax de* T *triés par longueur décroissante* $\}$ **faire**
    $I_{\neg a}$ *est l'ensemble des occurrences non-associées de* I ;
    Cet ensemble comprend au moins $I.min$;
    $I_{\neg a} \longleftarrow I_{\neg a} + \{I.min\}$;
    Boucle d'association entre occurrences propres du farmax et occurrence minimale ;
    pour $k \in [a..b]$ *avec soit* k *représentant une occurrence propre, soit* $k \in I_{\neg a}$ **faire**
      si $\alpha_k > I.\alpha_{\min} \vee (\alpha_k = I.\alpha_{\min} \wedge 0 \leq c < l)$ **alors**
        si $\alpha_k \neq I.\alpha_{\min}$ **alors**
          Aucun chevauchement entre les deux occurrences ;
          $c \longleftarrow 0$;
        sinon
          Calcul des lexèmes de chevauchement pour $\alpha_k = I.\alpha_{\min}$
          $c \longrightarrow \beta_{\min} + l - \beta_k$;
        Ajout des associations $\alpha_k\,[\beta_k|l-c] \longrightarrow I.\alpha_{\min}\,[I.\beta_{\min}|l-c]$ (ainsi que de l'association $\alpha_k\,[\beta_k + c|l-c] \longrightarrow I.\alpha_{\min}\,[I.\beta_{\min} + c|l-c]$ pour $c > 0$) ;
      sinon
        Ajout de l'occurrence non-associée à $I'_{\neg a}$ où I' sont les farmax immédiatement préfixe et immédiatement suffixe ;
fin

Algorithme 3 : Algorithme linéaire de recherche des factorisations de t_k dans $T_{<k}$

6.5.2 Exemple de décomposition

Nous présentons un exemple de décomposition en facteurs sur les chaînes $T = \{t_1 = \alpha = abcd, t_2 = \beta = cdada, t_3 = \gamma = eabcdae\}$ dont nous avons précédemment déterminé l'arbre et la table de suffixes ainsi que l'arbre des intervalles et le graphe des farmax. Nous obtenons la table d'association des occurrences de facteurs suivante :

Facteur	Occurrence dans t_i	Occurrence dans $T_{\leq i}$
$abcd$	$\gamma[2..5]$	$\alpha[1..4]$
cda	$\gamma[4..6]$	$\beta[1..3]$
da	$\beta[4..5]$	$\beta[2..3]$
a	$\gamma[6]$	$\alpha[1]$
e	$\gamma[1]$	$\gamma[7]$

Nous pouvons ainsi écrire chaque chaîne t_i en la décomposant en facteurs de $T_{<i}$:

$$\begin{aligned} t_2 = \beta = &\quad cda \cdot (da : \beta[2..3]) \\ t_3 = \gamma = &\quad e \cdot (abcd : \alpha[1..4]) \cdot (a : \alpha[1]) \cdot (e : \gamma[1]) \\ &\qquad (cda : \beta[1..3]) \end{aligned}$$

Nous notons que pour l'exemple présenté, β est décomposable en facteurs non-chevauchants ce qui n'est pas le cas de γ où les facteurs de décomposition $abcd$ et cda se chevauchent. Il est ainsi nécessaire, si des facteurs non-chevauchants sont souhaités, d'utiliser une méthode de couverture de la chaîne par de tels facteurs comme cela sera étudié en section 7.2).

6.6 Arbre ou table de suffixes ?

Nous avons précédemment étudié les structures d'indexation de suffixes que sont les arbres et tables de suffixes pour introduire ensuite la structure nouvelle de graphe de farmax adaptée à l'approche de factorisation de chaîne que nous souhaitons mener sur des fonctions de lexèmes. Nous nous interrogeons maintenant sur les modalités implantatoires pratiques de ces structures. Doit-on privilégier arbre ou table de suffixes dans le cadre de la recherche de facteurs répétés pour des outils de détection de clones ? Nous passons en revue quelques améliorations et optimisations existantes dans la construction et l'utilisation de ces structures afin notamment d'économiser l'espace mémoire ou de paralléliser le processus d'indexation des suffixes afin de permettre le traitement de volumes importants de données.

6.6.1 Complexité mémorielle

Pour un ensemble de chaînes de longueur totale L lexèmes, une table de suffixe, représentée par une séquence de L entiers pour désigner chaque suffixe, nécessite un espace de $L \log_2 L$ bits (à comparer aux $L \log_2 \Sigma$ bits nécessaires pour stocker les chaînes originales non-compressées). Implanter un arbre de suffixes nécessite, outre le stockage des feuilles représentant chaque suffixe, le maintien en mémoire de l'ensemble des nœuds internes. Chaque nœud requiert l'indication de son nœud suffixe ainsi que des nœuds enfants (sous forme de liens premier fils et frère droit) et de l'étiquette (deux entiers de position dans les chaînes) pour les nœuds internes ce qui entraîne l'utilisation, d'au plus $5 \log_2 L$ bits par nœud soit $10L \log_2 L$ bits pour l'ensemble de l'arbre compact comportant au plus $2L - 1$ nœuds. On notera que toutefois, pour la majorité des applications pratiques d'une structure d'indexation de suffixes (dont la

recherche de facteurs répétés), si l'arbre des suffixes est autosuffisant, la table de suffixe est en elle-même inconsistante car il faut lui adjoindre l'arbre des intervalles avec liens suffixes (soit au plus $5 \log_2 L$ bits par facteur répété) dont la construction repose sur la présence de la table des LCP et la table de suffixes inverse. En pratique la seule construction de la table de suffixes, pour la méthode par doublement, requiert en plus une table inverse temporaire ainsi que deux tableaux de booléens, soit au total $2L(\log_2 L + 1)$ bits.

Dans le cadre de la recherche de facteurs répétés pour l'exploration de similitudes dans du code source représenté par séquences de lexèmes, la question du mode de stockage de la structure d'indexation de suffixes se pose. Si l'indexation est incrémentale, la structure d'arbre de suffixes stocké sur une mémoire de masse devient incontournable. Si un jeu de projets fixe doit être analysé, une table de suffixes avec maintien de la structure en mémoire centrale est plus avantageux (même si une mémorisation sur une mémoire de masse est possible). Pour ces deux possibilités, il est intéressant de rechercher des méthodes pour réduire la consommation mémorielle. Pour un arbre de suffixes stocké sur une mémoire de masse, la localité des nœuds et des nœuds suffixes est importante pour réduire les accès disque lors de l'ajout incrémental de chaînes.

Réduction de l'espace mémoire requis

Arbres de suffixes Concernant les arbres de suffixes, Kurtz [33] propose un classement des différents nœuds de l'arbre en petits et grands nœuds. Ainsi un nœud interne xu ($x \in \Sigma, u \in \Sigma^*$) est un nœud large ssi la position de la deuxième occurrence du facteur u dans les chaînes n'est pas précédée par x. Dans le cas contraire, xu est un nœud étroit. Certains champs comme le lien suffixe ou la position du suffixe peuvent être omis et déduits d'après un nœud large. Expérimentalement, cette méthode permet de diviser par un facteur 2 la taille moyenne des nœuds. D'autres structures d'indexation de suffixes dérivées d'arbres de suffixes [34] occupent approximativement le même espace mémoire mais au prix d'une flexibilité moindre.

Tables de suffixes Les algorithmes de construction de tables de suffixe se contentent en général d'un espace mémoire restreint, peu supérieur à la mémoire finale nécessaire de $2N \log N$ pour la conservation de la table et de son inverse pour des chaînes de longueur cumulée N. La table des suffixes, son inverse et la table des LCP permettent la construction de l'arbre des intervalles puis du graphe des farmax. Combiné à la table de suffixes, ce graphe nécessite un espace mémoire largement inférieur à celui d'un arbres de suffixes car les suffixes mono-feuilles non-partagés ainsi que les facteurs répétés extensibles n'y figurent pas.

Stockage sur disque et parallélisation Lorsque la longueur cumulée N des chaînes manipulées est trop importante pour stocker les structures d'indexation en mémoire centrale, il est possible d'opter pour un stockage sur mémoire de masse [26]. Néanmoins le faible caractère local des opérations d'accès des algorithmes exploitant des liens suffixes limite l'intérêt d'un tel stockage, sauf à exploiter une structure incrémentale. Si l'on souhaite exploiter une structure d'indexation à usage unique, il est préférable alors d'exploiter la mémoire centrale d'une grappe de machines afin d'y distribuer les calculs. On pourra par exemple confier à chaque machine le calcul d'une structure d'indexation pour les suffixes commençant par un préfixe donné [27], en distribuant astucieusement la charge, sans toutefois de gain en complexité temporelle.

6.6.2 Incrémentalité de la construction

Nous discutons de la nature incrémentale des structures d'indexation de suffixes : celle-ci s'avère indispensable lorsque la structure doit être utilisée pour la constitution d'une base de projets évolutive. Une structure non-incrémentale telle que la table de suffixes nous limite à l'étude d'un jeu fixe de projets lexemisés.

Incrémentalité de l'arbre des suffixes

La structure d'arbre de suffixes est naturellement incrémentale : il est possible de l'étendre par l'ajout de nouvelles chaînes de lexèmes. De la même façon, il est également envisageable de supprimer une chaîne u. Nous citons deux méthodes afin de réaliser cette opération. La première consiste à parcourir l'ensemble des feuilles de l'arbre et à modifier (pour les multi-feuilles) ou supprimer[4] (pour les mono-feuilles) celles concernant les occurrences de suffixes de la chaîne. Cette opération est néanmoins coûteuse car nécessitant le parcours de toutes les feuilles. Elle peut être accélérée par l'utilisation d'une table construite lors de l'ajout de la chaîne dans l'arbre, liant chaîne et feuilles des occurrences de suffixe de la chaîne. Cette solution est malheureusement coûteuse en espace. Une deuxième approche consiste à chercher la feuille correspondant au suffixe $u[i..] = xyz$ ($x \in \Sigma, y, z \in \Sigma^*$) et son nœud interne prédécesseur xy. Le lien suffixe de xy vers y permet de trouver le nœud interne prédécesseur de $u[i+1..]$ et ainsi sa feuille correspondante.

L'ajout ou la suppression d'une chaîne de lexème s'avère en pratique utile pour ajouter ou supprimer un projet d'une base d'indexation. Si un projet est modifié et que l'on souhaite conserver dans la structure d'indexation la version la plus récente, on peut alors supprimer la version plus ancienne pour ensuite ajouter celle plus récente. Une démarche plus astucieuse à explorer pourrait utiliser le jeu de différences entre les deux versions pour modifier directement l'arbre de suffixes.

Incrémentalité de la table des suffixes

La table de suffixes est une structure d'indexation qui n'est pas aisément modifiable. L'ajout des suffixes d'une chaîne u dans la table des suffixes de chaînes de T nécessite soit le recalcul complet de la table pour $T + \{u\}$, soit la fusion des tables de suffixes de T et u avec une approche naïve en complexité temporelle dans le pire cas en $O(|t_1| + |t_2| + \cdots + |t_n| + |u|)$. La table de suffixes est donc une structure qui ne se prête qu'à l'indexation d'un jeu fixe de projets. Il reste néanmoins possible comme présenté par [29], en utilisant une structure de liste de suffixes doublement chaînée, de remplacer un facteur par un nouveau lexème sans nécessiter un recalcul complet de la liste ainsi que des structures annexes [38] (table de LCP et arbre des intervalles).

[4] On notera que lors de la suppression d'une mono-feuille, il peut être nécessaire, afin de conserver la compacité de l'arbre, de transformer le nœud interne prédécesseur en feuille si son arité atteint 1 en lui intégrant l'étiquette de sa désormais feuille unique.

7

Factorisation de chaînes de lexèmes

Sommaire

7.1 Aperçu général

7.1.1 Problématique et motivations

La recherche de similarité sur un ensemble de chaînes de lexèmes peut être menée soit par des méthodes de comparaison globale, soit par la recherche de similitudes sur toutes les paires de l'ensemble. Dans le second cas la complexité temporelle est plus défavorable même si cela permet l'usage de méthodes algorithmiques tel que l'alignement de séquences (cf chapitre 5) permettant la recherche de correspondances approchées. Pour la comparaison globale, des structures d'indexation de suffixes (cf chapitre 6) permettent la recherche de groupes de facteurs répétés et donc de groupes de correspondances exactes. Toutefois la recherche de facteurs répétés bruts comporte certaines limitations : elle ne tient notamment pas compte de relations d'inclusion ou de chevauchement entre occurrences reportées. En effet, si xyz participe aux facteurs répétés xy et yz, il serait intéressant de considérer le chevauchement de ces deux facteurs : y. La factorisation doit donc être réalisée à plusieurs niveaux.

D'autre part, il serait avantageux de pouvoir localiser des similarités et déduire une métrique afférente offrant une résistance aux opérations de factorisation et de développement de fonctions. À cet effet, nous considérons comme unité syntaxique de comparaison la fonction. Deux fonctions doivent être considérées comme similaires si celles-ci permettent l'exécution de la même séquence d'instructions, quelles que soient les fonctions intermédiaires appelées.

Dans un premier temps, nous cherchons à obtenir un graphe d'appels factorisé de l'ensemble des projets traités. Le graphe d'appels factorisé sera ensuite analysé afin de déduire une métrique de similarité entre fonctions basée sur leur couverture de code commune.

7.1.2 Quelques définitions préalables

Définition 7.1. *(Lexèmes primitifs et lexèmes d'appel)* Une fonction est représentée par une séquence de lexèmes composée de lexèmes primitifs et de lexèmes d'appel. Les lexèmes primitifs (Σ^p) sont dérivés du lexique du langage traité tandis que les lexèmes d'appel (Σ^c) représentent chacun un appel vers une fonction spécifique du projet.

L'ensemble des projets étudiés est ainsi préalablement représenté sous la forme de fonctions de séquences de lexèmes primitifs et d'appel, ce qui implique la connaissance du graphe d'appels des projets (cf 3.2.3). Une analyse syntaxique légère est donc nécessaire pour le découpage des projets en fonctions et une résolution des appels de fonctions doit être menée. Cette étape peut être rendue plus difficile voire impossible par l'usage de certains procédés d'obfuscation visant à masquer les liens d'appel (cf 4.9.1).

Les lexèmes primitifs peuvent correspondre aux différents termes du lexique du langage avec un certain niveau d'abstraction. Généralement les identificateurs et types sont abstraits afin de gérer les obfuscations à base de renommage de types ou de variables. Certains mots-clés peuvent être confondus par un unique représentant. D'autres mots-clés, méta-informatifs, peuvent être également tout simplement ignorés (comme par exemple les modificateurs de visibilité de certains langages).

Les liens d'appel vers des fonctions de bibliothèque, i.e. des fonctions dont l'implantation n'est pas présente au sein des projets, sont représentés par un lexème primitif spécifique à chaque fonction appelée. Il est en effet nécessaire de limiter le processus de factorisation au niveau des fonctions de projet dans un souci de maîtrise de la complexité. Il n'est néanmoins pas inimaginable qu'un obfuscateur puisse développer un appel d'une fonction de bibliothèque par le code source de son implantation lorsque celui-ci est disponible.

L'utilisation d'un modèle unidimensionnel par séquences de lexèmes primitifs et d'appels est gênant pour la manipulation d'appels de fonction imbriqués. L'arbre des appels de fonction correspondant à une expression est donc préalablement linéarisé par des instructions simples d'affectation des appels imbriqués à des variables temporaires. Ainsi l'expression $a = min(f(k1), g(k2, h(k3)))$ est linéarisé par `tmp1 = f(k1) ; tmp2 = h(k3) ; tmp3 = g(k2, tmp2) ; a = min(tmp1, tmp3) ;`. Il n'existe cependant pas de forme linéarisée unique.

Définition 7.2. *(Fonctions primitives et fonctions composées)* Nous distinguons deux types de fonctions de séquences de lexèmes :

1. les fonctions primitives constituées uniquement de lexèmes primitifs sur Σ^{p*} ;
2. les fonctions composées constituées uniquement de lexèmes d'appel sur Σ^{c*}

Pour la suite, nous considérons uniquement des fonctions primitives et composées. Une fonction mixte est donc préalablement transformée en fonctions composée par l'externalisation des séquences de lexèmes consécutifs pour créer de nouvelles fonctions primitives.

7.1.3 Aperçu de la méthode de factorisation

Afin d'obtenir le graphe d'appels factorisé d'un ensemble de projets, nous réalisons la factorisation des chaînes de lexèmes primitifs consécutifs des projets considérés. Cette factorisation est réalisée par la détermination de facteurs répétés maximaux sur les chaînes de lexèmes primitifs : un tel facteur est remplacé par un lexème d'appel dans sa chaîne originale, et, si nécessaire, une nouvelle fonction contenant ce facteur est externalisée. Par exemple, les chaînes de lexèmes primitifs $f_1 = abcdefgh$ et $f_2 = ijkabcdebcd$ disposent d'un facteur répété $abcde$ qui est externalisé dans une fonction $f_3 = abcde$. Chaînes de lexèmes primitifs et d'appels n'étant pas mélangées, nous obtenons $f_1 = f_3 f_1'$ et $f_2 = f_2' f_3 f_2''$ avec $f_1' = fgh, f_2' = ijk, f_3' = bcd$.

Il s'avère qu'une seule itération de factorisation est généralement insuffisante : les fonctions issues de facteurs communs et celles issues du découpage peuvent potentiellement accueillir elles-aussi, des facteurs répétés. Pour l'exemple traité, f_3 aurait pu être factorisé en $af_3'e$. Il est toutefois nécessaire de fixer un seuil minimal t à la longueur des facteurs répétés factorisés afin d'éviter la création de fonctions liées à des chaînes de lexèmes trivialement répétées en nombre dans le code (déclarations de variable, instruction fréquentes...). Lorsque l'itération courante de factorisation ne permet pas de trouver un facteur répété de longueur d'au-moins t, l'algorithme est arrêté.

Nous obtenons alors finalement un jeu de fonctions feuilles qui sont des chaînes de lexèmes primitifs ainsi que de fonctions internes composées de lexèmes d'appel. Nous en déduisons un graphe d'appels factorisé que nous utilisons afin de visualiser les zones de code similaires

entre projets ainsi que pour calculer une métrique de similarité entre fonctions du code. Cette métrique a pour vocation d'offrir une résistance acceptable aux opérations d'obfuscation par ajout ou suppression de code inutile ainsi que par la factorisation ou le développement de fonctions.

L'algorithme 4 résume les différentes étapes du processus de factorisation que nous allons maintenant examiner plus en détails.

Données : Graphes d'appel de fonctions de lexèmes $\{G_1, G_2, \cdots, G_k\}$ pour les projets $p_1, p_2, \cdots, p_k$
Résultat : graphe d'appels factorisé G_f et matrice de similarité entre fonctions internes M_I

début
  Pré-traitement des graphes d'appels : distinction entre fonctions de lexèmes primitifs (fonctions feuilles) et fonctions de lexèmes d'appel (fonctions internes) [7.1.2] ;
  $\iota \longleftarrow 0$ *(itération courante)* ;
  $L^\iota \longleftarrow$ fonctions feuilles avant la première itération, triées par longueur croissante ;
  tant que $i = 0 \vee L^\iota \neq L^{\iota-1}$ **faire**
    $\iota \longleftarrow \iota + 1$;
    $I^\iota \longleftarrow I^{\iota-1}$ *(initialisation des fonctions internes pour l'itération ι)* ;
    $L^\iota \longleftarrow L^{\iota-1}$ *(initialisation des fonctions feuilles pour l'itération ι)* ;
    pour $f_i \in L^{\iota-1}$ (pour chaque fonction feuille, de la plus courte à la plus longue) **faire**
      $\mathcal{C}(f_i) \longleftarrow$ 2-correspondances de facteurs répétés sur f_i et $L^{\iota-1}_{\leq i}$ [7.2] ;
      $\bar{\mathcal{C}}(f_i) \longleftarrow$ élimination des chevauchements de $\mathcal{C}(f_i)$ [7.2] ;
      si $\left|\bar{\mathcal{C}}(f_i)\right| > 0$ **alors**
        $\kappa \longleftarrow 1$;
        $f'_i = []$ *(nouvelle fonction interne f'_i)* ;
        pour $(u, v) \in \bar{\mathcal{C}}(f_i)$ *(avec $u = f_i[j|p]$), triés par j croissant* **faire**
          $L^\iota \longleftarrow L^\iota \cup \{\dot{u} = f_i[\kappa..j-1], u\}$;
          $f'_i = f'_i + [\dot{u}, u]$;
          $\kappa \longleftarrow j + p$;
        $L^\iota \longleftarrow L^\iota - \{f_i\}$ *(suppression de la fonction feuille)* ;
        $I^\iota \longleftarrow I^\iota \cup \{f'_i\}$ *(la fonction feuille f_i devient une fonction interne)*;
    $L^\iota \longleftarrow$ tri de L^ι par fonctions de longueur croissante ;
  $G_f \longleftarrow (I^\iota, L^\iota)$;
  $M_I \longleftarrow$ calcul de la matrice de similarité des fonctions internes (en utilisant C_L) [7.5] ;
  retourne (G_f, M_I) ;
fin

Algorithme 4 : Factorisation de fonctions de lexèmes

7.2 Recherche de facteurs répétés de t_i par des facteurs non-chevauchants de $T_{<i}$

Recherche de facteurs répétés pour chaque t_i

Lors de l'itération de factorisation ι, nous cherchons à factoriser les chaînes de lexèmes primitifs issues de l'itération $\iota-1$. À cet effet, nous trions les chaînes de l'itération précédente par ordre croissant de longueur et nous déterminons, pour chaque chaîne t_i, l'ensemble des facteurs répétés comportant au moins une occurrence sur t_i et sur $T_{\leq i}$, i.e. l'ensemble des chaînes d'indice inférieur ou égal à i dont toutes celles plus courtes que t_i. Cette problématique a été traitée en 6.5. Nous construisons ainsi à cet effet une table de suffixes des chaînes de T ainsi que des structures annexes aboutissant à l'obtention du graphe des farmax. Nous déterminons pour chaque chaîne t_i les farmax qui la composent et qui comportent une occurrence sur t_i et au moins une occurrence non-chevauchante sur $T_{\leq i}$: pour chaque facteur répété, nous préférons associer à l'occurrence de t_i une occurrence dont la couverture sur sa fonction d'origine est maximale. Ainsi par exemple pour les chaînes $T = \{t_1 = bc, t_2 = bca, t_3 = dbc\}$, nous préférons lorsque nous traitons t_3 relever la correspondance $(t_3[2..3], t_1)$ plutôt que $(t_3[2..3], t_2[1..2])$ qui nécessite l'externalisation inutile du facteur bc au sein d'une nouvelle fonction redondante avec t_1 (lors de l'itération suivante, cette nouvelle fonction sera identifiée avec t_1). On liera ainsi une occurrence de farmax sur t_i à l'occurrence de position minimale (appartenant ainsi à la plus petite fonction).

Nous obtenons finalement un jeu de couples de correspondances $\mathcal{F}(t_i) = \{(u_1, v_1), (u_2, v_2), \cdots, (u_n, v_n)\}$ avec $u_1, u_2, \cdots, u_n \in \text{fact}(t_i)$ et $v_1, v_2, \cdots, v_n \in \text{fact}(T_{\leq i})$.

Élimination des chevauchements

Présentation Nous pouvons noter qu'il est possible que les farmax (par définition non-extensibles) relevés sur t_i puissent se chevaucher. Ceci est problématique dans la mesure où une seule factorisation est attendue pour t_i. Plusieurs méthodes d'élimination de chevauchements peuvent être employées afin d'optimiser certains critères concernant les facteurs non-chevauchants finaux obtenus. La problématique de l'élimination de correspondances chevauchantes est également rencontrée lors de la recherche de facteurs communs par la méthode de tuilage glouton (cf section 8.4). Nous optons ici pour une réponse via une approche similaire : nous privilégions les plus longs facteurs. Si la longueur minimale des facteurs relevée est fixée à $t > 1$, il est possible que la couverture de la chaîne t_i par les facteurs non-chevauchants sélectionnés ne soit pas optimale.

Principe Nous rappelons que la méthode de sélection de correspondances non-chevauchantes avec priorité aux facteurs les plus longs est implantée par l'usage d'une file de priorité contenant initialement l'ensemble des correspondances initiales. À chaque itération, la correspondance la plus longue (u_j, v_j) est sélectionnée : si l'occurrence de sa composante sur t_i, u_j, intersecte une correspondance de la file, cette correspondance est supprimée de la file pour y insérer ensuite la zone non-chevauchante de cette correspondance. Lorsque la file est vide ou ne contient que des correspondances liées à des facteurs de longueur strictement inférieure à t, la sélection est terminée. Nous notons que toutefois ces correspondances peuvent se chevaucher quant à leur

composante sur $T_{\leq i}$. Il est du ressort des itérations suivantes de factoriser ces chevauchements si ceux-ci sont de taille supérieure au seuil de factorisation.

Exemple Considérons la chaîne $t_i = acdeefgh$ avec, parmi les chaînes d'indice inférieur à i, $T_{<i} = \{\cdots, t_\alpha = acd, \cdots, t_\beta = cde, \cdots, t_\gamma = cdee, \cdots, t_\delta = efgh\}$ avec le relevé de correspondances suivantes : $\mathcal{F}(t_i) = (acd : (t_i[1..3], t_\alpha), cdee : (t_i[2..5], t_\gamma), efgh : (t_i[5..8], t_\delta))$ avec un seuil de report de $t = 2$ lexèmes au minimum. Le facteur $efgh$ le plus long est sélectionné, tronquant le lexème de droite de *cdee* qui devient *cde* : *cde* reste lié à $t_\gamma[1..3]$ qui n'est pas l'occurrence de position minimale du farmax *cde* (t_β). La file contient les correspondances liées au facteur *cde* et *acd* : à égalité de longueur, on sélectionne arbitrairement *cde* et *acd* est tronqué en *a*. Finalement la file ne contient plus qu'une correspondance liée au facteur *a* : celle-ci n'est pas sélectionnée car $t = 2$. t_i est donc ainsi décomposé : $t_i = a(cde)(efgh)$ avec *cde* externalisé en nouvelle fonction et $efgh$ remplacé par un lexème d'appel vers t_δ. On note que pour $t = 2$, cette décomposition n'est pas optimale : la décomposition $t_i = (ac)(de)(efgh)$ en revanche l'est.

Complexité Nous notons préalablement à l'évaluation de la méthode de résolution de recouvrement que chaque chaîne de t_i de T ne peut faire l'objet que, d'au plus, $|t_i| - t + 1$ correspondances sur $T_{<i}$. En effet, la non-extensibilité des facteurs répétés a pour conséquence qu'il existe au plus un facteur répété de longueur supérieure ou égale à t commençant à une position j de t_i avec $1 \leq j \leq |t_i| - t + 1$. Il s'agit donc de déterminer au plus k facteurs non-chevauchants à partir de k facteurs chevauchants, avec $k \leq |t_i| - t + 1$. Lorsque le plus grand facteur $\alpha = t_i[j..j + p - 1]$ est extrait de la file à chaque itération, celui-ci peut potentiellement intersecter p facteurs débutant par un lexème initial dans $t_i[j..j + p - 1]$. Quant aux facteurs intersectant α débutant sur $t_i[1..j - 1]$, ils sont au plus au nombre de $p - 1$. Ainsi globalement au plus $2p - 1$ facteurs intersectent α. La détermination des facteurs restant dans la file intersectant α peut être réalisé naïvement par test exhaustif de ceux-ci. Il est plus avantageux d'employer une structure d'arbre de segments représentant les intervalles d'indices occupés sur t_i par chacun des facteurs. Un tel arbre est généralement implanté par une structure d'arbre binaire avec indexation de l'indice de départ de chaque intervalle. La détermination des facteurs chevauchant α, leur suppression et la remise en file des reliquats de facteurs est ainsi réalisable en temps $O(p \log |t_i|)$. Finalement comme la somme des longueurs des facteurs sélectionnés est au plus de $|t_i|$ lexèmes, la complexité temporelle s'élève à $O(|t_i| \log |t_i|)$.

Utilisation d'une fonction de volume

Dans une optique simplificatrice, nous avons présenté la recherche de facteurs répétés et la résolution des chevauchements en prenant en considération la longueur des facteurs. Il est possible d'y substituer une fonction de volume sur les chaînes de lexèmes pondérant une chaîne selon l'importance du code sous-jacent qu'elle représente. Une telle fonction $\mathcal{V}$ doit être croissante, i.e. si u est un facteur propre de v, $\mathcal{V}(u) < \mathcal{V}(v)$.

Il est envisageable d'utiliser un vecteur de volumes unitaires pour chaque lexème et de définir une fonction de volume sur une chaîne comme la somme des volumes unitaires de chaque occurrence de lexème. Une progression hyperlinéaire de la fonction peut aussi être

```
1  void estPremier(int n, int * primalite) {
     int k; *primalite = -1 /* Primalité inconnue */
3    if (n <= 2) *primalite = 0;
     for (k=2; *primalite < 0 && k <= sqrt(n); k++) if (n % k == 0) *primalite = 0;
     if (*primalite < 0) *primalite = 1;
   }

8  /* Avant développement */
   int * t;
   ...
   /* Le z-ième prédecesseur d'une puissance de 2 est-il Mersennien ? */
   /* L'opérateur ^ désigne la fonction puissance */
13 estPremier(2^z-1, &(t+a));
   ...

   /* Après développement */
   ...
18 int p; *(&(t+a)) = -1;
   if (2^z-1 <= 2) *(&(t+a)) = 0;
   for (p=2; *(&(t+a)) < 0 && p <= sqrt(2^z-1); k++) if (2^z-1 % p == 0) *(&(t+a)) = 0;
   if (*(&(t+a)) < 0) *(&(t+a)) = 1;
   ...
```

FIG. 7.1 – Micro-redondances d'expressions après développement de fonction

utilisée par une fonction corrigée $\mathcal{V}' : x \longrightarrow \mathcal{V}(x)^{1+\epsilon}$ privilégiant une chaîne longue sur deux chaînes courtes de longueur additionnée équivalente.

7.3 Auto-factorisation de t_i

Certaines fonctions initiales du code source sont susceptibles de contenir des zones répétées en leur sein. Ceci peut être lié à des pratiques abusives de copier-coller de la part du programmeur ou bien à des opérations d'obfuscation consistant à insérer du code inutile en plusieurs exemplaires ou à dupliquer du code préexistant en déroulant, par exemple, des boucles. Rechercher des facteurs répétés sur une fonction représentée par une chaîne de lexèmes peut donc être intéressant : ceci est réalisé par la recherche de décomposition de la fonction t_i sur $T_{\leq i}$. Nous pouvons obtenir alors des 2-correspondances non-chevauchantes sur t_i de volume supérieur au seuil de report t. Des fonctions peuvent également comporter de micro-redondances inférieures au seuil t qui n'ont cependant pas de sens rapportées aux fonctions extérieures. C'est le cas par exemple de paramètres de fonctions qui auraient été remplacés par des expressions correspondant aux arguments d'appel lors d'une opération de développement. Ces expressions répétées pourraient être détectées et remplacées par un lexème primitif abstrait d'identificateur. Un processus d'auto-factorisation interne à la fonction peut être mené en calculant le graphe des farmax propre à la fonction pour sélectionner des groupes de micro-clones intéressants. Un exemple est présenté en figure 7.1 où une fonction vérifiant la primalité d'un entier est développée.

7.4 Factorisation des facteurs approchés

La méthode précédemment décrite permet de localiser des facteurs répétés entre fonctions lexemisées du code ainsi que leurs imbrications. Elle se révèle limitée lorsque des opérations

d'édition locales sont réalisées car n'opérant que sur des facteurs de lexèmes exacts. À l'issue de la dernière itération de factorisation, nous pouvons nous interroger sur l'existence, au sein des fonctions feuilles du graphe d'appels obtenu, de facteurs approchés partagés. Ces facteurs partagés comportant des fossés de non-correspondance ne peuvent être localisés par indexation de suffixes : nous utilisons à cet effet un algorithme d'alignement local avec coupure des trop longs fossés tel que décrit en 5.4.2. Afin de déterminer les facteurs approchés entre chaque fonction feuille f_i et les autres fonctions feuilles, nous devons réaliser un processus d'alignement local sur chacune des paires de fonctions feuilles. Ce procédé est de complexité temporelle $\Theta(\sum_{l_i \in L} |l_i|^2)$ pour l'ensemble des fonctions feuilles F. Cette complexité est plus avantageuse que celle nécessaire à l'auto-alignement local de la concaténation des projets lexemisés car nous comparons deux à deux des fonctions feuilles de taille limitée, chacune pouvant représenter, lorsqu'elle n'est pas intersticielle, une portion de code qui ne sera pas comparée plusieurs fois. Il demeure cependant que certaines fonctions feuilles, ne présentant pas de similarité manifeste, pourraient faire l'économie d'un traitement par alignement : nous introduisons à cet effet une étape de filtrage présélectionnant les fonctions feuilles à comparer.

Présélection La présélection des fonctions feuilles à comparer est réalisée en calculant une matrice creuse employant une métrique de similarité dégradée mais simple à calculer. On peut à cet effet utiliser la métrique présentée ultérieurement en 12.2.2 utilisant une fenêtre glissante sur la table de suffixes des fonctions feuilles à comparer. Elle repose sur l'idée intuitive que des fonctions feuilles possédant des longs et/ou nombreux facteurs communs voient leur occurrences rapprochées dans la table de suffixes. Pour des fonctions feuilles de longueur cumulée N, nous pouvons limiter la taille de la matrice creuse de similarité à kN valeurs non-nulles, k étant un paramètre arbitrairement choisi. Ainsi seuls les couples de fonctions feuilles considérées comme les plus similaires par cette heuristique feront l'objet d'un alignement local.

Alignement et factorisation Le processus de factorisation est similaire à celui employé pour rechercher des facteurs exacts. Pour chaque fonction feuille l_i, nous déterminons les fonctions $\mathcal{L}$ de $L_{\leq i}$ de longueur inférieure ou égale présentant une similarité pré-sélectionnante selon l'heuristique de filtrage employée. Nous recherchons les correspondances approchées par alignement local pour chaque couple $(l_i, \mathcal{L}_j)_{\mathcal{L}_j \in \mathcal{L}}$ avec une certaine tolérance pour les fossés. Pour l'ensemble des correspondances approchées trouvées, nous éliminons les chevauchements selon la méthode précédemment décrite en 7.2 afin de trouver une factorisation de l_i favorisant les plus grands facteurs approchés, tout en fixant un seuil minimal au volume des correspondances. l_i est réécrite en fonction interne par des lexèmes d'appels vers les occurrences correspondantes avec externalisation si nécessaire de nouvelles fonctions. On note que pour une correspondance approchée $(u \in l_i, v \in L \leq i)$, en supposant que v n'est pas une fonction entière, une fonction feuille contenant uniquement u est créée par externalisation : elle représente le facteur identifié sur l_i. Lors de la prochaine itération, cette fonction sera mise en correspondance avec v : il est néanmoins nécessaire de conserver une trace au sein de cette fonction des deux facteurs différents u et v. Lors de l'étape de filtrage des itérations suivantes, un des facteurs contenus par la fonction externalisée pourra être utilisée comme son représentant à titre d'approximation.

7.5 Similarité des fonctions internes

7.5.1 Problématiques et similarité proposée

Les fonctions internes du graphe d'appels factorisé sont représentées par des chaînes de lexèmes d'appel. Deux types de fonctions internes peuvent être distingués : les fonctions internes originelles issues de fonctions existant dans le code source ainsi que les fonctions internes synthétiques créées par le processus de factorisation. Nous nous intéressons tout spécialement au calcul d'une métrique de similarité entre paires de fonctions internes originelles, qui peuvent être regroupées par unité structurelle d'appartenance (projet, paquetage...).

Dans la mesure où les chaînes de lexèmes factorisées depuis les fonctions originelles peuvent être potentiellement transposées, l'usage d'une métrique ensembliste fait sens. Il s'agit de définir sur les chaînes de lexèmes primitifs une fonction de volume $\mathcal{V}$ pour quantifier leur importance. La similarité entre deux fonctions internes f et g est alors quantifiée par les volumes additionnés de toutes les fonctions feuilles de lexèmes primitifs (ou leur groupe d'appartenance) atteignables depuis f et g.

Cette métrique quantifie un volume de code partagé entre f et g sans considération d'ordre. Si la vulnérabilité aux opérations de transposition est réduite, des couples de fonctions faussement similaires sont envisageables si le seuil t de longueur minimale de chaîne factorisée est particulièrement bas. D'autre part un obfuscateur pourrait factoriser à outrance du code source copié afin que le graphe d'appels du projet soit déjà fortement factorisé avec des chaînes de lexèmes primitifs de longueur faible inférieure à t. Dans ce cas aucune factorisation mutuelle avec un autre projet n'est possible, sauf à abaisser t et à augmenter le taux de paires de fonctions faussement similaires. Les conséquences négatives de ce type d'attaque obfuscatoire peuvent être limitées par un pré-traitement du code source consistant à y développer les appels vers des fonctions de faible taille.

7.5.2 Calcul de la similarité

Le calcul de la similarité entre deux fonctions f et g interne du graphe d'appels factorisé nécessite la détermination de l'intersection des ensembles de fonctions feuilles transitivement atteignables depuis f ($\mathcal{R}^{+L}(f)$) et depuis g ($\mathcal{R}^{+L}(g)$). Pour chaque fonction f, $\mathcal{R}^{+L}(f)$ est calculable par le parcours des arêtes du graphe d'appels. Comme noté en 3.4.4, il est possible de déduire du graphe d'appels un graphe acyclique (DAG) où les composantes fortement connexes du graphe d'appels sont remplacées par un unique sommet représentatif. Ces composantes fortement connexes représentent des fonctions mutuellement récursives : l'ensemble des fonctions atteintes par des fonctions de la même composante connexe sont les mêmes et donc *a fortiori* l'ensemble des fonctions feuilles atteintes. Nous déterminons ainsi pour f et g $\mathcal{R}^{+L}(f \cap g) = \mathcal{R}^{+L}(f) \cap \mathcal{R}^{+L}(g)$ en utilisant les fonctions feuilles atteintes depuis les composantes fortement connexes du DAG d'appels dérivé. Pour chaque paire, cette opération est réalisée en temps $O(n)$ pour n composantes fortement connexes de fonctions. Il en résulte une complexité temporelle en $O(n^3)$ pour le calcul du volume d'intersection des fonctions feuilles atteignables depuis toutes les paires de sommets du DAG d'appels.

Nous remarquons que l'ensemble des fonctions feuilles atteintes et partagées par deux fonctions internes ne peut pas inclure des fonctions feuilles interstitielles. De l'ensemble des feuilles partagées est dérivée une métrique s'appuyant sur la somme des volumes de ces fonctions. Cette valeur est à mettre en parallèle pour normalisation soit avec le volume global des fonctions feuilles potentiellement factorisables[1] et atteignables depuis la fonction f ou g, soit le volume global de l'union de ces deux ensembles. Nous examinons en 12.3.2 différents types de normalisation ainsi employables.

7.6 Exploitation du graphe d'appels factorisé

Si le graphe d'appels factorisé peut être utilisé sous sa forme brute pour le calcul des métriques de similarité entre paires de fonctions, certaines opérations de filtrage permettent une manipulation plus aisée dans certaines situations. Nous examinons ici le filtrage du graphe d'appels par la considération des multiplicités des fonctions feuilles ainsi que par la sélection des fonctions originelles racines.

7.6.1 Filtrage sur la multiplicité des fonctions feuilles

Dans certains cas, un jeu de projets analysé peut comporter des clones de code présents dans la majorité de ceux-ci. Cela survient notamment pour des projets d'étudiants de même finalité. Ces clones sont généralement accidentels et révèlent l'emploi d'un code assez standardisé pour traiter un problème similaire. On pourra par exemple s'attendre à ce qu'un algorithme de tri simple soit implanté de manière uniforme. D'autre part, il peut s'agir de code pré-fourni par l'ordonnateur du projet. Il est préférable ainsi d'ignorer ce type de clone pour le calcul de similarité.

Nous pouvons gérer ce type de situations en choisissant d'ignorer les fonctions feuilles du graphe d'appels factorisé atteintes par plus de γ projets. Nous définissons pour chaque fonction du graphe une mesure de multiplicité spécifiant le nombre de projets l'atteignant par appels transitifs. Celle-ci est calculée en associant à chaque fonction un vecteur booléen d'appartenance aux projets. Les fonctions originelles (racines) appartiennent à leur projet source, tandis que récursivement une fonction interne appartient à l'union des projets d'appartenance des fonctions qui l'appellent directement. Les fonctions feuilles de multiplicité d'au moins γ sont alors ignorées.

Plutôt que d'ignorer complètement les fonctions feuilles de forte multiplicité, nous pouvons leur assigner une valeur déduite d'une métrique $\mathcal{M}$ dérivée de la multiplicité. Cette métrique sera utilisée pour pondérer le volume des fonctions feuilles pour le calcul de la similarité. Celle-ci peut être une fonction décroissante de la multiplicité tendant vers 0 pour de fortes valeur de multiplicité, telle que la fonction affine $\mathcal{M} : m \longrightarrow \frac{\max(0,\gamma-m)}{\gamma-2}$ de valeur 1 pour la multiplicité la plus faible 2 et atteignant 0 pour une multiplicité de γ.

[1] Une fonction feuille est potentiellement factorisable ssi son volume est supérieur ou égal au seuil de factorisation t employé.

7.6.2 Filtrage des fonctions originelles racines

Lorsque les duplications sont peu nombreuses, la taille du graphe d'appels factorisé ne diffère pas sensiblement de l'union des graphes d'appels initiaux pour le jeu de projets considéré. En revanche pour de nombreuses duplications, le graphe factorisé peut être difficile à examiner dans sa globalité par un humain. Il peut être utile alors d'examiner seulement un extrait de celui-ci. On pourra ainsi n'examiner que le graphe d'un sous-ensemble de projets en ne sélectionnant que leur fonctions originelles comme racines et en ne conservant que les fonctions synthétiques et feuilles accessibles depuis celles-ci. Nous pouvons également ne conserver que les fonctions de multiplicité minimale de 2. Les fonctions de multiplicité unitaire sont soit des fonctions interstitielles reliquats du découpage des chaînes lors de la factorisation, soit des fonctions partagées uniquement au sein d'une unité structurelle : elles peuvent être écartées par un examinateur humain.

7.6.3 Retour au code source à partir des fonctions du graphe d'appels factorisé

Il est essentiel de pouvoir, à partir d'une fonction synthétique interne du graphe, obtenir la zone de code source s'y rapportant lors de l'examen de similarités. À cet effet nous maintenons pour chaque occurrence de lexème présente dans les fonctions initiales un lien spécifiant sa position dans le code source. Si l'on se limite à un granularité de retour à la ligne de code correspondant au lexème (et non aux caractères), nous utilisons une structure d'arbre de segments afin de spécifier, pour chacune des chaînes, l'intervalle des lexèmes appartenant à une même ligne associé à l'identificateur de la ligne et de l'unité de compilation. Lorsqu'une chaîne de lexèmes est décomposée en plusieurs sous-chaînes lors du processus de factorisation, chacune de ces sous-chaînes reçoit une copie de l'extrait de l'arbre de retour au source la concernant.

7.7 Analyse d'un exemple : copie et obfuscation d'une fonction de tri

Nous présentons ici en figure 7.2 un cas d'obfuscation simple sur une fonction de tri récursive de complexité quadratique. La fonction originale *subsort* utilise deux fonctions annexes : *min_index* afin de déterminer l'élément minimal sur la fin du tableau et *exchange* pour réaliser l'échange de deux éléments. L'opération d'obfuscation consiste à développer les appels à *min_index* et *exchange*. Nous y ajoutons d'autre part une boucle inutile. Pour la factorisation, les déclarations sont ignorées et les identificateurs sont abstraits. La fonction de volume utilisée est basée sur la longueur en lexèmes de la chaîne. Nous imposons un seuil de $t = 10$ lexèmes pour la factorisation.

Pré-traitement Une analyse syntaxique légère permet de mettre en évidence les fonctions, lexèmes primitifs et lexèmes d'appels correspondant aux appels de fonctions. Nous obtenons les fonctions (@ préfixe les lexèmes d'appel) de la figure 7.3.

Première itération Lors de la première itération, nous recherchons des facteurs répétés sur le jeu de fonctions de lexèmes primitifs $\{j, i, n, g, f, m\}$ triées par longueur croissante. Si

Fonction de tri originale

```
1  int find_min(int[] tab, int from)
   {
3   int min_index = −1; int min_value = −1;
    for (int i=from; i < tab.length(); i++)
     if (min_index < 0 || tab[i] < min_value)
     {
      min_index = i;
8     min_value = tab[i];
     }
    return min_index;
   }

13 void exchange(int[] tab, int i, int j)
   {
    int tmp;
    tmp = tab[i]; tab[i] = tab[j]; tab[j] = tmp;
   }
18
   void sort(int[] tab, int start)
   {
    if (start <= tab.length − 1)
    {
23   int i = find_min(tab, start);
     if (i != start) exchange(tab, start, i);
     sort(tab, start+1);
    }
   }
28
   void sort(int[] tab)
   {
    sort(tab, 0);
   }
```

Fonction de tri obfusquée

```
1  void sort2(int[] tab, int start)
   {
3   if (start <= tab.length − 1)
    {
     int min_index = −1; int min_value = −1;
     for (int i=from−0; i < tab.length(); i++)
      if (min_index <= −1 || tab[i] < min_value)
8     {
       min_index = i∗1;
       min_value = tab[i];
      }
     if (min_index != start)
13   {
      tmp = tab[start]; tab[start] = tab[min_index];
            tab[min_index] = tmp;
     }
     sort2(tab, start+1);
    }
18  for (int i=0; i < start; i++)
     System.err.println("Code_inutile...");
   }

   void sort2(int[] tab)
23 {
    sort2(tab, 0);
   }
```

FIG. 7.2 – Fonction de tri originale et version obfusquée par développement

Fonction originelle	Fonction créée	Chaîne de lexèmes	Volume
find_min	f	FOR (... RETURN ID ;	43
exchange	g	ID = ID[ID] ... ID[I] = ID ;	24
sort(int[], int)	h	@i @find_min @j @exchange @sort(int[], int)	
sort(int[], int)	i	if (...)	10
sort(int[], int)	j	if (ID != ID)	6
sort(int[])	k	@sort(int[], int)	
sort2(int[], int)	l	@m @sort2(int[], int) @n	
sort2(int[], int)	m	if (ID ≤ ID.length - CONST) ... ID[ID] = ID) }	84
sort2(int[], int)	n	for (int ID ... CONST) ;	23
sort2(int[])	o	@sort2(int[], int)	

FIG. 7.3 – Fonctions de lexèmes issues du code source des fonctions de tri

le seuil de similarité était inférieur ou égal à 7, nous pourrions trouver par exemple sur la fonction f des facteurs répétés partagés avec la fonction g telle que l'instruction d'affectation `ID = ID[ID]`. Avec le seuil de similarité fixé à 10, seuls des facteurs répétés partagés avec m sont reportés : il s'agit du début de la fonction g d'échange externalisé en g' pour m. Quant à la fonction f, elle n'est pas identifiée avec une portion de m car des petites opérations d'édition

par réécriture d'expressions ont été introduites avec une fréquence supérieure à 1/10 lexèmes : par découpage la fonction m' issue de m est créée (volume de 49). À l'issue de la première itération, la fonction m est substituée par la chaîne de lexèmes d'appel $m_1 = @i@m'@g'$ (cf figure 7.4). L'ensemble des chaînes de lexèmes primitifs est $L_1 = \{j, i, n, g, g', f, m'\}$.

m	if (ID $\leq$ ID.length - CONST)	{ ID = CONST ... IF (ID != ID) {	ID = ID[ID] ... ID[ID] = ID }
m_1	$@i$	$@m'$	$@g'$

FIG. 7.4 – Réécriture de la fonction feuille m en fonction interne m' avec externalisation de fonctions identifiées

Deuxième itération La seconde itération permet de constater l'égalité entre g et g' : g' est donc remplacé par un unique lexème d'appel à g. De façon générale lorsque qu'une fonction entière ou partielle $\rho[\alpha..\beta]$ a été trouvé dans une fonction ω et externalisée en ω' au cours de l'itération i, l'itération $i+1$ permet l'identification de ω' dans $\rho[\alpha..\beta]$ et le remplacement par un lexème d'appel. Finalement, nous obtenons le graphe d'appels suivant :

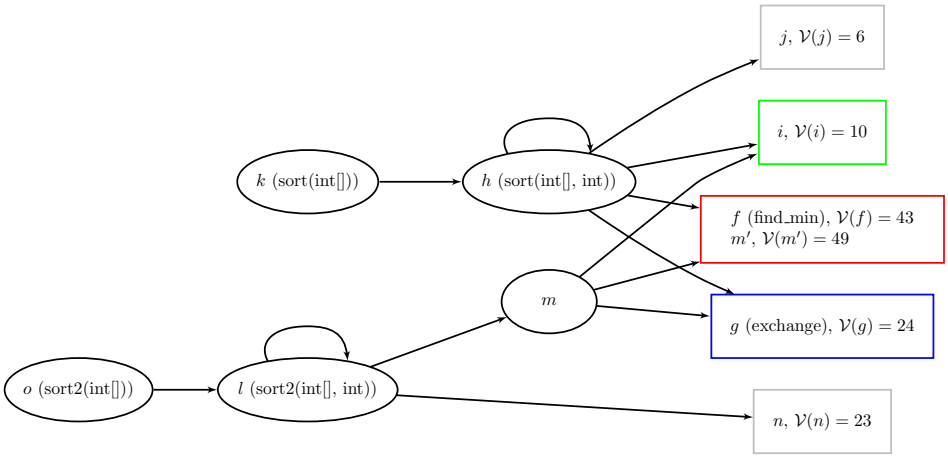


FIG. 7.5 – Graphe d'appel des fonctions de tri après factorisation

Similarité des fonctions feuilles Nous pouvons alors calculer la similarité des paires de fonctions feuilles. Sans détailler le processus, il apparaît que f et m', sans disposer de facteurs répétés d'au moins 10 lexèmes qui aurait permis une factorisation plus approfondie, possèdent une similarité élevée par l'usage d'une métrique de proximité sur fenêtre. Ces fonctions feuilles peuvent ainsi être factorisées en une fonction par alignement local.

Similarité des fonctions originelles L'obtention du DAG d'appels à partir du graphe d'appels factorisé permet de confondre les fonctions k, h ainsi que l, o qui forment des composantes connexes. En employant le volume de l'intersection des fonctions feuilles atteignables depuis chacune des paires des fonctions, nous obtenons la matrice suivante portant sur les fonctions originelles :

	f (find_min) $\mathcal{V} = 43$	g (exchange) $\mathcal{V} = 24$	h, k (sort) $\mathcal{V} = 86$
g (exchange) $\mathcal{V} = 24$	0		
h, k (sort) $\mathcal{V} = 86$	46	24	
l, o (sort2) $\mathcal{V} = 103$	46	24	80

FIG. 7.6 – Matrice des volumes partagés par les fonctions de tri

Ces volumes partagés peuvent ensuite être mis en rapport avec le volume des fonctions feuilles atteignables depuis l'une des fonctions comparées ou bien par rapport au volume de l'union des fonctions feuilles atteignables depuis les deux fonctions originelles comparées. Ces méthodes de normalisation seront décrites plus en détails ultérieurement en 12.3.2.

7.8 Étude expérimentale de projets d'étudiants

Nous souhaitons étudier les similarités d'un jeu de projets d'étudiants en factorisant leurs graphes d'appels. Nous nous intéressons à des projets en langage C dont la résolution de liens d'appels de fonctions peut être réalisée sans ambiguïté (en faisant abstraction de l'usage de pointeurs de fonction). La série de mini-projets étudiés a été réalisée au cours de travaux-dirigés par différents groupes d'étudiants sur quatre années, certains énoncés de projet ayant été traités par plusieurs promotions. Nous cherchons à quantifier les plagiats réalisés entre étudiants d'une même promotion sur un énoncé donné de projets, au sein d'une même promotion ainsi qu'entre différentes promotions. Nous nous intéressons également au phénomène d'auto-plagiat où un étudiant réutilise son propre code d'un projet précédent : cette approche est encouragée dans le cadre d'un développement modulaire et générique.

Projet	Composantes	Nombre de projets			
		2007	2008	2009	2010
Jeu de serpent	File	17	15	16	15
Calculatrice RPN	Pile	14	15	13	14
Compresseur *Move To Front*	Liste chaînée, bibliothèque de codage d'entier	11	12	11	15
Arbre d'expression	Pile	10			
Autres projets (Huffman, réussites, calculatrice modulaire)	Pile, liste...	3	10		6

FIG. 7.7 – Jeu de projets d'étudiants étudié

7.8.1 Étude des feuilles et calcul de la similarité des projets

Les projets étudiés ainsi que l'effectif des promotions l'ayant traité sont indiqués en figure 7.7. L'ensemble des projets rendus sont lexemisés afin de générer pour chacun un graphe de séquences d'appel ; ces graphes étant factorisés ensuite par la recherche de facteurs de lexèmes exactement égaux (moyennant une abstration des identificateurs et types) selon la méthode décrite précédemment au cours de ce chapitre. Le seuil de report de correspondances est fixé à $t = 10$ lexèmes. Nous recherchons pour chaque paire de projets le volume additionné des l'intersection des différentes fonctions feuilles que leurs fonctions initiales atteignent. Sur

les 199 projets initiaux et 19 701 paires correspondantes, 19 181 partagent au moins une fonction feuille de volume d'au moins 10 lexèmes.

Il est probable que certaines fonctions feuilles sont issues de factorisation de code courant. Du tel code peut être lié à des formes idiomatiques de séquences de lexèmes ou alors à l'inclusion dans le projet rendu de bibliothèques fournies en énoncé (e.g. une bibliothèque graphique pour le jeu de serpent). À cet effet, nous introduisons une métrique de multiplicité sur les fonctions. Une fonction possédant pour fonctions racines dans le graphe d'appel factorisé des fonctions extraites de m projets distincts est de multiplicité m. Le graphe factorisé comporte 34 498 fonctions feuilles dont 2 391 issues de correspondances relevées (feuilles non-intersticielles). Nous classons ces dernières selon leur multiplicité (figure 7.8) :

Multiplicité	1	2	3..4	5..9	10..19	20..49	50..99	100..150
Nombre de feuilles	559	686	493	389	152	85	24	3

FIG. 7.8 – Répartition des fonctions feuilles par multiplicité

On choisit par la suite d'exclure les feuilles de multiplicité d'au moins M projets. Ce seuil ne doit pas être inférieur au plus grand groupe de projets similaires sans toutefois être trop élevé en permettant la prise en compte de similarité commune. Nous choisissons $M = 50$ projets. Ce choix élimine environ 20 % de paires de projets ne possédant en commun que des feuilles de multiplicité supérieure ou égale à 50.

Les paires de projets de similarité supérieure à $\frac{7}{10}$ (similarité $\mathrm{sim}_{\min}$ normalisée par rapport au volume des feuilles du plus petit projet) sont examinées manuellement. On en dénombre 19 dont 9 présentant une très forte similarité (supérieure à $\frac{9}{10}$). Trois types de paires de projet similaires se distinguent :

1. Les paires concernant un même projet copié au sein d'une même promotion (12 paires). Sans surprise, ces paires fortement similaires de projet plagié ont déjà pu être identifiées par le correcteur sans faire appel à un outil.
2. Les paires de projet copié entre différentes promotions (4 paires). Ces paires sont indecelables par un correcteur humain qui ne peut avoir une mémoire globale de projets rendus sur différentes années[2]
3. Les paires de projet auto-plagiées où certains modules communs sont réutilisés (3 paires).

Les frontières entre ces familles sont cependant floues : un projet rendu peut en effet emprunter du code de plusieurs projets rendus, inter- ou intra-promotion, eux-mêmes ayant fait l'objet d'auto-plagiat et ainsi induire par transitivité de nouvelles relations de similarité indirecte. On notera que la similarité étant considérée symétrique, nous ne pouvons déterminer automatiquement l'auteur originel d'un morceau de code[3].

[2]Un seul cas a pu être observé manuellement : l'étudiant avait rendu par mégarde la copie conforme d'un projet d'un étudiant d'une année antérieure disponible publiquement sur le web au lieu de son propre projet. Cette copie comportait toujours le nom de l'auteur original dans sa documentation. Ce même projet a également été repris par un autre étudiant mais en y introduisant des modifications.

[3]Cela peut justifier l'organisation d'une soutenance orale de projet.

Types de paires	Normalisation	Effectif total	$]\frac{8}{10}..1]$	$]\frac{5}{10}..\frac{8}{10}]$	$]\frac{2}{10}..\frac{5}{10}]$	$]\frac{1}{10}..\frac{2}{10}]$	$]0..\frac{1}{10}]$
Jeu de serpent	$\text{sim}_{\min}$	2 016	2	11	53	379	1 062
Calculatrice	$\text{sim}_{\min}$	1 540	4	2	27	150	511
Calculatrice 2007 Arbre d'expression 2007	$\text{sim}_{\min}$	238	1	4	7	13	55
Jeu de serpent Compresseur	$\text{sim}_{\max}$	3 087	0	0	0	1	1 301
Tous projets Projet externe	$\text{sim}_{\max}$	198	0	0	0	0	62

FIG. 7.9 – Étude des valeurs de similarité entre paires de projets sur différents sous-ensembles de paires

Utiliser différentes méthodes de normalisation (telles que discutées en 12.3.2) permet un meilleur aperçu des formes d'emprunt de code. Si la normalisation par rapport au plus petit projet ($\text{sim}_{\min}$) est une métrique intéressante pour quantifier le plagiat, les volumes respectifs des feuilles atteintes par chaque projet comparé ($\mathcal{V}(p_1)$ et $\mathcal{V}(p_2)$) permet d'affiner le scénario de copie. Si $\mathcal{V}(p_1) << \mathcal{V}(p_2)$, $\text{sim}_{\min} >> \text{sim}_{\max}$. En supposant $\text{sim}_{\min}$ élevé, nous pouvons envisager l'hypothèse, comme exprimé au chapitre 4 consacré à l'obfuscation, que le plagiaire ait copié le projet original en y ajoutant du code inutile. En pratique le code rajouté n'est jamais trivialement inutile : dans le cas contraire, son caractère serait rapidement mis en évidence par le correcteur. Plusieurs scénarios sont envisageables : soit l'ajout de fonctionnalités supplémentaires originales à partir d'une base plagiée, soit au contraire la copie uniquement des fonctionnalités essentielles abandonnant ainsi l'usage de code présent dans le projet original.

Nous étudions plus finement les paires similaires en considérant quelques sous-ensembles. Les premiers concernent des paires concernant un même projet, les seconds des projets différents comportant plus ou moins de modules pouvant faire l'objet d'un partage par auto-plagiat. Nous ignorons les feuilles de multiplicité strictement supérieure à 20. Le résultat est présenté en figure 7.9.

Les paires de forte similarité ont déjà été examinées précedemment : il est intéressant de considérer les paires de similarité moyenne ($\text{sim}_{\min} \in [\frac{2}{10}..\frac{8}{10}]$). Elles sont proportionnellement plus nombreuses pour le jeu de serpent que pour la calculatrice. Une analyse humaine confirme les emprunts de code entre les différents projets ; plus ou moins importants. Dans quelques cas, on pourrait subjectivement considérer la valeur de similarité sous-évaluée liée à des opérations répétées de réécriture d'expression (telles que `var++` réécrit en `var = var + 1` ou `tab[i]` en `*(tab+i)`...). La lexémisation d'un arbre de syntaxe abstrayant les petites expressions auraient probablement amélioré le rappel. Un exemple de graphe d'appels entre deux rendus de projet calculatrice en promotion 2007 ($\text{sim}_{\min} \sim 0,69, \text{sim}_{\max} \sim 0,59$) est présenté en figure 7.10.

En ce qui concerne les rendus de projet du jeu de serpent, nous exposons en figure 7.11 une carte de similarité (représentation graphique présentée ultérieurement en sous-section 14.2.2) permettant d'avoir un meilleur aperçu global des relations de similarité entre rendus : chaque rectangle modélise une paire de projets rendus, les côtés étant de longueur proportionnelle aux volumes globaux des projets et leur luminosité une fonction affine de leur similarité (en utilisant une normalisation par rapport au volume de l'union des feuilles). Étant donnée la méthode de normalisation, cette représentation est symétrique. Les projets y étant ordonnés

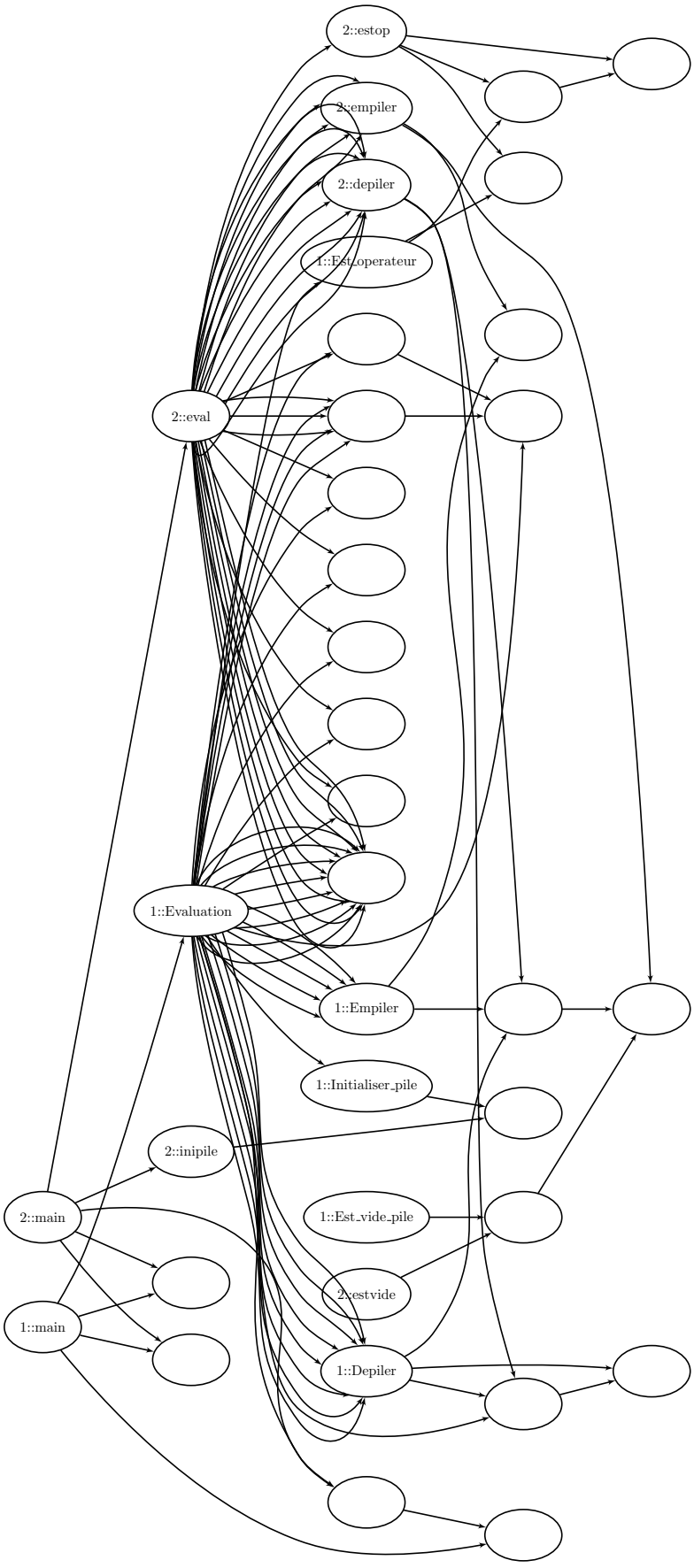


FIG. 7.10 – Graphe d'appels de fonctions factorisées pour deux projets plagiés de calculatrice

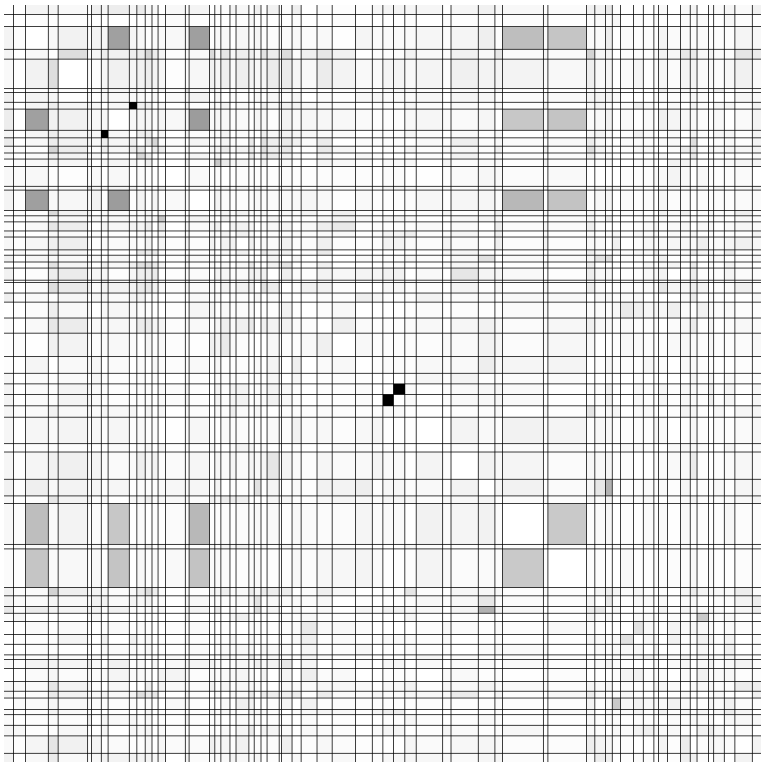


FIG. 7.11 – Carte de similarité des rendus de projets de jeu de serpent

selon leur chronologie de rendu, la distance d'une paire par rapport à la diagonale permet de déduire la proximité temporelle des rendus et ainsi repérer des similarités au sein ou entre promotions comme pour le 3^{e} rendu présentant une similarité importante avec deux rendus de la même promotion et deux autres rendus d'une promotion différente.

Des paires composées de projets différents ont également été comparées. Les rendus de projet de calculatrice de la promotion 2007 ont été confrontés à ceux de l'arbre syntaxique d'expression de la même promotion. Les fortes valeurs de similarité s'expliquent par l'utilisation d'une même structure de pile générique au sein des deux projets. Dans certains cas les étudiants ont choisi de réimplanter une nouvelle variante de pile. Contrairement à ces deux projets, le jeu de serpent et le compresseur ne disposent pas *a priori* de composantes pouvant être réutilisées. Les valeurs de similarité normalisées par rapport au projet le plus volumineux (généralement le compresseur) sont majoritairement inférieures à 3%. Lorsque nous comparons tous les projets étudiés à un autre projet externe quelconque (sans possibilité de modularisation de composante commune), les valeurs de similarité suivent la même répartition. Les quelques feuilles partagées sont de poids faible et représentent du code assez commun mais n'étant toutefois pas éliminé par le seuil de multiplicité maximale adopté (un exemple est présenté en figure 7.12).

On notera que l'on ne s'est pas intéressé ici à la similarité intra-projet qui pourrait également être utile afin d'évaluer des projets d'étudiants. Ceux-ci pourraient utiliser eux-mêmes un outil de recherche de similarité dans l'optique d'une amélioration de la modularité et généricité de leur code. Un tel service pourrait être intégré à une plate-forme de rendu de projets [136].

```
56   if ((chemin=(char *) malloc(sizeof(char)*j))==NULL) {
       libereArbreLexicographique(&dico);
       return 3;
```

(a) Projet externe

```
64  if (NULL==(plugin=(char *)malloc(sizeof(char)*MAX))){
65    perror("erreur_malloc_plugin\n");
      exit(EXIT_FAILURE);
```

(b) Rendu de calculatrice modulaire

FIG. 7.12 – Un exemple de clone commun (de multiplicité 5) partagé entre le projet externe et un rendu de la calculatrice modulaire

Itération	1	2	3	4	5	6	7
Nombre de feuilles (longueur $\geq t$)	8 877	17 589	8 494	4 915	4 171	3 999	3 969
Longueur cumulée des feuilles	493 590	340 321	123 829	60 282	48 436	45 942	45 548
Longueur du LCP maximal	535	147	96	96	21	13	11
Nombre de correspondances reportées	19 311	15 565	4 776	995	211	34	3
Longueur moyenne des correspondances	21,9	16,7	14,4	13,1	12,0	10,7	10,3
Temps d'exécution (en UA[a])	20,6	7,34	2,50	1,27	0,931	0,902	0,896

FIG. 7.13 – Caractéristiques d'itérations de factorisation

[a]Une UA équivaut à 1 seconde en mono-fil avec le JRE Sun 1.6 64 bits sur un CPU Intel P8600 2,4 Ghz (cache : 3 Mio, RAM : 4 Gio, ∼4787 bogomips).

7.8.2 Étude expérimentale de la factorisation et du graphe d'appel factorisé résultant

L'exécution de plusieurs itérations de factorisation permet la recherche de facteurs imbriqués. Le processus de factorisation s'achève lorsque le plus long facteur partagé sur les feuilles est de longueur inférieure au seuil de report t. Nous nous intéressons ici aux feuilles du graphe factorisé à l'issue de chaque itération. On rappelle que la mise en évidence à l'itération k d'une correspondance sur une feuille présente à la fin de l'itération $k-1$ conduit à la création d'une nouvelle feuille pour le sous-facteur correspondant ssi ce sous-facteur n'est pas une feuille entière : cette opération n'a pas d'incidence sur la longueur cumulée des feuilles. En revanche l'identification à une feuille entière permet de supprimer un exemplaire de cette feuille dans la fonction en cours de factorisation. Au cours des itérations, la longueur cumulée des feuilles diminue. Le graphe final ne comprend, parmi les feuilles de longueur d'au moins le seuil de report, aucune duplication de longueur au moins t. On notera que la taille du graphe rend son exploitation par visualisation globale ardue : toutefois, comme illustré en figure 7.10, le filtrage des nœuds atteints uniquement par quelques projets, lorsqu'ils sont de taille raisonnable, reste envisageable. D'autre part, même si le seuil de report t adopté est faible, on pourra choisir *a posteriori* d'ignorer des nœuds du graphe représentant des correspondances trop courtes.

7.9 Quelques perspectives

Des premiers résultats expérimentaux montrent un certain intérêt à l'utilisation de la méthode de factorisation présentée afin de modéliser les similarités existantes à l'échelle de la

fonction. L'utilisation de fonctions de chaînes de lexèmes permet de manipuler des projets par simple analyse lexicale suivie d'une analyse syntaxique basique pour délimiter les fonctions. Il reste toutefois possible de prendre en considération certaines informations sémantiques afin d'affiner les correspondances reportées. On pourra par exemple définir des lexèmes spéciaux délimitateurs afin de synchroniser le départ de correspondances rapportées et éviter le commencement par une fin d'instruction ou d'expression.

L'utilisation d'un alphabet de t-grams — dont il sera question au chapitre suivant — (pour un seuil de report de correspondances de t lexèmes) est une approche envisageable pour apporter une amélioration temporelle à l'exécution du processus de factorisation. Ceci permet de réduire de $t-1$ éléments la longueur des chaînes manipulées ce qui est appréciable lorsque nous manipulons de nombreuses chaînes telles que des petites fonctions feuilles. Les structures d'arbre des intervalles et graphe de farmax générées sont alors plus légères.

Enfin, il est important de souligner l'importance de la phase de suppression des chevauchements lors de la factorisation d'une fonction à partir de correspondances brutes issues de l'exploitation du graphe des farmax. Nous avons choisi une approche privilégiant les plus longues correspondances sur les plus courtes au détriment potentiel du taux de couverture des fonctions et de la minimisation du nombre de nouvelles fonctions externalisées. On notera que la factorisation d'une fonction est ainsi dépendante de son contexte : comparer des projets regroupés en une base disparate tel que réalisé en section précédente n'est donc pas nécessairement judicieux ; ceci pouvant entraîner la création de feuilles résiduelles de multiplicité unitaire minimisant la quantification de la similarité. Il serait intéressant de quantifier réellement cet effet. D'autre part l'usage de nouvelles méthodes d'élimination de chevauchement et leur comparaison expérimentale pourrait apporter un certain éclairage. Cela pourrait aboutir sur la possibilité d'utiliser un processus de factorisation sur une base incrémentale de projets.

8 Méta-lexémisation

Sommaire

La manipulation de longues chaînes de lexèmes peut être particulièrement contraignante pour les applications de recherches de similitudes sur des projets de taille importante. Une approche présentée dans le chapitre 6 consiste à utiliser des structures d'indexation de suffixes. Elle présente l'avantage de permettre la détection de similitudes avec un seuil minimal de lexèmes paramétrable mais au prix d'une utilisation importante de mémoire de masse, même pour les implantations les plus économes.

Afin de permettre un passage à l'échelle, l'usage d'empreintes afin de représenter des échantillons des séquences de lexèmes à indexer en base s'avère intéressant. Nous étudions tout d'abord la génération d'empreintes à partir de k-grams pour ensuite nous interroger sur les méthodes de sélection d'empreintes. Celles-ci permettent de choisir uniquement certaines empreintes de méta-lexèmes à référencer afin de diviser la taille de la base d'indexation par un facteur fixe. Ces empreintes sont ensuite utilisées pour la recherche de zones de similarité entre projets indexés et un projet requête.

Dans un dernier temps, nous nous intéressons à la recherche de correspondances exactement similaires et non chevauchantes entre séquences de lexèmes : à cet effet, nous utilisons une méthode de méta-lexémisation avec longueur de k-gram variable. Ceci est l'occasion de présenter la méthode de tuilage *Greedy String Tiling* [82] avec quelques améliorations que nous proposons afin d'améliorer la complexité temporelle dans certaines situations. Cette technique est alors adaptée à la recherche de correspondances sur une base de projets indexés.

8.1 Génération d'empreintes

8.1.1 Translation d'une chaîne en k-grams

Définition 8.1. Soit $t = t_1 t_2 \cdots t_n$ une chaîne de lexèmes de Σ. Nous définissons la chaîne de k-grams dérivée de t comme la chaîne $\mathcal{G}_k(t) = u = u_1 u_2 \cdots u_{n-k+1}$ avec :

$$\begin{array}{lcllll}
u_1 & = & t_1 & t_2 & \cdots & t_k \\
u_2 & = & t_2 & t_3 & \cdots & t_{k+1} \\
 & & & \cdots & & \\
u_i & = & t_i & t_{i+1} & \cdots & t_{i+k-1} \\
 & & & \cdots & & \\
u_{n-k+1} & = & t_{n-k+1} & t_{n-k+2} & \cdots & t_n
\end{array}$$

La représentation d'une chaîne en k-grams permet l'introduction d'un nouvel alphabet de méta-lexèmes de Σ^k pour représenter la chaîne originale. Les similitudes sont ensuite recherchées en utilisant le nouveau méta-alphabet de cardinalité plus importante mais dont la fréquence d'apparition de chaque méta-lexème est diminuée. Nous notons qu'aucune similitude portant sur des séquences identiques de lexèmes de taille inférieure à k ne peut plus alors être détectée.

Un mot sur la cardinalité de l'alphabet des k-grams La cardinalité de l'alphabet des k-grams est bornée par $|\Sigma|^k$; cependant parmi tous les k-uplets de lexèmes, seule une faible proportion représentent des successions de lexèmes grammaticalement et sémantiquement admissibles. Ainsi, par exemple en langage C, le 2-gram "({" est interdit alors que le 2-gram "){" peut être rencontré. Les k-grams grammaticalement valides peuvent ainsi être énumérés par analyse de la grammaire pour des valeurs faibles de k. Nous présentons en figure 8.1 le nombre de k-grams différents pour un projet Java de volume important (OpenJDK 1.6 avec environ 1 million de lignes de code) ainsi que la répartition de leur fréquence. Ayant considéré les lexèmes obtenus après parcours en profondeur de l'arbre de syntaxe concret, nous constatons qu'il existe $|\Sigma| = 154$ 1-grams, 1670 2-grams, 7480 3-grams et 20666 4-grams[1] dont le nombre n'augmente que peu avec le volume de code analysé. En revanche le nombre de k-grams distincts pour des valeurs plus élevées de k est approximativement linéaire avec le volume de code traité : il s'établit à $N - k + 1 - d$, d étant le nombre de k-grams dupliqué (avec typiquement $d \longrightarrow 0$ lorsque $k \longrightarrow N$).

8.1.2 Hachage des k-grams

Introduction

Étant donné l'espace des k-grams Σ^k issu de l'alphabet de lexèmes Σ, nous souhaitons représenter chaque k-gram par un entier. À cet effet, nous introduisons une fonction de hachage réalisant cette correspondance. Si l'alphabet Σ est de cardinalité finie, il est possible d'utiliser la fonction de hachage bijective fp_b suivante :

$$fp_b : (g_1, g_2, \cdots, g_k) \longrightarrow b^{k-1}g_1 + b^{k-2}g_2 + \cdots + b^0 g_k$$

où $b = |\Sigma|$ et les lexèmes $g_1, g_2, \cdots, g_k$ de Σ sont confondus avec une valeur entière les représentant en utilisant un ordre total arbitraire sur Σ. Il peut cependant être avantageux de limiter l'étendue de l'intervalle des entiers représentatifs en sacrifiant l'injectivité de la fonction de hachage afin de limiter l'espace mémoire occupé par chaque valeur de hachage. Afin de tester si deux valeurs de hachage désignent le même k-gram, il est alors nécessaire d'accompagner chacune de ces valeurs d'un lien vers l'occurrence de k-gram qu'elle désigne afin de pouvoir vérifier si deux valeurs de hachage sont des faux-positifs ou non.

Nous pouvons toutefois noter que seule une certaine portion de l'espace Σ^k des k-grams est intersectée par des k-grams en pratique. Cela peut rendre intéressant l'ajout d'un niveau d'indirection pour la représentation des k-grams. Ceux-ci sont indexés dans une base par une valeur de hachage non obligatoirement bijective ou un arbre lexicographique et associés à une valeur entière bijective pour k petit.

[1] Les valeurs de cet exemple sont à comparer avec les puissances de $|\Sigma|$: $|\Sigma|^2 = 23716$, $|\Sigma|^3 \ 3,65 \cdot 10^6$ et $|\Sigma|^4 \ 562 \cdot 10^6$.

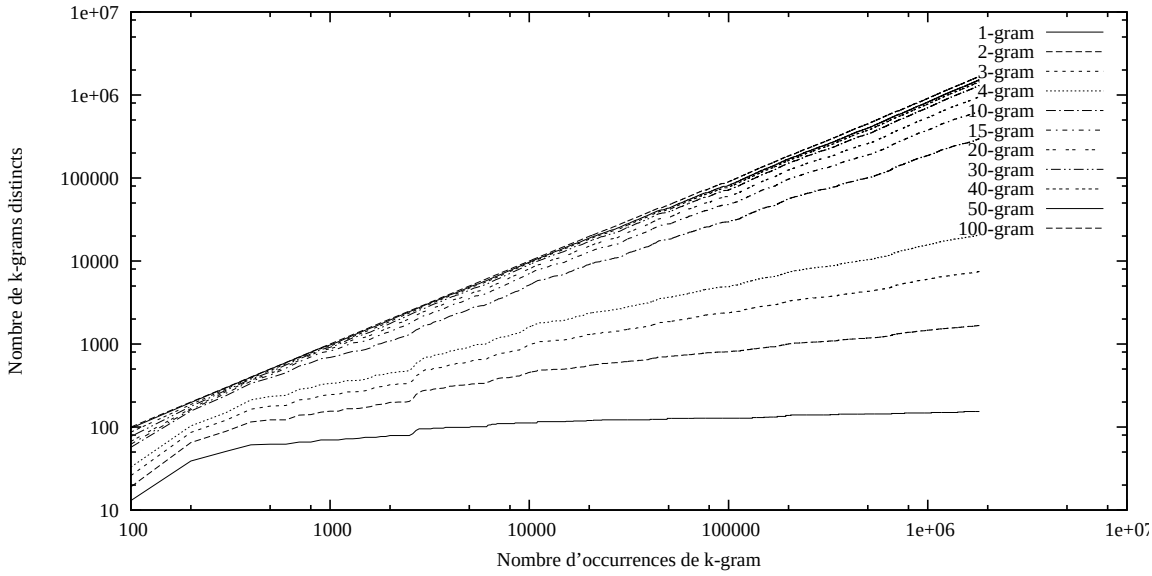


FIG. 8.1 – Nombre de k-grams distincts pour le paquetage *java* du projet OpenJDK 1.6

Hachage incrémental de Karp-Rabin

Calculer indépendamment les valeurs de hachage des $N - k + 1$ k-grams d'une séquence de N lexèmes est de complexité temporelle dépendante de k (typiquement $\Theta(k)$ pour chaque k-gram). Nous souhaitons par une méthode incrémentale de hachage calculer l'ensemble des valeurs de hachages en temps $\Theta(N)$.

Choix de la base En utilisant la fonction de hachage bijective $fp_{|\Sigma|}$ polynomiale présentée précédemment, nous pouvons calculer chaque valeur de hachage de k-grams en temps $O(k)$ (seules k additions et $k-1$ multiplications sont nécessaires). Nous limitons la cardinalité de l'espace de hachage à N (en pratique une puissance de 2) en considérant la fonction non injective $fp_{|\Sigma|} \mod N$. Dans cette situation, l'utilisation de la base $b = |\Sigma|$ peut s'avérer peu judicieuse en particulier si N et b présentent des facteurs communs : la surjectivité de f_b est alors altérée. Les possibilités de collisions pour des k-grams différents sont ainsi augmentées. Ainsi, par exemple, si b et N sont pairs, $f_b \mod N$ est à valeurs paires. b et N doivent être premiers entre eux ce qui est assuré par le choix d'un entier b premier. Les bases les plus couramment utilisées sont 33 et 65599 [54].

Incrémentalités

Définition 8.2. *(Incrémentalité simple)* Une fonction de hachage f sur des k-grams est dite incrémentale si le k-gram d'indice i ($i > 1$) est calculable à partir de la donnée du k-gram précédent d'indice $i-1$ en un temps constant (indépendant de la valeur de k).

La fonction de hachage polynomiale fp_b est effectivement incrémentale. Connaissant la valeur de hachage de $u_i = t_i t_{i+1} \cdots t_{i+k-2} t_{i+k-1}$, la valeur de hachage du k-gram $u_{i+1} = t_{i+1} t_{i+2} \cdots t_{i+k-1} t_{i+k}$ peut être calculée ainsi en temps constant :

$$fp_b(u_{i+1}) = u_{i-1} \times b - t_i \times b^k + t_{i+k}$$

La séquence des valeurs de hachage de la séquence de k-grams $\mathcal{G}_k(t)$ est ainsi calculable en temps $O(n)$. Il est alors possible de rechercher un facteur de k lexèmes sur une chaîne en temps linéaire indépendant de k en calculant les valeurs de hachage de tous ses k-grams comme l'ont proposé Karp et Rabin [53]. Nous présentons en figure 8.2 un exemple de hachage incrémental d'une chaîne de lexèmes après attribution d'une valeur numérique à chaque lexème.

Définition 8.3. *(Incrémentalité forte)* Une classe de fonctions de hachage $F = \{f_1, f_2, \cdots\}$ sur des $1, 2, \cdots$-grams est dite fortement incrémentale si pour $k = k' + k''$ ($1 \leq k' \leq k''$) il existe une fonction g permettant le calcul du k-gram $u_i u_{i+1} \cdots u_{i+k-2} u_{i+k-1}$ à partir des k'-gram et k''-gram $u_i u_{i+1} \cdots u_{i+k'-1} u_{i+k'-2}$ et $u_{i+k'} u_{i+k'+1} \cdots u_{i+k-2} u_{i+k-1}$ avec une complexité temporelle de $o(k)$ (par exemple $\Theta(\log k)$).

Pour un facteur $u = vw$ ($|v| = k', |w| = k''$), il est possible de calculer la valeur de hachage de u en connaissant celles de v et w avec la classe de fonctions de hachage polynomiales fp_b avec une complexité temporelle meilleure que $O(|u|)$. En effet, nous pouvons constater l'égalité suivante :

$$fp_b(u) = fp_b(vw) = fp_b(v) * b^{k''} + fp_b(w)$$

Lexème	for	lpar	id	eq	0	;	lt	MAX	++	rpar
Valeur	1	2	3	4	5	6	7	8	9	10

(a) Identificateurs numériques des lexèmes

4-gram u_i	Calcul de $h(u_i)$	Valeur de hachage ($b = 33$, mod 16)
$u_1 =$ for lpar id eq	$((v(\text{for}) * b + v(\text{lpar})) * b + v(\text{id})) * b + v(\text{eq})$	10
$u_2 =$ lpar id eq 0	$h(u_1) * b - v(\text{for}) * b^4 + v(0)$	14
$u_3 =$ id eq 0 ; ;	$h(u_2) * b - v(\text{lpar}) * b^4 + v(;)$	2
$u_4 =$ eq 0 ; id ;	$h(u_3) * b - v(\text{id}) * b^4 + v(\text{id})$	2
$u_5 =$ 0 ; id lt	$h(u_4) * b - v(\text{eq}) * b^4 + v(\text{lt})$	5
$u_6 =$; id lt MAX	$h(u_5) * b - v(0) * b^4 + v(\text{MAX})$	8
$u_7 =$ id lt MAX ;	$h(u_6) * b - v(;) * b^4 + v(;)$	8
$u_8 =$ lt MAX ; id	$h(u_7) * b - v(\text{id}) * b^4 + v(\text{id})$	8
$u_9 =$ MAX ; id ++	$h(u_8) * b - v(\text{lt}) * b^4 + v(++)$	10
$u_{10} =$; id ++ rpar	$h(u_9) * b - v(\text{MAX}) * b^4 + v(\text{rpar})$	12

(b) Valeurs de hachage des 4-grams

FIG. 8.2 – Séquence de $k = 4$-grams u et valeurs de hachage de la chaînes de lexèmes for *lpar id eq 0 ; id lt MAX ; id ++ rpar*, $b = 33$

De manière générale, le calcul de $fp_b(u)$ à partir de $fp_b(v)$ et $fp_b(w)$ est réalisable en temps équivalent à $\log_2 k''$ afin de réaliser l'exponentiation rapide de b. Cependant, si l'on suppose k'' fixé, le coût initial de l'exponentiation est amorti sur toutes les instances calculées.

8.2 Sélection d'empreintes

Le souci d'économie de mémoire de masse pour le stockage de la base des empreintes de k-grams nous conduit à ne prélever qu'un échantillon représentatif d'empreintes. Nous utilisons à cet effet des méthodes de sélection d'empreintes dont certaines sont décrites ci-après.

8.2.1 Quelques définitions

Définition 8.4. *(Fonction de sélection d'empreintes)* Soit une chaîne de méta-lexèmes $u_1 u_2 \cdots u_n$ représentées par des empreintes que nous confondons avec les méta-lexèmes. Une fonction de sélection d'empreintes S associe à la sous-chaîne $u_i u_{i+1} \cdots u_j$ une sous-séquence de q méta-lexèmes sélectionnés $S(u_i u_{i+1} \cdots u_j) = u_{p_1} u_{p_2} \cdots u_{p_q}$ avec $i \leq p_1 \leq p_2 \leq \cdots \leq p_q \leq j$. On note $\rho = \frac{q}{j-i+1}$ la densité de sélection de cette fonction sur $u_i \cdots u_j$.

Définition 8.5. *(Seuil de garantie)* Une fonction de sélection d'empreintes de k-grams présente un seuil de garantie $t = l + k - 1$ ssi il existe au moins une empreinte sélectionnée pour toute chaîne de l k-grams (représentant une chaîne sous-jacente de $l + k - 1$ lexèmes).

Définition 8.6. *(Localité)* Une fonction de sélection d'empreintes S est locale ssi pour toutes chaînes $u_i u_{i+1} \cdots u_j$ et toutes chaînes contextuelles de méta-lexèmes c_1, c_2, c_3, c_4, $S(c_1 u_i u_{i+1} \cdots u_j c_2)$ et $S(c_3 u_i u_{i+1} \cdots u_j c_4)$ sélectionnent les mêmes méta-lexèmes centraux de $u_i \cdots u_j$ (une des sélections pouvant comporter des méta-lexèmes supplémentaires sélectionnés au début et/ou à la fin de $u_i \cdots u_j$).

Fonction	Seuil de garantie	Densité de sélection	Localité
Sélection aléatoire (8.2.3)	Probabiliste	ρ fixé	Oui
Sélection par position (8.2.4)	$\frac{1}{\rho} + k - 1$	ρ fixé	Non
Sélection sur fenêtre glissante de l lexèmes...			
...d'empreintes minimales (8.2.5)	$l + k - 1$	En moyenne $\rho = \frac{2}{l+1}$	Oui
...d'empreintes les moins fréquentes (8.2.5)			Possible[a]

[a]Présente un caractère local ssi les fréquences sont figées.

FIG. 8.3 – Caractéristiques principales de fonctions de sélection

Une fonction de sélection idéale serait donc locale pour produire un résultat de sélection homogène pour une même sous-chaîne au sein de contextes différents, permettrait de minimiser le seuil de garantie pour permettre une détection de petits facteurs similaires tout en minimisant la densité de sélection. Un récapitulatif des différentes fonctions introduites ci-après est présenté avec leurs principales caractéristiques en figure 8.3

8.2.2 Filtrage syntaxique préalable

Aussi bien la langue naturelle que le code source respectent une certaine construction syntaxique. L'utilisation de séquences de lexèmes fait perdre toute information syntaxique. Ainsi la génération de suites de k-grams à partir de telles séquences de lexèmes peut faire apparaître des k-grams non signifiants. Par exemple pour le début de ce paragraphe, un 4-gram chevauchant la première et la deuxième phrase et étant souligné est composé des lexèmes (`construction,syntaxique,.,l'`). Pour un code source, il s'agirait de k-grams englobant plusieurs structures syntaxiques (intersectant deux instructions par exemple). Si une analyse syntaxique permet de délimiter précisément les entités syntaxiques, il est également possible, dans le cadre d'une approche moins coûteuse, de définir des lexèmes interdits Σ_f pour les k-grams (typiquement des symboles de fin d'instruction ou de délimitation de blocs). Tout k-gram contenant au moins un lexème interdit est alors ignoré. Cette remarque n'est valable que pour des valeurs faibles de k n'englobant que des fractions d'unité syntaxique élémentaire (par exemple l'instruction) : pour des valeurs plus importantes de k il est nécessaire de considérer des unités syntaxiques de plus haut niveau. Si l'on souhaite alors supprimer des k-grams chevauchant plusieurs unités, une analyse syntaxique légère peut être nécessaire.

8.2.3 Sélection aléatoire

Afin de réduire le volume d'empreintes à indexer, une première approche consiste à sélectionner un ratio ρ d'empreintes à conserver à l'aide d'un générateur de nombres pseudo-aléatoire[2]. Cette approche présente néanmoins l'inconvénient majeur de ne pas garantir qu'un certain facteur de la séquence de lexèmes, quel que soit sa longueur, puisse être représenté par au moins une empreinte de k-grams. Dans cette situation le facteur devient « invisible » pour un procédé de recherche de similitudes. Cette situation peut être rencontrée lorsque la dispersibilité de la fonction de hachage choisie est mauvaise.

[2]Considérant une bonne dispersibilité de la fonction de hachage, il est également possible de ne garder que les valeurs de hachage dont le modulo par $1/\rho$ est nul.

8.2.4 Sélection par position

Une autre méthode simple de sélection consiste à ne conserver que les empreintes de k-grams débutant aux positions $0 \mod (1/\rho)$ de la séquence de lexèmes afin d'obtenir un ratio de sélection de ρ. Cela permet également de garantir que tout facteur de plus d'au moins $\frac{1}{\rho} + k - 1$ lexèmes puisse être représenté par au moins une empreinte sélectionnée. Cette méthode présente néanmoins l'inconvénient de ne pas être locale (un décalage d'une sous-chaîne produisant des sélections différentes).

8.2.5 Sélection sur fenêtre

Les méthodes de sélection d'empreintes sur fenêtre ont pour objectif de garantir qu'au-delà d'une certaine longueur, tout facteur sera représenté par au moins une empreinte sélectionnée tout en conservant un mode de sélection local, indépendant du contexte global.

Principe Les méthodes de sélection d'empreintes sur fenêtre utilisent une fenêtre glissante sur la séquence d'empreintes issues du hachage de k-grams. L'objectif est alors de sélectionner sur chaque position de la fenêtre un nombre fixe d'empreintes. Ainsi, sur une fenêtre de l empreintes de k-grams, s'il est décidé de sélectionner α empreintes, nous pouvons garantir que toute portion de $k + l - 1$ lexèmes de la séquence originelle est représentée exactement par α empreintes de k-grams. Pour un processus futur de recherche de similitudes, k est considéré comme le seuil de bruit — aucune similarité portant sur moins de k lexèmes ne peut être détectée — tandis que $k + l - 1$ est le seuil de garantie de détection — aucune similarité portant sur au moins $l + k - 1$ ($\alpha \geq 1$) lexèmes ne peut être ignorée car représentée par au moins α empreintes —. Il reste à définir une fonction $f : N^l \longrightarrow [1..l]^\alpha$ de sélection sur fenêtre. Typiquement une telle fonction a un coût temporel en $O(l)$ pour son premier appel : les appels suivants sur les fenêtres translatées pourrait être menés incrémentalement en temps constant (selon la méthode de sélection).

Fonction de sélection arbitraire

La méthode de sélection sur fenêtre Winnowing [56] implantée par Moss [94] utilise une fonction de sélection arbitraire se basant sur la valeur de l'empreinte : par exemple sur chaque position de fenêtre, la fonction de sélection choisit les α (généralement $\alpha = 1$) empreintes de valeur minimale. En cas d'égalité, les empreintes de valeur minimale sont départagées par leur position (choix de l'empreinte de plus grande position). La figure 8.4 présente un exemple de sélection sur fenêtre. Dans le pire des cas, si la suite des valeurs de hachage est strictement croissante, la densité de sélection ρ (ratio du nombre d'empreintes sélectionnées sur le nombre total d'empreintes) est de 1, l'empreinte sélectionnée à chaque position de fenêtre étant la première. [56] montre que la densité moyenne de sélection sur fenêtre pour un critère parfaitement aléatoire est de $\rho = \alpha\frac{2}{l+1}$. Un processus de sélection aléatoire d'une empreinte de probabilité ρ permet d'obtenir la même densité moyenne de sélection mais sans seuil de garantie (il existe une probabilité $(1 - \rho)^i$ qu'une séquence de i k-grams ne soit représentée par aucune empreinte). D'autre part, cette méthode de sélection est bien locale, la sélection étant réalisée sur des fenêtres avec un critère de sélection indépendant du contexte.

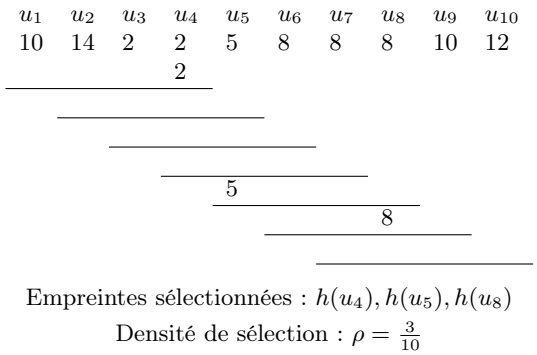


FIG. 8.4 – Sélection d'empreintes sur fenêtre de type Winnowing ($\alpha = 1$)

Fonction de sélection objective

Une méthode de sélection plus objective que le choix des empreintes de valeur de hachage minimale peut consister à considérer des propriétés statistiques des empreintes. En particulier, nous pouvons précalculer sur des corpus de code source conséquents les fréquences d'apparition de certains k-grams ou alors les déterminer incrémentalement durant le processus d'indexation. Il est alors judicieux de sélectionner sur fenêtre les empreintes correspondant aux k-grams les moins fréquents car étant plus caractéristiques. Une séquence de lexèmes est alors représentée par les k-grams les plus spécifiques ce qui permet de limiter les résultats faux-positifs lors de requêtes sur bases d'empreintes tout en maintenant un seuil de garantie. Ce procédé est notamment utilisé par [77] mais uniquement pour extraire les empreintes de k-grams les plus rares d'instructions individuelles et non à l'aide d'une fenêtre glissante.

On note que si l'usage de fréquences d'apparition constantes permet d'obtenir une méthode de sélection locale, ce n'est pas le cas lorsque les fréquences sont variables (calcul incrémental).

8.3 Recherche de correspondances sur base d'empreintes

8.3.1 Indexation des empreintes

Des codes sources peuvent être indexés en base par leurs empreintes caractéristiques générées. Ces empreintes, représentées par la valeur de hachage de k-grams, peuvent être classées par l'usage d'une structure dynamique de tri. L'usage d'un arbre binaire de recherche équilibré peut nécessiter jusqu'à $\log_2 n$ lectures ou écritures de blocs pour l'accès, l'ajout ou la suppression d'un élément avec une importante fragmentation mémorielle liée à la suppression ou mise à jour d'éléments. L'usage de structures d'arbres k-aires telle que le k-B+-tree [6], populaire pour l'implantation d'index de bases de données généralistes, limite le nombre d'opérations d'entrée-sortie coûteuses ($\log_k n$ pour n empreintes indexées) ainsi que la fragmentation. Le k-B+-tree est un arbre k-aire dont chaque nœud est contenu dans un bloc : les nœuds internes possèdent entre $k/2$ et k nœuds enfants indexés par des clés tandis que les noeuds feuilles disposent de $k/2$ à k empreintes. Cependant, aucune relation de séquentialité ne peut être inférée

à partir de l'arbre de tri sur la valeur de hachage contrairement à une structure d'indexation de suffixes.

L'usage d'une structure dynamique offre la possibilité de réaliser des modifications sur la base de projets ce qui peut être nécessaire pour la recherche de similitudes au sein de projets en cours de développement. Nous pouvons ainsi aussi bien ajouter que supprimer des empreintes correspondant à certaines unités du code.

8.3.2 Recherche d'un projet requête dans une base de projets

Étant donnée une unité structurelle requête q (pouvant être un projet, une unité de compilation, une fonction, un bloc de code...), nous cherchons à déterminer les clones (exacts ou approchés) entre ce projet et la base d'unités structurelles déjà indexées $B = p_1 \cdots p_k$. Nous déterminons tout d'abord la séquence $\mathcal{F}(q) = f_1 f_2 \cdots f_m$ d'empreintes de q. Chacune des valeurs de hachage de ces empreintes est recherchée sur la structure d'indexation de la base : des valeurs de hachage concordantes signifient que les deux empreintes représentent potentiellement des morceaux de code similaires. Il est néanmoins nécessaire de vérifier si l'égalité des valeurs de hachage n'est pas liée à une collision induisant un faux-positif : le code sous-jacent représenté doit donc être comparé. Si l'espace de hachage est suffisamment important, cette étape peut être évitée. Nous faisons pour hypothèse, pour la suite, de l'usage d'une fonction de sélection locale, i.e. insensible au contexte.

Nous introduisons en figure 8.5 un exemple d'unité requête ainsi que d'unités indexées en base qui nous servira d'illustration aux méthodes de recherche de similarité décrites ci-après.

$$
\begin{array}{lcl}
q & = & a\mathbf{b}c\mathbf{d}e\mathbf{fg}h \\
p_1 & = & \mathbf{bi}a\mathbf{d}c\mathbf{gf}h \\
p_2 & = & \mathbf{j}a\mathbf{b}c\mathbf{de}j\mathbf{k} \\
p_3 & = & a\mathbf{b}i\mathbf{d}i\mathbf{f}j\mathbf{g}
\end{array}
$$

FIG. 8.5 – Exemple d'unité requêtes et d'unités indexées en base. $a..j$ est l'alphabet des méta-lexèmes utilisés (à des fins de simplification, les méta-lexèmes sont identifiés à des $k = 1$-grams). Une méthode de sélection locale (non explicitée) sur une fenêtre de taille $k + 2$ garde les méta-lexèmes spécifiés en gras.

Détermination de métrique de similarité

Finalement nous obtenons pour chaque unité structurelle de la base p_i la liste exhaustive des empreintes $\mathcal{F}(p_i) \cap \mathcal{F}(q)$ représentant des morceaux de code similaires avec des empreintes de q. Nous pouvons ainsi en déduire une métrique de similarité ensembliste (aucune relation de séquentialité n'étant considérée sur les empreintes) basées sur le volume du code représenté par $\mathcal{F}(p_i) \cap \mathcal{F}(q)$ comparé à $\mathcal{F}(p_i)$ et $\mathcal{F}(q_i)$ comme expliqué au chapitre 12.

La métrique de similarité ainsi déduite présente une résistance intéressante aux opérations d'édition par transposition au sein des unités structurelles mais peut occasionner des cas de fausse-positivité. L'utilisation d'une telle métrique peut alors être vue comme une technique

de présélection d'unités structurelles intéressantes de la base afin de les comparer plus intensivement par l'usage, par exemple, d'algorithmes d'alignement local (voir chapitre 5), soit sur les lexèmes, soit les méta-lexèmes.

Si nous considérons l'exemple indiqué en figure 8.5, nous établissons le vecteur des méta-lexèmes partagés entre q et les p_i ainsi que le vecteur des volumes correspondants (chaque méta-lexème ayant un volume de 1) :

$\mathcal{F}(q) \cap ...$	p_1	p_2	p_3
Empreintes (méta-lexèmes)	$bdgf$	bd	$bdfg$
Volume	4	2	4

FIG. 8.6 – Vecteur de méta-lexèmes partagés entre paires d'unités

Nous normalisons ensuite chacune des valeurs par rapport à l'union des empreintes de la paire d'unités correspondante : ainsi $s(q, p_i) = \frac{\mathcal{V}(\mathcal{F}(q) \cap \mathcal{F}(p_i))}{\mathcal{V}(\mathcal{F}(q)) + \mathcal{V}(\mathcal{F}(p_i)) - \mathcal{V}(\mathcal{F}(q) \cap \mathcal{F}(p_i))}$. D'où les valeurs de similarités suivantes :

i	1	2	3
$s(q, p_i)$	$\frac{4}{4+5-4} = \frac{4}{5}$	$\frac{2}{4+5-2} = \frac{2}{7}$	$\frac{4}{4+4-4} = 1$

FIG. 8.7 – Similarité normalisée (unioniste) entre paires d'unités

Nous constatons que la plus forte similarité unioniste concerne la paire (q, p_3) : en effet q et p_3 sont caractérisés par les mêmes méta-lexèmes sélectionnés avec des espaces lacunaires dissemblables. p_1 est également semblable à q avec des méta-lexèmes apparaissant dans un ordre différent. Quant à p_2 il présente le plus long facteur commun avec q ($abcde$) mais la plus faible similarité.

Recherche de correspondances exactes

Comme vu précédemment, la détermination d'une métrique de similarité sur unité structurelle peut être utilisée comme technique de présélection de paires à examiner de façon approfondie. Nous nous intéressons ici à la recherche de correspondances exactes, i.e. des occurrences de facteurs égaux, à partir de la connaissance du jeu d'empreintes de k-grams de chaque unité de la base en intersection avec les empreintes de q, $\mathcal{F}(q)$.

Pour ce faire, nous marquons toutes les empreintes de $\mathcal{F}(q)$ trouvées au moins sur une unité p_i de la base. Ces empreintes sont triées par position sur q pour obtenir la séquence s_q. De même, pour chaque unité p_i, les empreintes en intersection avec $\mathcal{F}(q)$ sont triées par position des occurrences sur p_i pour obtenir la séquence s_{p_i}. Les séquences s_q et leurs homologues de la base s_{p_i} sont ensuite découpées en sous-chaînes d'empreintes représentant des méta-lexèmes consécutifs : $u_{i,1}, u_{i,2}, \cdots, u_{i,n}$.

En suivant l'exemple introduit en figure 8.5, nous pouvons relever :

L'ensemble des chaînes d'empreintes consécutives des unités de la base et de l'unité requête doivent ensuite être analysées afin d'y trouver des facteurs répétés comprenant au moins une occurrence de facteur de l'unité requête.

$$\begin{aligned} s_q &= bdfg \\ s_{p_1} &= b, dgf \\ s_{p_2} &= bd \\ s_{p_3} &= bdfg \end{aligned}$$

FIG. 8.8 – Sous-chaînes de méta-lexèmes partagés

Il est néanmoins nécessaire de noter que deux sous-chaînes d'empreintes consécutives à valeurs de hachage identiques peuvent ne pas représenter des k-grams et des lexèmes sous-jacents identiques. Soient γ et γ' de telles chaînes d'empreintes de méta-lexèmes représentant des chaînes de lexèmes sous-jacentes $\alpha[i..i']$ et $\beta[j..j']$. L'égalité de γ et γ' implique que les méta-lexèmes sélectionnés soient identiques sous réserve d'une fonction de hachage n'engendrant pas de collision. Or l'espace des méta-lexèmes est potentiellement de cardinalité plus importante que celui des valeurs de hachage générées, la fonction de hachage utilisée n'est donc pas injective et peut engendrer des collisions. Dans l'exemple traité, nous avons choisi d'ignorer cette possibilité en considérant des 1-grams (lexèmes élémentaires) en bijection avec leurs empreintes.

D'autre part, rien ne garantit pour une fenêtre de sélection de taille supérieure à 1, que les lexèmes compris entre deux k-grams consécutifs sélectionnés soient égaux : il est donc nécessaire de les vérifier. Pour s'en convaincre, on pourra considérer l'unité p_3 de l'exemple traité : $s_q = s_{p_3}$, cependant les méta-lexèmes intermédiaires entre ceux sélectionnés diffèrent.

Nous soulignons également que si la chaîne de méta-lexèmes γ n'est, par définition, ni extensible sur la gauche, ni sur la droite[3], alors s'il existe une correspondance exacte avec q impliquant γ, celle-ci possède une longueur en nombre de lexèmes comprise entre $i' - i + 1$ et $i' - i + 1 + 2(l-1)$ (l étant la taille de la fenêtre de sélection). En effet, la correspondance trouvée inclut nécessairement $\alpha[i..i']$: s'il existait une correspondance incluant également l lexèmes à gauche de cette chaîne, au moins une empreinte de k-gram aurait été sélectionnée sur la fenêtre de position $[i - l..i - 1]$. Par contre si une correspondance incluait $l - 1$ lexèmes à gauche, il est possible qu'aucune empreinte k-gram ne commençant aux positions $[i - l + 1..i - 1]$ soit sélectionnée, ce cas survenant uniquement si l'empreinte de k-gram de position i est localement minimale. Une remarque similaire s'applique pour les lexèmes à droite. Pour l'exemple traité, la sous-chaîne bd de p_2 est effectivement impliquée dans une correspondance exacte avec q. Un méta-lexème concordant c est présent entre b et d et une extension de un méta-lexème peut être réalisée aussi bien sur la gauche que la droite pour obtenir la correspondance exacte $abcde$.

De ces observations, nous créons une structure d'indexation de suffixes pour les chaînes de lexèmes représentées par les chaînes d'empreintes consécutives de valeurs de hachage comprises dans $\mathcal{F}_q$. Ces chaînes sont augmentées de $l - 1$ lexèmes à gauche et à droite afin de compenser l' « aveuglement » possible dû à la procédure de sélection.

[3]Ce qui signifie que les empreintes précédant et suivant γ n'ont pas de valeur de hachage commune avec celles de $\mathcal{F}(q)$.

Pour l'exemple de la figure 8.5, nous obtenons en utilisant les sous-chaînes de méta-lexèmes sélectionnées les chaînes de lexèmes développées suivantes (on rappelle que l'on manipule des 1-grams à des fin de simplification) :

$$\begin{aligned} s_q &= abcdefgh \\ s_{p_1} &= bi, adcgfh \\ s_{p_2} &= abcde \\ s_{p_3} &= abidifjg \end{aligned}$$

FIG. 8.9 – Chaînes de lexèmes développées à indexer

Nous pouvons ainsi utiliser une table de suffixes et un arbre d'intervalle afin de déceler les facteurs répétés de lexèmes à partir des chaînes de lexèmes précédemment obtenues et ainsi calculer leur farmax. Cette procédure garantit que les facteurs répétés de longueur supérieure au seuil de détection de $l + k - 1$ lexèmes (générant au moins une empreinte de k-gram sélectionnée) soient exhaustivement reportés. Cette remarque n'est cependant valide que si le critère de sélection des empreintes est cohérent pour l'ensemble des unités indexées de la base et de l'unité requête. Ainsi, si le critère de sélection est basé sur la fréquence d'apparition du k-gram, le vecteur de fréquence d'apparition doit être constant et non évolutif pour chaque unité indexée.

Pour l'exemple cité, nous pouvons calculer le farmax des chaînes de lexèmes d'intérêt trouvées en figure 8.9. Celui-ci comporte deux facteurs de volume d'au moins $k + 1$ ($k = 1$) : *abcde* et *ab*.

La méthode exposée peut donc être rapprochée d'une indexation exhaustive de tous les suffixes des unités manipulées. L'utilisation d'empreintes de k-grams sélectionnés et indexés permet cependant de ne construire qu'à la demande une structure d'indexation de suffixes (telle qu'une table de suffixes) portant uniquement sur des zones dont la similarité est fortement suspectée par l'existence de k-grams communs. L'utilisation d'un critère de sélection pertinent telle que la fréquence des k-grams permet de réduire la fausse positivité liée à des séquences d'empreintes de k-grams sélectionnés similaires. Le volume de la base d'empreintes de k-grams sélectionnés sur fenêtre de l lexèmes est de l'ordre de $N(b + \log_2(N))/l$ bits pour des empreintes de b bits avec des unités totalisant N lexèmes alors qu'un arbre de suffixes incrémental nécessiterait au moins $10N$ octets. Toutefois, contrairement à un arbre de suffixes dont le seuil de détection peut être paramétré à l'utilisation, aucun facteur répété de longueur inférieure à k lexèmes ne peut être trouvé et des facteurs de taille inférieure à $l+k-1$ lexèmes peuvent être potentiellement ignorés.

Les paires d'unités syntaxiques présentant au moins deux facteurs répétés communs peuvent ensuite faire l'objet d'un procédé d'extension des correspondances exactes pour créer des correspondances approchées consolidées en utilisant des méthodes d'alignement local présentées en 5.4. Si ces méthodes permettent de consolider des macro-similarités, elles ne permettent toutefois pas de pallier l'éventuelle absence de détection de correspondances liée à la présence d'opérations d'édition très localisées portant sur quelques lexèmes. De telles opérations peuvent modifier quelques empreintes de k-grams sur une fenêtre de sélection et donc *in fine* les empreintes sélectionnées ou alors au contraire n'avoir d'impact que sur des k-grams non

sélectionnés. Dans ce dernier cas de figure, il pourrait être envisageable de comparer par une méthode d'alignement les paires de chaînes sous-jacentes aux séquences d'empreintes de k-grams similaires. Un obfuscateur cherchera quant à lui à minimiser les opérations d'édition tout en maximisant l'impact de ces modifications sur les empreintes sélectionnées. Il est alors préférable de choisir une fonction de hachage utilisant une clé secrète (telle que la base pour une fonction de hachage polynomiale) pour la génération d'empreintes.

Utilisation d'une fonction de sélection non-locale

Une fonction de sélection non-locale n'est pas déterministe par la seule considération du contexte local de sélection. Ainsi si deux facteurs exacts de chaînes de lexèmes au sein d'unités différentes conduisent à l'obtention de mêmes méta-lexèmes, les empreintes de méta-lexèmes sélectionnées peuvent différer. Par exemple si nous considérons la fonction sélectionnant les méta-lexèmes de position $1 \mod 2$ sur les unités q et p_2 de la figure 8.5, nous obtenons les méta-lexèmes sélectionnés *aceg* pour q et *jbdj* pour p_2. Ces deux unités ne partageant aucun méta-lexème, aucune similarité les concernant ne peut être relevée. Il est donc nécessaire, lorsque nous utilisons une fonction non-locale de ne pas opérer de sélection sur l'unité requête pour conserver l'ensemble de ses méta-lexèmes. Nous pouvons alors effectivement relever la sous-chaîne de méta-lexèmes communs *bde* entre q et p_2 et ainsi indexer les suffixes sur les chaînes de lexèmes sous-jacentes.

Cette solution offre plus de flexibilité car permettant l'usage de fonctions de sélection diverses au sein d'une même base mais multiplie le nombre de requêtes nécessaires puisque tous les méta-lexèmes de l'unité requêtes doivent être recherchés en base. Il devient également impossible de rechercher des similarités entre des unités internes à la base sans recalcul de l'ensemble des empreintes.

8.4 Méta-lexémisation avec k-grams de taille variable

8.4.1 Philosophie

Méta-lexémiser un code source représenté sous la forme de séquence de lexèmes en k-grams nous contraint à la recherche de similitudes à k-lexèmes. Il n'est alors plus possible de rechercher des correspondances de taille inférieure et la recherche de clones de taille $k' > k$ lexèmes nécessite de rechercher $(k' - k + 1)$ k-grams consécutifs identiques ce qui est contraignant pour des valeurs élevées de k'.

Une idée envisageable est alors de générer, pour chaque unité structurelle $u = u_1 u_2 \cdots u_n$, les séquences des k'-grams $\mathcal{F}'_k$ pour tout $k' = k2^i$ avec $k' \leq |u|$. Ceci nous permet de trouver plus rapidement des k'-grams de longueur importante sans compromettre la recherche de similarités plus petites. Un exemple d'une méta-lexémisation sur des k-grams puissances de 2 est présentée en figure 8.10 pour une instruction simple.

Indice	1-gram	2-gram	4-gram	8-gram
1	for	for lpar	for lpar id eq	for lpar id eq 0 ; id lt
2	lpar	lpar id	lpar id eq 0	lpar id eq 0 ; id lt MAX
3	id	id eq	id eq 0 ;	id eq 0 ; id lt MAX ;
4	eq	eq 0	eq 0 ; id	eq 0 ; id lt MAX ; id
5	0	0 ;	0 ; id lt	0 ; id lt MAX ; id ++
6	;	; id	; id lt MAX	; id lt MAX ; id ++ rpar
7	id	id lt	id lt MAX ;	
8	lt	lt MAX	lt MAX ; id	
9	MAX	MAX ;	MAX ; id ++	
10	;	; id	; id ++ rpar	
11	id	id ++		
12	++	++ rpar		
13	rpar			

FIG. 8.10 – Méta-lexémisation de l'instruction « for lpar id eq 0 ; id lt MAX ; id ++ rpar » avec k-grams de taille variable

8.4.2 Algorithme classique de tuilage glouton de chaîne de lexèmes

Objectif

L'algorithme Running Karp-Rabin Greedy String Tiling (RKR-GST) a été proposé par Wise [82, 83] en 1993 à l'origine pour la recherche de correspondances exactes sur des séquences biologiques. Il a ensuite été implanté originellement par l'outil JPlag [76] disponible par le biais d'un service web [93] pour la recherche de code dupliqué. Il permet la recherche de paires de clones exacts non-chevauchants sur une paire de séquence de lexèmes. Cela signifie ainsi que chaque exemplaire participant à une paire de clones :

- ne peut participer à une autre paire ;
- et ne peut posséder une intersection non-nulle avec un autre exemplaire participant à une autre paire de clones.

Nous notons que l'utilisation d'une méthode de recherche de facteurs répétés par l'utilisation d'une structure d'indexation de suffixes (cf chapitre 6) permet de trouver l'ensemble des clones exacts ; cependant ces clones peuvent présenter des zones de chevauchement qu'il est nécessaire ensuite d'éliminer si celles-ci ne sont pas désirées. D'autre part, les paires de clones intra-unité étant interdites, si un groupe de clones exacts de $m = m_1 + m_2$ exemplaires contient m_1 exemplaires extraits de l'unité 1 et m_2 exemplaires extraits de l'unité 2, seules $\min(m_1, m_2)$ paires de clone peuvent coexister, $\max(m_1, m_2) - \min(m_1, m_2)$ exemplaires restant orphelins.

Maximisation de la couverture Les paires de clones obtenues par l'utilisation du RKR-GST maximisent la couverture des séquences de lexèmes. La couverture des séquences de lexèmes u et v pour l'ensemble $\mathcal{R}$ des l paires de clones non-chevauchants avec $\mathcal{R} = \{(r_{11}, r_{12}), (r_{21}, r_{22}), \cdots, (r_{l1}, r_{l2})\}$ est quantifiée par le ratio $\mathcal{C}(\mathcal{R}) = \frac{|r_{11}|+|r_{21}|+\cdots+|r_{l1}|}{\min(|u|,|v|)}$ (on rappelle que $|r_{i1}| = |r_{i2}|$).

Algorithme

Aperçu général L'algorithme utilise une file de priorité pour la mémorisation des correspondances entre les chaînes de lexèmes u et v par longueur décroissante. Fondamentalement, l'algorithme se déroule en deux principales phases, elles-mêmes composées de passes itératives. Chaque itération d'une phase consiste à examiner la séquence des k-grams (pour k fixé) aussi bien pour la séquence de lexèmes u et la séquence v. Chaque k-grams de u est d'abord représenté par une empreinte basée sur l'utilisation d'une fonction de hachage incrémentale : les empreintes sont insérées dans une table de hachage $T_k(u)$. Une autre table $T_k(v)$ accueille les empreintes des k-grams v. L'objectif est ensuite de parcourir la séquence d'empreintes de u : pour chaque empreinte f de u correspondant à un k-gram non encore couvert par une correspondance définitive, nous cherchons dans la table de hachage $T_k(v)$ des empreintes de k-grams (également non-couverts) de v de valeur de hachage égale. Pour chaque couple de valeur de hachage identique trouvé, et après vérification de l'absence de collision par comparaison exhaustive des k-grams représentés, nous cherchons à étendre cette paire de k-grams identiques vers la droite. L'extension vers la gauche est elle inutile car déjà traitée avec l'empreinte du k-gram précédent de u. Après extension, nous obtenons une paire de facteurs de lexèmes de u et v identiques non-extensibles à droite. Cette paire est ajoutée dans la file de mémorisation des correspondances. À l'issue de chaque itération, les correspondances candidates sont extraites de la file de la plus longue à la plus courte : les lexèmes participant à celles-ci sont marqués afin de ne pas recouvrir d'autres correspondances. Ainsi, si une correspondance candidate est extraite de la file et est basée sur le facteur $x_1y_1x_2$ comportant des facteurs déjà couverts x_1 et x_2 alors seul le facteur non-couvert y_1 est ajouté à la file pour récupération ultérieure.

Première phase : détermination du plus long facteur partagé Lors de la première phase, nous démarrons d'une longueur initiale s pour les k-grams ($k = s$) recherchés. Les correspondances trouvées sont ajoutées dans la file de mémorisation. Toutefois, si une correspondance étendue comprend au moins $2k$ lexèmes, l'itération est arrêtée et les correspondances trouvées ignorées. En effet, la mise en évidence d'une longue correspondance peut témoigner de la présence d'autres correspondances longues qui pourraient bénéficier d'une valeur de k plus importante pour limiter les comparaisons lexème par lexème. Une nouvelle itération a donc lieu avec s fixé à la taille de la plus longue correspondance trouvée ($s \geq 2k$). Si au contraire aucune correspondance de longueur d'au moins $2k$ lexèmes n'est trouvée, l'itération est la dernière de la première phase : les correspondances trouvées de la file peuvent être marquées.

Seconde phase : récupération des correspondances Lors de la seconde phase, la longueur des k-grams recherchés est divisée par 2 à chaque itération successive. Il s'agit alors de trouver des correspondances de plus petites longueurs qui n'auraient pu être trouvées à une itération précédente. Finalement, l'algorithme se termine lorsque k est plus petit que le seuil de longueur minimale de correspondance fixé t. Nous notons que l'examen des correspondances candidates de la file n'est réalisé que lors de la dernière itération de la première phase (pour k maximal) et pour toutes les itérations de la seconde phase.

Exemple Considérons les chaînes de lexèmes $u = aabcdada$ et $v = dabcdadaa$. Lors de la première phase, nous recherchons le plus long facteur partagé. Pour la première itération, nous examinons les 1-grams. Nous constatons par exemple la correspondance $(u[1], v[8])$ sur le lexème a extensible à $(u[1..2], v[8..9])$ avec extension à aa. L'itération est arrêtée et nous

abordons la 2e itération avec $k = 2$. Une correspondance sur le facteur partagé *abcd* est trouvée (de longueur $4 \geq 2 * k$) : l'itération est alors terminée. Pour la 3e itération, $k = 4$: on relève les correspondances sur *abcd* et *dada* de longueur k sans autre facteur partagé de longueur plus importante. Ces correspondances sont placées en queue : seule l'une d'entre-elles est choisie, *abcd* par exemple, et marquée : la correspondance sur *dada* est tronquée en *ada*. Celle-ci est à son tour sélectionnée et marquée. Nous divisons par $2k$ pour la seconde phase : $k = 2$; aucune correspondance non marquée n'est trouvée. Pour $k = 1$, la correspondance non encore marquée $(u[1], v[9])$ pour le facteur a est reportée.

À propos de la maximalité de la couverture

Les paires de clones obtenues par l'utilisation du RKR-GST maximisent la couverture des séquences de lexèmes à condition de ne pas imposer de longueur minimale de correspondance.

Si chaque lexème $a \in \Sigma$ possède α occurrences dans u et β occurrences dans v, $\min(\alpha, \beta)$ occurrences exactement sont marquées comme participant à des correspondances : si plus d'occurrences étaient marquées, celles-ci appartiendraient à plusieurs correspondances qui se chevaucheraient donc ; si moins d'occurrences étaient marquées, il serait possible de créer de nouvelles correspondances de taille au moins unitaire. D'autre part, nous notons que, de part sa construction, l'algorithme accorde une priorité plus haute à la sélection des correspondances les plus longues. Si une condition de longueur minimale est imposée, il est possible que le ratio de couverture $\mathcal{C}(\mathcal{R})$ ne soit pas maximal. En effet, fixons un seuil minimal de longueur de correspondance à t et prenons pour exemple le facteur de u xay avec $x \in \Sigma^{t-1}$, $a \in \Sigma$ et $y \in \Sigma^{\gamma}$ ($\gamma >> t$). xay est un facteur également présent dans v. La priorité donnée aux plus longues correspondances permet le report du clone portant sur le facteur ay, x ne pouvant être reporté car de longueur inférieure au seuil. Si nous avions sélectionné les correspondances portant sur xa et y, la couverture aurait été supérieure.

La maximalité de la couverture est assurée (pour $t = 1$) par l'extension systématique sur la droite des correspondances trouvées à partir d'un germe de k lexèmes. Par ailleurs, la sélection prioritaire de la correspondance la plus longue dans la file garantit sa non extensibilité sur la gauche.

Complexité de l'algorithme original

L'algorithme RKR-GST original nécessite deux phases d'itérations : la première avec longueur k de k-grams croissante et la deuxième avec des valeurs décroissantes divisées par 2 à chaque itération. Le pire cas de complexité temporelle est obtenue par exemple pour $v = a^n$ et $u = a^{2l}$ (avec $2l < n$). Il existe ainsi $n - l' + 1$ correspondances possibles contenant le facteur $a^{l'}$ ($l' \leq l$) selon la position de départ choisie dans v. Si nous considérons l'itération cherchant les k-grams avec $k = l$, nous recherchons les l-grams identiques dans v (extensibles jusqu'à $2l$ lexèmes) pour $u[i]$ avec $i \leq l$: il y en a $n - l + 1$. Ainsi, pour chaque position $i \in [1..l]$, il est nécessaire de traiter entre $n - l + 1$ empreintes de v et étendre les correspondances de l lexèmes au plus, soit un coût temporel global en $O(nl^2)$, et ce pour pour une unique itération.

8.4.3 Pistes d'amélioration du tuilage glouton

Utiliser une seule phase d'itérations sur k-grams décroissants

L'algorithme original procède à la recherche et l'extension de k-grams en deux phases : les itérations de ces deux phases peuvent s'avérer redondantes. Il serait donc préférable de n'utiliser qu'une unique phase où chaque itération procéderait à la recherche et l'extension de k-grams par longueur k-décroissante. Il reste alors à déterminer la longueur du plus long facteur commun entre u et v pour la valeur de démarrage[4] de k.

Accélérer l'extension

L'extension de k-grams correspondants entre u et v est temporellement réalisée dans le pire des cas en $O(\min(|u|, |v|))$. Lorsqu'un k-gram est mis en correspondance, l'extension pourrait être réalisée plus rapidement en utilisant des tables de hachage de k'-grams avec $k' < k$ plutôt que de simples 1-grams. À cet effet, nous recherchons des k-grams identiques avec $k = 2^i$. Tous les k-grams avec $k = \{2^1, 2^2, \cdots, 2^{\lfloor \log_2(\min(m,n)) \rfloor}\}$ sont hachés et stockés dans des tables de hachage. Lorsqu'une paire de k-grams identiques $(u[i|k], v[j|k])$ est trouvée, l'extension est entreprise : on vérifie alors l'égalité des $\frac{k}{2}$-grams (par leur valeur de hachage) $u[i+k|k/2]$ et $v[j+k|k/2]$. S'il y a égalité, l'extension est réalisée, sinon elle ne l'est pas. On recherche ensuite récursivement s'il est possible d'étendre la correspondance précédemment obtenue de $\frac{k}{2^2}, \cdots, \frac{k}{2^{\log_2 k}} = 1$ lexèmes. Ainsi dans tous les cas, l'extension est réalisée en au plus $\log_2(\min(m,n))$ itérations. Toutefois, cela nécessite le calcul et le stockage de $\log_2(\min(m,n))(m+n)$ empreintes indexées par valeur de hachage mais aussi par position de début.

Éviter des extensions redondantes sur des k-grams consécutifs

Dans certains cas, le procédé d'extension est redondant. En effet, si le k-gram $u[i|k]$ est identique au k-gram $u[j|k]$ et que le procédé d'extension augmente la taille de la correspondance de $p \geq 1$ lexèmes pour obtenir $(u[i|k+p], v[j|k+p])$, lorsque nous rechercherons les k-grams de v identiques à $u[i+1|k], \cdots, u[i+p|k]$, nous trouverons parmi eux nécessairement $v[j+1|k], \cdots, v[j+p|k]$. Ainsi, pour les k-grams identiques $u[i+q|k]$ et $v[j+q|k]$ avec $1 \leq q \leq p$, il est inutile de réentreprendre le processus d'extension : nous obtenons directement la correspondance $(u[i+q|(p-q)+k], v[j+q|(p-q)+k])$. Cette correspondance est inintéressante et peut être ignorée car extensible à gauche en la correspondance $(u[i|k+p], v[j|k+p])$. Pour le cas présenté en 8.4.2, ceci nous permet de réaliser l'étape d'extension uniquement pour le premier k-gram : la complexité temporelle est alors réduite à $O(nl)$.

8.4.4 Algorithme amélioré de tuilage glouton

Calcul des tables de hachage

De ces observations, nous proposons une variante au RKR-GST permettant la recherche de correspondances permettant une couverture maximale avec une complexité d'itération dans

[4] Ou à défaut, il serait possible de fixer k à la longueur de la plus petite séquence comparée, quitte à ne trouver aucune correspondance pour les premières itérations.

le pire cas quadratique en la longueur des chaînes. Elle utilise $\lfloor \log_2(z) \rfloor$ itérations où z est la longueur du plus grand facteur répété entre les deux chaînes comparées u et v et se décompose en deux étapes principales. La première consiste à calculer les tables de hachage de k-grams $T_k(u)$ et $T_k(v)$ pour $k \in \left\{2^1, 2^2, \cdots, 2^{\lfloor \log_2(z) \rfloor}\right\}$: l'objectif est alors de trouver les facteurs de u et v communs qui soient de longueur puissance de 2. $\lfloor \log_2(z) \rfloor + 1$ est la première itération pour laquelle nous ne constatons aucun k-gram commun entre u et v. Les empreintes des tables sont indexées à la fois par leur valeur de hachage pour retrouver les k-grams communs mais aussi par leur position pour permettre l'extension rapide. La table $T_k(u)$ est calculée avec $k + (|u| - k)$ opérations avec une fonction de hachage incrémentale pouvant être réduites à $1+(|u|-k)$ opérations en utilisant les deux premières valeurs de $T_{k/2}(u)$ pour l'initialisation de la première valeur de hachage de $u[1..k]$. Le calcul des tables est donc de complexité temporelle et mémorielle $\log_2(z)z$. Nous notons que la taille des tables peut être réduite en éliminant les empreintes de u ne partageant leur valeur de hachage avec aucune des empreintes de v (et réciproquement).

Itérations

À chaque itération i, nous recherchons les k-grams communs entre u et v avec $k = 2^{\lfloor \log_2 z \rfloor - i + 1}$. Nous examinons ainsi chaque k-gram de u de $u[1..k]$ à $u[m-k+1..m]$. Pour chacun de ces k-grams, nous recherchons les k-grams identiques sur v grâce à $T_k(v)$. Pour chaque paire de k-grams $(u[i..i+k-1], v[j..j+k-1])$, nous appliquons la méthode d'extension logarithmique décrite précédemment afin d'étendre le k-grams d'au plus $\frac{k}{2} + \frac{k}{2^2} + \cdots + (\frac{k}{2^{\log_2 k}} = 1) = k - 1$ lexèmes. Nous notons que si la paire de k-grams avait été extensible à droite sur k lexèmes, elle aurait été mise en évidence à l'itération précédente lorsque les $2k$-grams similaires avaient été recherchés. Pour chaque k-gram de u, nous conservons toutes les paires de k-grams étendues de u et v de longueur maximale. Par ailleurs, nous maintenons dans un tableau E indicé par leur position de fin toutes les paires de correspondances trouvées : ainsi lorsque nous examinons le k-gram de u de position i, $c = E[i + (k-1) + 1] \cup E[i + (k-1) + 2] \cup \cdots$ contient toutes les correspondances dont le membre de u se termine strictement au-delà de la position $i + (k-1)$ et qui englobe donc nécessairement le k-gram $u[i..i + (k-1)]$. Nous pouvons donc ignorer immédiatement tous les k-grams de v contenus dans c, ceux-ci ayant déjà conduit à l'obtention d'une correspondance plus étendue sur la gauche. Finalement, nous obtenons au plus $(m - k + 1)(n - k + 1)$ correspondances candidates de longueurs comprises entre k et $2k-1$: ces correspondances sont ajoutées dans une file de priorité permettant la sélection de la correspondance la plus longue à la plus courte sans chevauchement. Deux arbres d'intervalles sont utilisés afin de marquer les intervalles de u et de v déjà utilisés. Si une correspondance trouvée chevauche au moins une autre, celle-ci est découpée en parties non-chevauchantes qui sont réinsérées dans la file si celles-ci ont une longueur supérieure ou égale à k : dans le cas contraire, celles-ci seront gérées par une future itération. Afin de limiter le nombre de comparaisons d'empreintes d'itérations futures liées à l'extension, nous supprimons des tables de hachage toutes les empreintes participant totalement à des correspondances définitives : pour une correspondance de longueur k, cela concerne $O(k \log(k))$ empreintes.

Complexité

Comme vu en 8.4.3, dans le pire des cas, pour une itération, la complexité temporelle est quadratique en nombre de lexèmes manipulés. Mémoriellement, la file de correspondances

candidates peut également contenir un nombre quadratique de correspondances. Si nous nous basons sur un cas médian où u et v (avec $|u| = |v| = n$) comportent f facteurs maximaux communs de longueur maximale l de répartition équilibrée nécessitant $O(\log l)$ itérations, avec un taux de duplication sur v de ρ, la recherche des k-grams communs et l'extension nécessite un temps en $O(f\rho \log l)$ tandis que le hachage nécessite $O(n \log l)$ opérations d'où une complexité temporelle en $O((f\rho + n) \log l)$. Il reste à évaluer la complexité expérimentale sur des paires de séquences de lexèmes issues de projets réels.

8.4.5 Adaptation à la recherche de similitudes sur une unique chaîne

Lors de la recherche de k-grams communs, si u et v sont une unique chaîne sur laquelle des correspondances non-chevauchantes sont recherchées, nous veillons à sélectionner des paires d'occurrences de k-grams distincts non chevauchants mais similaires ainsi qu'à limiter l'extension des k-grams pour éviter des cas de chevauchement.

8.4.6 Adaptation à la recherche sur une base de projets indexés

L'algorithme proposé est facilement généralisable pour la recherche de correspondances non-chevauchantes entre une séquence de lexèmes requête u et une base de séquences de lexèmes V représentant plusieurs projets avec une indexation par valeur de hachage ainsi que par position de k-gram. Les tables $T_k(V)$ sont mises à jour lors de l'indexation de chaque projet. Le principal inconvénient réside cependant dans la taille de la base gérée nécessitant au minimum $2\log_2(h)(b + \log_2(|V|))$ bits par lexème pour des valeurs de hachage de b bits et une longueur maximale de séquence de lexèmes de h. Ceci est à rapprocher des $5\log_2(|V|)$ bits par lexèmes nécessaires pour l'indexation de ces chaînes par une structure d'arbre de suffixes optimisée (cf 6.6.1).

Réduction d'espace par sélection de méta-lexèmes

Certaines pistes peuvent être explorées afin de limiter l'espace de la base d'indexation. Si l'on peut se permettre d'ignorer des similarités portant sur un nombre de lexèmes inférieur à un seuil, une idée serait d'utiliser une méthode de sélection des empreintes sur fenêtre locale. Le déroulement de l'algorithme serait inchangé à la différence que l'unité élémentaire de correspondance serait l'empreinte sélectionnée de k-gram et non plus le lexème unitaire : les tables $T_l(V)$ contiendraient les valeurs de hachage des l-grams d'empreintes de k-grams.

Indexation paresseuse

Une deuxième piste concernerait l'indexation paresseuse des méta-lexèmes de taille variable. Dans un premier temps, l'ensemble des unités sont indexées en 1-grams, 2-grams, $2^{l_{\min}}$-grams. $l_{\min}$ reste volontairement petit afin de ne pas alourdir la taille de la base mais est suffisamment grand afin que la cardinalité moyenne de chaque groupe de $l_{\min}$-gram reste faible. Nous associons à chaque unité indexée u_i la taille maximale des k-grams indexée, $l(u_i)$ avec $l(u_i) \geq l_{min}$. Lorsqu'une unité requête q est comparée par rapport à la base, les k-grams pour $k = \{2^0, 2^1, \cdots, 2^{l_{min}}\}$ sont calculés. La table des k-grams (avec $k = 2^{l_{min}}$) de la base est interrogée pour les unités u_i avec $l(u_i) = l_{min}$. Pour toute unité u_i possédant au moins deux

k-grams k-consécutifs[5] (ou du moins leurs valeurs de hachage) concordant avec deux k-grams k-consécutifs de q, nous décidons de calculer les k-grams avec $k = 2^{l(u_i)+1}$ de u_i, ces k-grams étant ajoutés à la base. Nous incrémentons alors la valeur de $l(u_i)$. Le procédé est itérativement répété pour $l = \{l_{\min}, l_{\min+1}, \cdots, \}$ jusqu'à trouver une valeur de l pour laquelle il n'existe aucune unité u_i de la base telle que $l = l(u_i)$ et qu'il existe deux 2^l-grams 2^l-consécutifs de q concordant avec deux 2^l-grams 2^l-consécutifs de u_i. Si $2^l > |u_i|$, le procédé est également terminé. On rappelle que les valeurs de hachage des 2^l-grams peuvent facilement être calculées d'après les deux 2^{l-1}-grams 2^{l-1}-consécutifs les constituant si la fonction de hachage est fortement incrémentale. Après cette opération, s'il existe un k-gram de q identique à un k-gram d'une unité u_i de la base (pour k puissance de 2), nous avons la garantie que celui-ci est indexé par une valeur de hachage dans la base. L'algorithme de RKR-GST amélioré peut donc ensuite être appliqué. On pourra remarquer que le processus d'indexation paresseuse peut en fait être assimilé à la première phase de l'algorithme original de RKR-GST.

À propos de la condition de non-chevauchement des correspondances

Nous pouvons penser que la problématique de recherche de correspondances non-chevauchantes n'est pas nécessairement adaptée à la situation de recherche de correspondances d'une unité requête sur une base dans la mesure où une correspondance d'un projet u par rapport à un certain projet peut masquer, afin de respecter la condition de non-chevauchement sur u, des correspondances, certes plus courtes mais potentiellement plus intéressantes sur d'autres projets. Ainsi si le projet u possède une copie conforme u' indexé dans la base ainsi que d'autres projets u'', u''' quasi-identiques, seule une correspondance sera reportée entre u et u', la similarité entre u' et u'' n'étant pas mise en valeur. Il peut donc être intéressant de proposer une condition de non-chevauchement plus large. Ainsi chaque exemplaire de q (unité requête) d'une paire de clone relevée comme correspondance pourrait participer à une autre correspondance ssi elle concerne une unité différente de la base.

[5] Deux k-grams sont k-consécutifs si la différence de leur position est de k. Ils peuvent être regroupés en un seul $2k$-gram.

Troisième partie

Recherche de correspondances sur des arbres syntaxiques

L'utilisation de chaînes de lexèmes pour la recherche de similarité peut se réveler peu flexible. Outre la mauvaise délimitation de frontières des correspondances pouvant intersecter plusieurs unités syntaxiques, peu d'opérations de normalisation peuvent être réalisées. Certains types de lexèmes peuvent être ignorés voire confondus mais pas des régions entières de code. L'emploi d'arbres de syntaxe pallie à ces inconvénients : des nœuds ou sous-arbres complets peuvent être abstraits. Les similarités sur les sous-arbres de syntaxe peuvent être aisément mises en évidence par des méthodes de hachage exact que nous explorons au chapitre 9. L'utilisation de profils d'abstraction variés permet de personnaliser le niveau de granularité des similitudes trouvées. Mais tout comme des k-grams identiques de chaînes de lexèmes, des sous-arbres similaires individuels ne sont pas exploitables à l'état brut, particulièrement s'il s'agit de petits sous-arbres. Il s'agit donc d'agglutiner ces sous-arbres germes proches dans leur arbre de syntaxe hôte afin de former des correspondances de volume plus important permettant ainsi de trouver des macro-similarités malgré des opérations d'édition se caractérisant par la suppression de sous-arbres ou de nœuds ou le déport de code dans une nouvelle fonction (factorisation) ou au contraire le développement d'un appel de fonction. On adjoindra à cet effet aux arbres de syntaxe d'un projet leur graphe d'appel. Un heuristique de consolidation de correspondances sur arbres de syntaxe est proposée au chapitre 11. Nous discuterons également auparavant d'une méthode d'indexation de sous-arbres d'un arbre de syntaxe en utilisant plusieurs profils d'abstraction dans le chapitre 10.

If I had eight hours to chop down a tree, I'd spend six hours sharpening my ax.

Abraham Lincoln

9

Fonctions de hachage sur l'espace des sous-arbres

Sommaire

La recherche de similarité sur du code source représenté sous la forme d'arbres de syntaxe nécessite des méthodes efficaces et de complexité praticable pour mettre en évidence des sous-arbres similaires. La comparaison exhaustive de toutes les paires de sous-arbres, soit n^2

paires pour deux projets représentés par des arbres de n nœuds, n'est pas envisageable en pratique, d'autant plus si au-delà de la détermination d'une similarité exacte, des techniques d'alignement d'arbres sont mises en oeuvre. Il devient alors indispensable de réduire le nombre de comparaisons coûteuses d'arbres en utilisant des métriques ou valeurs de hachage afin de regrouper des sous-arbres de similarité supposée.

Dans ce chapitre, nous évoquons quelques métriques utilisables sur les sous-arbres ainsi que certaines méthodes de hachage. Nous présentons également quelques techniques d'abstraction d'arbres de syntaxe permettant ainsi l'obtention de valeurs de hachage communes pour des sous-arbres présentant la même abstraction.

9.1 Quelques méthodes de hachage de sous-arbres

9.1.1 Métriques

Les métriques sur des unités structurelles du code source sont des mesures permettant de quantifier certaines propriétés du code. Elles s'expriment le plus souvent dans un espace vectoriel réel. Nous passons maintenant en revue quelques unes des métriques les plus couramment utilisées.

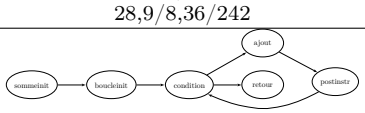
Métriques de quantification de la complexité du code et du style de programmation

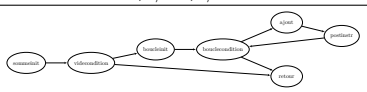
Mesurer la complexité du code source a toujours été une préoccupation majeure en ingénierie logicielle. Ces métriques permettraient d'estimer la complexité du code et donc son coût de maintenabilité. Nous présentons ici les métriques de Halstead et la métrique de complexité cyclomatique pour finir sur une métrique basée sur le vecteur de comptage des types de nœuds. La figure 9.1 présente deux versions d'un court code de calcul de moyenne sur lesquels les métriques présentées ont été appliquées.

Métriques de Halstead

Les métriques de Halstead [108] sont calculées à partir du comptage des opérateurs et opérandes totaux et distincts présents dans l'unité structurelle analysée : le vecteur $V = (\alpha, \bar{\alpha}, \beta, \bar{\beta})$ de l'unité comportant α opérateurs distincts, $\bar{\alpha}$ opérateurs totaux, β opérandes distincts et $\bar{\beta}$ opérandes totaux est calculé. Des mesures de longueur $(\bar{\alpha} + \bar{\beta})$, de vocabulaire $(\alpha + \beta)$, de volume $((\bar{\alpha} + \bar{\beta}) \log_2(\alpha + \beta))$, de difficulté $(\frac{\alpha\bar{\beta}}{2\beta})$ et d'effort (produit du volume et de la difficulté) sont dérivées de ce vecteur. Les opérateurs sont alors définis comme l'ensemble des opérateurs arithmétiques, des appels de fonction et des opérations d'affectation. Les opérandes sont quant à eux les variables locales, membres de structures et constantes.

Il est aisé de constater que l'ajout ou la suppression de code inutile ainsi que la réécriture d'expressions peut avoir un impact important sur le vecteur V. D'autre part, il existe des risques importants que plusieurs unités structurelles de vecteurs accidentellement proches n'abritent aucune similarité réelle.

<table>
<tr><th></th><th>Code original</th><th>Code modifié</th></tr>
<tr><td></td><td><pre>
double moyenne(double[] tab)
{
double somme = 0.0;
for (int k = 0; k < tab.length; k++)
 somme += tab[k];
return somme / (double)tab.length;
}
</pre></td><td><pre>
double moyenne(float[] tab)
{
 float somme = 0.0f;
 if (tab.length == 0) return 0.0f;
 else
  for (int k = tab.length; k >= 0; k−−)
   somme += tab[k];
 return somme / (double)tab.length;
}
</pre></td></tr>
<tr><td>Opérateurs
Opérandes
Longueur/vocabulaire
Volume/difficulté/effort</td><td>11 (9 distincts)
13 (7 distincts)
24/16
28,9/8,36/242</td><td>14 (10 distincts)
16 (7 distincts)
30/17
36,9/11,4/421</td></tr>
<tr><td>Flux de contrôle
Métrique cyclomatique</td><td> 

$E - N + 2P = 6 - 6 + 2 = 2$</td><td> 

$E - N + 2P = 8 - 7 + 2 = 3$</td></tr>
</table>

FIG. 9.1 – Une fonction et sa version obfusquée avec le calcul de leurs métriques de Halstead et cyclomatique

Nous pouvons noter que les métriques de Halstead ont servi de base à la réalisation du (très probable) premier outil de recherche de similarité sur du code source (en langage Fortran) proposé par Ottenstein [75] en 1976.

Complexité cyclomatique de McCabe

McCabe a proposé une métrique [110] sur le graphe de contrôle de flux d'une unité structurelle. Un tel graphe représente l'ensemble des instructions d'un programme (nœuds), deux instructions étant liées par une arête s'il existe un chemin d'exécution où ces deux instructions sont exécutées séquentiellement. La complexité cyclomatique C est définie par $C = E - N + 2P$ où E est le nombre d'arêtes, N le nombre de nœuds et P le nombre de composantes connexes du graphe. Cette valeur reflète le nombre d'expressions conditionnelles issues de structures de contrôles présentes dans le code.

Si la complexité cyclomatique peut être difficilement réduite, une augmentation artificielle par l'ajout de structures de contrôle inutiles (cf 4.6) est possible. Au-delà d'une analyse statique, une détermination de la complexité cyclomatique dynamique par le suivi des chemins d'exécution serait plus fiable pour localiser les conditionnelles suspectées invariantes. D'autre part, cette métrique dispose d'un pouvoir discriminateur réduit : l'espace de valeurs de complexité cyclomatique est trop faible afin de pouvoir disposer de groupes de sous-arbres d'effectifs suffisamment petits. Il est en effet rare de trouver des fonctions dont la complexité cyclomatique est supérieure à 20.

Vecteur de comptage de nœuds ou de sous-arbres

Étant donnée une unité structurelle représentée par un arbre de syntaxe, il est possible d'exprimer le nombre de chaque type de nœuds présents au sein de cet arbre : nous pouvons ainsi en extraire un vecteur de comptage de nœuds. La taille de ce vecteur, représentant le

nombre de types de nœud différents peut être réduite par une opération d'abstraction des types : nous traiterons de ce procédé en 9.2. Ce vecteur de comptage peut être généralisé au comptage des différents sous-arbres de taille ou de hauteur bornée. Cette technique est notamment employée par l'outil de recherche de similitudes Deckard [70] qui génère pour chaque sous-arbre à hacher un vecteur caractéristique comptant les petits sous-arbres inclus. Les différents sous-arbres vectorisés sont ensuite regroupés en utilisant une technique de hachage localement sensible (Locality Sensitive Hashing, LSH) [52]. Cette technique permet de regrouper des vecteurs de distance faible et donc des sous-arbres suffisamment voisins dans l'espace vectoriel des effectifs de sous-arbres caractéristiques.

Tout comme la cardinalité du nombre de k-grams pour k suffisamment faible est assez réduite par rapport à la cardinalité théorique des $|\Sigma|^k$ k-grams, il existe également un nombre limité de petits arbres, qu'ils soient définis par une taille k ou une hauteur h limite. Il est toutefois nécessaire de maintenir la taille ou la hauteur limite suffisamment basse pour restreindre la taille des vecteurs à manipuler.

L'utilisation d'un vecteur de comptage est totalement insensible aux opérations de transposition de code. Ce comportement ensembliste peut également induire des cas de structures non-similaires ayant un vecteur de comptage identique ou proche selon une fonction de hachage localement sensible. D'autre part, des modifications localisées comme la réécriture d'expressions peuvent modifier le type des petits sous-arbres considérés pour le comptage et donc le vecteur généré. L'ajout ou la suppression de code inutile peut également avoir une influence sur le vecteur.

Le vecteur de comptage peut également être projeté sur une unique valeur entière en sommant ses valeurs coefficientées par un représentant entier aléatoire de chacun des types (cf 9.1.2). Cette méthode se rapproche alors d'un hachage exact dans la mesure où les valeurs ne peuvent être comparées autrement que par une égalité stricte.

9.1.2 Hachage exact

Quelques propriétés du hachage exact

Les métriques sur les unités structurelles présentent la caractéristique d'être à valeurs sur un espace limité avec une sensibilité réduite en rapport avec certaines opérations d'édition du code. Ce qui peut être un avantage dans certaines situations d'obfuscation telle que la transposition de code, peut également générer des collisions entre structures manifestement différentes mais de métrique proche. D'autre part l'apparition de faux négatifs est inévitable pour certaines opérations d'édition agissant directement sur les propriétés du code utilisées pour le calcul de la métrique. *A contrario*, l'utilisation de méthodes de hachage exact ne permet de regrouper que des sous-arbres exactement similaires — moyennant l'abstraction choisie sur les types de nœuds — avec une probabilité de collision maîtrisable par le choix de la cardinalité de l'espace de hachage.

La valeur de hachage exact d'un sous-arbre ne possède aucune signification sémantique particulière : les valeurs de hachage de deux sous-arbres ne permettent pas de les comparer sur aucun critère. Dans ce sens, seule l'égalité de deux valeurs de hachage possède une signification :

si la fonction de hachage est parfaite et à valeurs sur un espace de cardinalité N, l'égalité signifie que les deux sous-arbres ont une probabilité $1 - \frac{1}{N}$ d'être identiques (moyennant l'abstraction usitée). Cette remarque conditionne le choix de la cardinalité de l'espace de hachage pour la mise en place d'une base d'empreintes : un compromis doit être alors trouvé sur la taille de la valeur pour minimiser la taille de la base tout en réduisant la probabilité de collision $\frac{1}{N}$. Nous introduirons dans le chapitre 10 un processus d'indexation utilisant des tailles de valeur de hachage adaptatives en fonction des collisions rencontrées.

Nous proposons ici deux méthodes récursives de hachage exact sur les arbres. Connaissant la valeur de hachage des sous-arbres enfants $C_1, C_2, \cdots, C_k$ d'un arbre A, nous pouvons obtenir la valeur de hachage de A en temps global $\Theta(k)$. Il en découle la possibilité de paralléliser le calcul de la valeur de hachage de A par l'emploi de fils d'exécution indépendants pour les calculs des valeurs de hachage de sous-arbres enfants.

On notera que pour des petits sous-arbres dont la cardinalité est réduite, l'usage d'une méthode de hachage peut être remplacé par l'énumération exhaustive de ceux-ci. Pour des sous-arbres de taille intermédiaire, une fonction de hachage exacte peut être utilisée mais nous pouvons prendre en compte la cardinalité réduite de l'espace de ces sous-arbres afin de réduire la taille de la valeur. La probabilité de collision théorique est de $\frac{1}{N}$ pour une fonction de hachage parfaite sur un espace de sous-arbres de cardinalité infinie. En revanche, sur un espace de sous-arbres de cardinalité C, la probabilité de collision s'élève à $p = \frac{(C \mod N)\cdot\alpha^+ + (2^k - C \mod N)\alpha^-}{C^2}$ avec $\alpha^+ = \lceil \frac{C}{N} \rceil$ et $\alpha^- = \lfloor \frac{C}{N} \rfloor$. Nous réalisons quelques tests présentés en section 9.1.3 avec certaines fonctions de hachage afin de quantifier les collisions (faux positifs) survenant lors du traitement d'arbres de différentes tailles.

Vecteur de valeurs de type

Préalablement à une opération de hachage exact d'une séquence de lexèmes ou d'un arbre, il est nécessaire de représenter chaque type de lexème ou de nœud par une valeur entière. L'ensemble des types manipulés est ainsi représenté par un vecteur de valeurs entières. Afin de limiter les possibilités de collision entre valeurs de hachage, le vecteur de valeurs devrait être généré aléatoirement. D'autre part, ce vecteur devrait être recalculable facilement avec la seule donnée des types, exprimés par exemple sous la forme de chaînes de caractères ou alors d'un identificateur entier séquentiel (qui peut être obtenu par tri lexicographique des chaînes). Nous pouvons ainsi utiliser un générateur pseudo-aléatoire linéaire congruentiel afin d'obtenir séquentiellement des valeurs pour les types d'identifiants séquentiels $1, 2, \cdots, n$.

Hachage de Karp-Rabin sur les arbres

Mots de Dyck Il est possible de représenter chaque arbre A par une séquence de lexèmes unique issue de son parcours en profondeur : il s'agit du mot de Dyck $\mathcal{DW}(A)$ de cet arbre. Nous pouvons choisir la convention suivante pour représenter l'arbre A dont la racine est de type a et ayant pour enfants les sous-arbres $C_1, C_2, \ldots, C_k$:

$$\mathcal{DW}(A) = (\rightarrow a) \cdot \mathcal{DW}(C_1) \cdot \mathcal{DW}(C_2) \cdots \mathcal{DW}(C_k) \cdot (a \leftarrow)$$

Si les arbres de syntaxe utilisent un alphabet Σ de types non terminaux ainsi qu'un alphabet Θ de types terminaux, le mot de Dyck est exprimé sur l'alphabet $(\rightarrow \Sigma) \cup (\Sigma \leftarrow) \cup \Theta$ afin

d'exprimer l'ensemble des lexèmes ouvrants et fermants pour chaque type de nœud non terminal ainsi que les lexèmes terminaux (d'arité nulle). Si chaque type de nœud possède une arité constante, l'utilisation de lexèmes fermants est inutile car l'interprétation du mot de Dyck n'est pas ambiguë.

Hachage de Karp-Rabin sur les mots de Dyck Les fonctions de hachage classiques manipulent des chaînes de lexèmes : il est possible de les utiliser pour calculer une valeur de hachage d'un arbre par l'intermédiaire de son mot de Dyck. Nous nous intéressons en particulier aux fonctions de hachage polynomiales, dites de Karp-Rabin, offrant des propriétés d'incrémentalité standard et d'incrémentalité forte (cf 8.1.2). Nous pouvons ainsi calculer, en temps linéaire de son nombre de nœuds et donc de la longueur de son mot de Dyck, la valeur de hachage d'un arbre. Lorsqu'à la fois la valeur de hachage de l'arbre et de tous ses sous-arbres sont nécessaires, comme c'est le cas pour les applications de recherche de sous-arbres similaires, le calcul peut être mené en temps $O(n \log n)$ pour un arbre de n nœuds. Nous présentons en figure 9.2 un exemple de calcul des valeurs de hachage de tous les sous-arbres de l'arbre de syntaxe déjà figuré 3.5.

Hachage cryptographique adapté aux arbres

Des fonctions de hachage cryptographique peuvent être utilisées afin de générer une valeur de hachage pour un arbre de syntaxe. Une fonction de hachage est dite de qualité cryptographique si, outre la minimisation du risque de collision accidentelle, elle permet de limiter les attaques portant sur la fabrication *a posteriori* de données correspondant à une valeur de hachage donnée ou la création simultanée d'un couple de données différentes de même valeur de hachage. Cette dernière propriété n'est cependant pas cruciale pour des applications de recherche de sous-arbres similaires. D'autre part, les fonctions de hachage cryptographiques possèdent une propriété d'effet d'avalanche stricte [57] : la modification de n'importe quel lexème ou type de nœud a pour conséquence une probabilité de $\frac{1}{2}$ d'inversion de chaque bit de l'empreinte. Ce n'est pas le cas d'une fonction de hachage polynomiale où la modification d'un des entiers d'entrée aura un effet moins important si celui-ci est proche de la fin de la séquence d'entrée. Toutefois l'usage de vecteurs de valeur de type pseudo-aléatoires permet de remédier à ce problème. La propriété d'avalanche stricte permet de réduire la taille d'une valeur de hachage de k bits à k' bits ($k' < k$) en sélectionnant k' bits quelconques : pour la suite, nous choisissons de garder les k' bits de poids faible (soit le modulo $2^{k'}$).

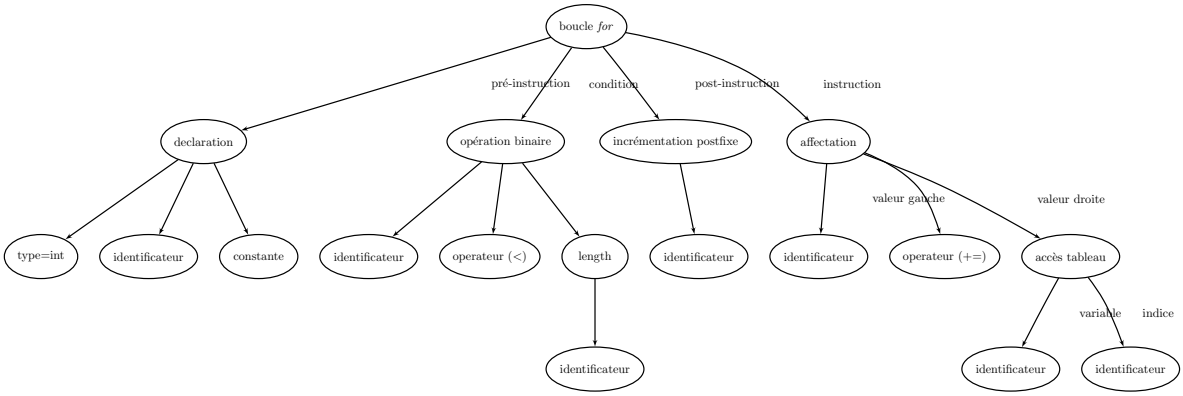
En règle générale, les fonctions de hachage cryptographique ne permettent pas de calculer rapidement la valeur de hachage de la concaténation de deux chaînes connaissant les valeurs individuelles liées à chacune des chaînes : elles ne satisfont pas les propriétés d'incrémentalité simple et forte. Elles ne sont donc pas adaptées au hachage de k-grams.

Nous proposons d'utiliser des fonctions de hachage cryptographique pour le calcul de valeur de hachage d'un arbre et de tous ses sous-arbres. À cet effet, nous calculons récursivement du bas de l'arbre vers le haut de l'arbre la valeur de hachage des sous-arbres. Si A est l'arbre traité de nœud racine a, $C_1, C_2, \cdots, C_k$ ses sous-arbres enfants et $\mathcal{C}$ la fonction de hachage cryptographique utilisée, nous calculons ainsi la valeur de hachage de A :

$$h(A) = \mathcal{C}(a \cdot h(C_1) \cdot h(C_2) \cdots h(C_k))$$

```
for (int i=0 ; i < tab.length ; i++) somme += tab[i] ;
```

(a) Code source



(b) Arbre de syntaxe abstrait (cf 3.5)

$$\begin{aligned}\mathcal{DW}(A_{\text{for}}) = {} & (\rightarrow \text{for }) \\ & (\rightarrow \text{declaration }) \,\text{type identificateur constante } (\text{declaration } \leftarrow) \\ & (\rightarrow \text{opbin }) \cdots (\text{opbin } \leftarrow) \\ & (\rightarrow \text{incr }) \cdots (\text{incr } \leftarrow) \\ & (\rightarrow \text{affectation }) \cdots (\text{affectation } \leftarrow) \\ & (\text{for } \leftarrow)\end{aligned}$$

(c) Mot de Dyck

$$h(A_{\text{for}}) = (((v((\rightarrow \text{for })) \times B + v((\rightarrow \text{declaration }))) + v(\text{type })) \times B + \cdots) + v((\text{for } \leftarrow))$$

(d) Hachage séquentiel polynomial du mot de Dyck

$$\begin{aligned}h(A_{\text{for}}) = {} & \Big(\Big(h(A_{\text{declaration}}) \times B^{|\mathcal{DW}(A_{\text{opbin}})|} + h(A_{\text{opbin}})\Big) * B^{|\mathcal{DW}(A_{\text{incr}})|} + h(A_{\text{incr}})\Big) \\ & * B^{|\mathcal{DW}(A_{\text{affectation}})|} + h(A_{\text{affectation}})\end{aligned}$$

(e) Hachage récursif polynomial des sous-arbres

$$h(A_{\text{for}}) = \mathcal{C}\left((\rightarrow \text{for }) \cdot h(A_{\text{declaration}}) \cdot h(A_{\text{opbin}}) \cdot h(A_{\text{incrémentation}}) \cdot h(A_{\text{affectation}})\right)$$

(f) Hachage récursif cryptographique des sous-arbres

FIG. 9.2 – Calcul des valeurs de hachage de Karp-Rabin pour les sous-arbres d'un arbre

Pour des sous-arbres réduits à une feuille, la valeur de hachage $h(a)$ se confond avec celle de la valeur entière de a. On notera que l'on ne conservera qu'une partie des bits de la valeur de hachage pour le stockage des empreintes en base.

Parmi les fonctions de hachage cryptographiques testées, nous pouvons citer MD5 [55] et SHA-1 [51] qui sont obsolètes pour des applications cryptographiques pures mais s'avèrent assez rapides et peu sujettes à des collisions pour le hachage d'arbre de syntaxe.

9.1.3 Évaluation expérimentales de fonctions de hachage exact d'arbres

Évaluation de la fréquence des collisions

Afin d'évaluer la qualité de différentes fonctions de hachage, nous quantifions le nombre de collisions générées sur des arbres de taille diverses afin de les comparer aux collisions attendues par l'usage d'une fonction de hachage parfaite. p étant la probabilité de collisions théoriques pour une fonction de hachage parfaite pour une paire arbitraire d'arbres, le nombre de collisions attendues sur un ensemble de n arbres s'élève à $\frac{pn(n-1)}{2}$.

Nous réalisons des tests sur des arbres binaires complets générés aléatoirement avec l'utilisation d'un alphabet de $t_i = 10$ types pour les noeuds internes et $t_l = 10$ types pour les feuilles. En ne considérant que la structure des arbres, il existe $\mathcal{C}_{\lfloor n/2 \rfloor}$ arbres binaires complets de n noeuds (où $\mathcal{C}_k$ est le k-ième nombre de Catalan), la cardinalité étant multipliée par ${t_i}^{\lfloor n/2 \rfloor} + {t_l}^{\lfloor n/2 \rfloor + 1}$ si les types sont pris en comptes. Les fonctions de hachage comparées sont les fonctions de hachage polynomiales avec deux bases différentes 32 et 33. 33 est un nombre premier jugé empiriquement efficace et assez populaire pour le hachage polynomial tandis que la base 32 étant un diviseur des cardinalités des espaces de hachage utilisées est suspectée de générer de nombreuses collisions. Les deux fonctions de hachage cryptographiques choisies sont MD5 et SHA-1.

Nous notons que pour des petits sous-arbres, les fonctions de hachage comparées peuvent occasionner des collisions pour des espaces de sous-arbres de cardinalité inférieure à l'espace de hachage ; là où une fonction parfaite serait bijective. Lorsque l'espace des valeurs est de cardinalité largement inférieure à l'espace des sous-arbres (sous-arbres de volume plus important et/ou valeurs de faible longueur), le nombre de collisions relevées entre fonctions est très similaire et s'approche de la valeur théorique d'une fonction de hachage parfaite. En revanche, comme attendu, l'utilisation d'une valeur de base présentant un PGCD non unitaire avec la cardinalité de l'espace de hachage handicape le hachage de type Karp-Rabin.

Évaluation des performances temporelles de hachage

Nous cherchons à comparer le temps d'exécution de différentes méthodes de hachage proposées. Le hachage par fonction polynomiale étant incrémental, hacher un chaîne est réalisée en un temps proportionnel à sa longueur. Lorsqu'il s'agit d'un arbre, nous hachons une chaîne qui est la représentation sérialisée de celui-ci : des exponentiations de la base sont nécessaires (cf 8.1.2)) ; le hachage se déroule dans le pire des cas en temps $\Theta(N \log N)$ pour un arbre de

Collisions sur arbres de 7 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
55 bits	3,381 656	$-\infty$	$-\infty$	$-\infty$	$-\infty$
41 bits	6,190 223	$-\infty$	$-\infty$	0	$-\infty$
39 bits	6,472 682	0	$-\infty$	0	$-\infty$
38 bits	6,612 903	0,301 030	0	0,477 121	$-\infty$
32 bits	7,223 124	2,113 943	2,079 181	2,056 904	$-\infty$
16 bits	8,908 199	6,882 315	6,882 251	6,882 445	6,881 385
8 bits	10,175 686	9,290 793	9,290 747	9,290 726	9,290 726

Collisions sur arbres de 15 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
63 bits	2,143 015	$-\infty$	$-\infty$	$-\infty$	$-\infty$
41 bits	5,401 569	$-\infty$	$-\infty$	0	$-0{,}643\ 263$
39 bits	5,709 803	0,301 030	0,301 030	0,301 030	$-0{,}041\ 675$
32 bits	6,377 303	1,986 772	2,113 943	2,045 323	2,066 010
16 bits	8,879 466	6,882 563	6,882 547	6,882 554	6,882 490
8 bits	10,175 052	9,290 744	9,290 735	9,290 733	9,290 730

Collisions sur 10^6 arbres de 127 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
64 bits	1,698 970	$-\infty$	$-\infty$	$-\infty$	$-7{,}567\ 030$
40 bits	5,381 287	$-\infty$	$-\infty$	$-\infty$	$-0{,}341\ 989$
39 bits	5,418 374	0	0	0	$-0{,}041\ 436$
32 bits	6,321 192	2,041 393	2,037 426	2,037 426	2,064 458
24 bits	2,426 876	4,473 764	4,474 012	4,473 385	4,474 245
16 bits	8,878 016	6,882 354	6,882 908	6,882 375	6,882 490
8 bits	10,175 532	9,290 719	9,290 716	9,290 721	9,290 730

FIG. 9.3 – Évaluation du nombre de collisions sur un jeu d'arbres binaires aléatoire pour différentes fonctions de hachage exactes

Opération	Temps (UA)
Désérialisation de l'arbre de syntaxe	17,76
Indexation des empreintes	8,23
Hachage additif (structure agnostique)	0,127
Hachage polynomial en base 33	0,210
Hachage cryptographique MD5	0,231
Hachage cryptographique SHA-1	0,329

FIG. 9.4 – Temps d'exécution[a] de différentes méthodes de hachage d'arbre comparé à d'autres opérations

[a]Temps d'exécution expérimentaux pour le traitement du paquetage Netbeans-Javadoc par une version de développement de Plade. Une UA équivaut à 1 seconde en mono-fil avec le JRE Sun 1,6 32 bits sur un CPU Intel Pentium 4 3 Ghz (cache : 2 Mio, RAM : 1 Gio, 5985 bogomips).

N nœuds. En pratique cependant, contrairement au hachage par fonction cryptographique, celui-ci se révèle plus rapide pour des arbres de taille modeste régulièrement rencontrés comme arbres de syntaxes. Il convient toutefois de relativiser l'importance de la rapidité de hachage : si ces opérations sont multipliées par le nombre de profils d'abstraction utilisés, elles sont de coût négligeable rapportées aux opérations d'entrée-sortie de manipulation de base de sous-arbres indexés. Le hachage peut également être facilement parallélisé sur plusieurs processeurs, le calcul de valeurs de hachage de sous-arbres pouvant être mené indépendamment pour le hachage d'un arbre.

La figure 9.4 présente une évaluation de temps d'exécution pour les méthodes de hachage polynomiales ainsi que par fonction cryptographique MD5 et SHA-1. Nous constatons que le temps consacré au hachage est négligeable par rapport au coût temporel d'autres tâches telle que la désérialisation de l'arbre. Même si le hachage polynomial se révèle asymptotiquement moins avantageux, en pratique la taille limitée des sous-arbres manipulés lors du traitement d'arbres de syntaxe occasionne l'usage d'un nombre plus faible d'opérations que les fonctions cryptographiques. Considérant une dispersibilité presque équivalente des fonctions testées, les performances temporelles pratiques, la facilité d'implantation du hachage polynomial de type Karp-Rabin ainsi que sa propriété de hachage incrémental fort constituent des arguments en faveur de son utilisation.

9.2 Abstraction des arbres

9.2.1 Motivations

L'utilisation de métriques permet la création de classes d'équivalence pour les différentes unités structurelles hachées, classes qui ne reflètent pas toujours la similitude structurelle entre les arbres de syntaxes comparés mais plutôt une similitude portant sur des critères ensemblistes. Le hachage exact est lui totalement sensible à toute modification structurelle ce qui ne permet de l'utiliser que pour la recherche d'arbres de syntaxe exactement similaires. Afin de pallier à ce problème et réaliser des recherches de sous-arbres approchés, nous souhaitons introduire un niveau d'abstraction sur les sous-arbres et leur type. L'objectif est ainsi de représenter des sous-arbres relativement proches par une même représentation commune ce

qui garantirait alors une valeur de hachage similaire par un procédé de hachage exact.

9.2.2 Abstraction des types

Principe

L'abstraction des types de nœuds manipulés est déjà utilisée pour la production d'un arbre de syntaxe abstrait. Il s'agit ainsi de représenter des types de nœuds différents par un unique type. En pratique ceci peut être réalisé en attribuant, au sein du vecteur de valeurs de type, des valeurs identiques à plusieurs types.

Exemples

Parmi les choix d'abstraction de types envisageables, nous citons ici quelques exemples que nous avons déjà évoqués en 3.2.2. Certaines opérations d'obfuscation utilisant le remplacement de types par des supertypes, il peut être intéressant des les abstraire partiellement (conservation d'une hiérarchie réduite de types) ou totalement. Les identificateurs peuvent également être abstraits pour lutter contre leur renommage : conserver un profil d'abstraction avec identificateurs non abstraits peut néanmoins demeurer utile pour le dépistage de copies exactes.

D'autres catégories d'abstraction peuvent être réalisées, quoique celles-ci soient moins efficaces qu'un procédé, plus coûteux sémantiquement, de normalisation des arbres de syntaxes. Ainsi des types de boucle peuvent être abstraits mais il faut noter que l'obfuscation par la transformation d'un type de boucle en une autre implique plus de modifications que la simple substitution du type de boucle. Certains opérateurs peuvent également faire l'objet d'une abstraction.

Une abstraction totale des types peut également être réalisée : elle utilise un vecteur de valeurs de type uniforme. Cette abstraction conduit à l'obtention d'une valeur de hachage ne dépendant que de la structure de l'arbre haché.

9.2.3 Abstraction de sous-arbres élémentaires

Abstraire de petits sous-arbres par l'utilisation d'une valeur de hachage commune présente un intérêt pour conserver une valeur de hachage identique pour des arbres n'ayant fait l'object que d'opérations d'éditions locales n'affectant la valeur de hachage que de sous-arbres élémentaires. Si l'abstraction considère comme élémentaires des sous-arbres de taille relativement conséquente, celle-ci permet la recherche de similitudes sur des clones creux (cf 2.1.2) afin de détecter potentiellement certains patrons de conception. Un sous-arbre élémentaire est un sous-arbre qui peut formellement être défini de plusieurs manières :

1. Il peut s'agir d'un sous-arbre dont la racine est d'un type particulier. Ainsi par exemple, des sous-arbres représentant une expression, une instruction voire même le corps d'une fonction entière peuvent être définis comme élémentaires afin d'obtenir plusieurs niveaux d'abstraction.
2. Un sous-arbre élémentaire peut être défini par un seuil de hauteur maximale. Cependant il peut exister des sous-arbres de hauteur faible mais couvrant un volume important du

code source avec un nombre de nœuds conséquent : si seules des opérations d'édition locales doivent être ignorées, le seul critère de hauteur peut être inadapté.

3. Ainsi nous pouvons également définir un sous-arbre élémentaire par un seuil de nombre de nœuds maximal.

9.2.4 Effacement de feuilles et sous-arbres

Effacement complet d'un sous-arbre

Certains sous-arbres des arbres de syntaxe manipulés n'ont quelquefois pas de véritable intérêt sémantique pour la recherche de similitudes. Dans cette catégorie, nous pouvons situer les feuilles liées à l'expression de modificateurs (dont les mots-clés de visibilité d'entités), des sous-arbres liés à des commentaires, à des opérations d'importation de structures externes ou des sous-arbres très fréquents témoignant de code idiomatique. Ces sous-arbres peuvent être aisément supprimés par obfuscation : un procédé de hachage des arbres doit donc de préférence les ignorer.

Effacement d'une racine d'un sous-arbre

Dans certains cas, il peut être utile de supprimer le nœud racine d'un sous-arbre C_i ayant pour parent l'arbre A. Les sous-arbres enfants de C_i sont alors reconnectés directement en tant qu'enfants de leur ex-grand-parent A. Cette modification peut être utile afin de supprimer des structures de contrôle de gestion d'exception ou de délimitation de zones de synchronisation. À l'extrême, il peut être envisagé d'applatir le code en supprimant toutes les structures de contrôle afin de contrer des procédés d'obfuscation liés à la modification de structures de contrôle.

9.2.5 Normalisation de l'ordre des sous-arbres enfants

L'ordre des sous-arbres enfants de certains types de nœuds ne présente pas d'importance. Cette remarque est valable pour les opérateurs commutatifs où l'ordre des opérandes enfants peut être modifié ainsi que pour certaines unités structurelles telles que les unités de compilation ou les classes où, pour certains langages, l'ordre des déclarations de fonctions et de variable peut être modifié sans conséquence sémantique. Les valeurs de hachage obtenues pour deux arbres dont l'ensemble des sous-arbres enfants est identique mais présenté dans un ordre différents ne sont pas égales sauf collision accidentelle. Nous pouvons alors normaliser l'ordre des sous-arbres enfants des nœuds dont le type est commutatif, par exemple en les ordonnant par valeur de hachage croissante.

On note que l'on pourrait normaliser l'ordre des instructions d'un bloc de code après transformation de l'arbre de syntaxe par regroupement des instructions par blocs indépendants pouvant ensuite être transposés.

La généralisation de ce procédé à tous les nœuds permet d'obtenir des classes d'équivalence d'arbres de syntaxe non-ordonnés. Deux arbres de syntaxe non-ordonnés identiques ne caractérisent pas nécessairement deux portions de code source pouvant être considérées comme similaires.

{ L'univers se résume à un trognon de pomme ; Le plus grand arbre est né d'une graine menue ; Le plus grand arbre est représentable en quelques bits. } ⟶ Quelques bits suffisent à représenter le résultat d'un résumé de l'univers.

Marc Gendron, Lao Tseu et un informaticien anonyme

10 Indexation et recherche de sous-arbres

Sommaire

Nous avons étudié au cours du chapitre précédent différentes méthodes de hachage d'arbres de syntaxe ainsi que plusieurs stratégies d'abstraction afin de pouvoir adopter la recherche de similarité à un niveau de détail plus ou moins important. Au-delà de l'obtention des valeurs de hachage pour différents profils d'abstraction se pose la question de leur stockage en vue de rechercher des sous-arbres similaires sur un ensemble conséquent d'unités structurelles de code. À cet effet, nous proposons ici une méthodologie pour l'indexation de différentes valeurs de hachage liées à des sous-arbres hachés pour des profils d'abstraction différents. Cette indexation permet le regroupement des sous-arbres par classes d'équivalence selon différents critères de similarité. L'étude des sous-arbres similaires internes à un ensemble de projets peut ainsi être réalisée par exploitation directe d'une base de classes d'équivalence obtenue après l'indexation de ces projets. Une unité structurelle externe requête peut également être confrontée, après calcul de ses empreintes, à une base afin de déterminer les sous-arbres similaires entre l'unité requête et les projets indexés de la base.

10.1 Indexation de sous-arbres par empreinte unique

Dans un premier temps, nous souhaitons représenter chaque sous-arbre par une empreinte unique correspondant à un profil d'abstraction fixé. Un procédé classique d'indexation implique la maintenance d'une table d'association entre empreintes de hachage et identificateur du sous-arbre correspondant. Les clés de cette table peuvent être indexés par une structure de B+-k-tree, arbre k-aire adapté au stockage de masse limitant les opérations d'entrée-sortie à $\Theta(\log_k(N))$ opérations pour la recherche ou l'ajout d'une empreinte pour N empreintes déjà indexées.

10.1.1 De la longueur des empreintes et de la duplication de sous-arbres

Longueur des empreintes Le principal écueil réside dans la difficulté à choisir une longueur d'empreinte adéquate. En supposant la fonction de hachage sous-jacente parfaite et l'espace des sous-arbres de cardinalité infinie, la probabilité que deux sous-arbres représentés par la même valeur de hachage de longueur k bits soit différents est de $\frac{1}{2^k}$. La probabilité de l'existence d'au moins une collision accidentelle sur une base de N empreintes ($N \leq 2^k$) s'élève à $p_c(k, N) = 1 - \frac{2^k(2^k-1)\cdots(2^k-N+1)}{2^{NK}}$. En supposant que chaque ligne de code soit représentée par 5 empreintes en moyenne[1], la probabilité de l'existence d'une collision accidentelle pour certains projets avec différentes longueurs de valeurs de hachage parfaites est exprimée en

[1]Il s'agit d'une estimation approximative, la densité d'empreintes par ligne de code dépendant du langage, du style de programmation et du seuil de volume minimal pour la conservation d'une empreinte. À titre d'exemple, le paquetage *netbeans-javadoc* (14 360 lignes) possède un volume cumulé d'arbres syntaxiques de 74 351 nœuds (soit 5,20 nœuds par ligne) ; 11 528 empreintes portent sur des sous-arbres d'au moins 20 nœuds).

Lignes de code	thttpd 2.25 11,1K	Apache httpd 2.2 341K	Noyau Linux 2.6.33 11,2M	Projets SourceForge $\sim$ 1G
$k = 32$	$0,30$	$1 - \epsilon$	$1 - \epsilon$	1
$k = 64$	$8,4 \cdot 10^{-11}$	$7,9 \cdot 10^{-8}$	$8,5 \cdot 10^{-5}$	$0,49$
$k = 80$	$1,2 \cdot 10^{-15}$	$1,2 \cdot 10^{-12}$	$1,3 \cdot 10^{-9}$	$1,0 \cdot 10^{-5}$

FIG. 10.1 – Estimation des probabilités de l'existence d'au moins une collision accidentelle d'empreintes de k bits pour différents projets

figure 10.1. Une longueur de valeur de hachage confortable peut rendre négligeable le risque de collision accidentelle et dispense ainsi d'une vérification approfondie de l'égalité de sous-arbres hachés identiquement. Cependant un souci d'économie de mémoire de masse peut nous conduire à limiter cette longueur : un compromis doit être réalisé. Nous proposons donc de choisir une longueur de valeur de hachage faible au démarrage de la constitution de la base et de vérifier incrémentalement, plutôt qu'*a posteriori* lors d'interrogations, la présence de collisions accidentelles. Dans ce cas, nous augmentons la longueur des nouvelles empreintes. La similarité des sous-arbres référencés en base par une valeur de hachage commune et garantie : une telle valeur définit ainsi une classe d'équivalence de sous-arbres selon le profil choisi.

10.1.2 Sélection des empreintes à indexer

Une unité de compilation est généralement représentée par des arbres de syntaxe de grand volume : indexer chacun des nœuds correspondant à un sous-arbre peut s'avérer peu pertinent. Un compromis doit être adopté quant au volume minimal des sous-arbres à indexer. Un volume trop faible conduira à l'indexation de petits sous-arbres comprenant de multiples occurrences, peu utiles pour la mise en évidence de similarités intéressantes s'inscrivant au-delà de clones idiomatiques. Quant à un volume trop élevé, il ne peut permettre la localisation que des clones les plus massifs, laissant masqués des clones plus modestes. L'adoption de certaines techniques d'abstractions comme l'abstraction de petits sous-arbres d'expression peut permettre d'augmenter le seuil de volume minimal d'indexation.

Lorsqu'un volume seuil d'indexation est fixé, les sous-arbres possédant un grand nombre de sous-arbres enfants de volume faible peuvent être eux-mêmes indexés mais pas leurs enfants. Cette situation est rencontrée pour de gros blocs d'instructions où chaque instruction prise individuellement n'est pas indexée mais le bloc entier l'est sous la forme d'une unique empreinte. La granularité d'indexation est alors trop importante. Nous introduisons alors une indexation des sous-arbres enfants sur fenêtre de taille variable : il s'agit d'une généralisation de l'indexation sur fenêtre de taille 1 utilisée jusqu'ici. Pour chaque sous-arbre d'une fratrie, nous démarrons une fenêtre d'indexation. Si le sous-arbre est de volume supérieur au seuil d'indexation, une empreinte le concernant est créée et indexée (fenêtre de taille 1). Dans le cas contraire, la fenêtre d'indexation est étendue aux sous-arbres frères à droite jusqu'à ce que celle-ci englobe une séquence de sous-arbres dont le volume dépasse le seuil d'indexation. Ainsi, l'ensemble d'une fratrie de sous-arbres est couvrable par des empreintes indexées, à l'exception éventuelle des sous-arbres les plus à droite dont le volume cumulé est inférieur au seuil d'indexation[2]. On notera que des empreintes peuvent concerner des séquences chevau-

[2]On pourra, afin de couvrir l'ensemble de la fratrie, ajouter en surplus de l'empreinte concernant la fenêtre la plus à droite une nouvelle empreinte débutant au mêmes sous-arbre mais s'étendant jusqu'à la fin de la

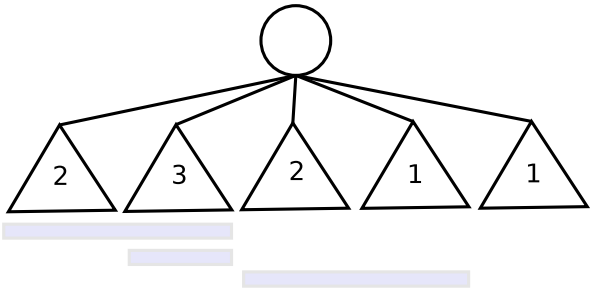


FIG. 10.2 – Indexation sur fenêtre d'une fratrie avec un seuil de volume de 3

chantes de sous-arbres. La figure 10.2 illustre l'indexation d'une fratrie (le dernier frère n'est pas indexé et deux empreintes chevauchent le deuxième frère).

Duplication de sous-arbres Deux sous-arbres $A(a_1, a_2, \cdots, a_k)$ et $A'(a'_1, a'_2, \cdots, a'_k)$ identiques ne doivent pas cacher la forêt de sous-arbres n'ayant pas pour parents A ou A' et pourtant identiques avec un des sous-arbres enfants de A ou A'. Toutefois si les sous-arbres $a_1 = a'_1, a_2 = a'_2, \cdots, a_k = a'_k$ ne sont partagés que par A et A', il paraît superflu de les mentionner explicitement dans la base d'indexation. Nous pourrions ainsi associer leurs valeurs de hachage à un pointeur vers la classe d'équivalence de leur parent (classe contenant A et A') avec leur place dans la fratrie de nœuds. Ainsi, si un nouveau sous-arbre A'' est indexé, celui-ci étant égal à A et A' et rejoignant ainsi leur classe d'équivalence, aucune opération d'indexation n'est nécessaire pour les descendants de A''.

10.1.3 Table de classes d'équivalence et ensembles de membres de classe

Une classe d'équivalence identifie l'ensemble des sous-arbres égaux selon le profil choisi. Une condition nécessaire à l'appartenance de sous-arbres à une même classe réside dans le partage d'une même valeur de hachage. Elle n'est pas suffisante en raison de possibles collisions accidentelles : on considère toutefois l'égalité des sous-arbres acquise lorsque la longueur des valeurs est suffisamment importante.

Au sein d'une structure de table de classes d'équivalence nous associons valeurs de hachage avec un identifiant de classe. Les valeurs de hachage peuvent être de longueur variable ; toutefois il n'existe pas de classe dont la valeur de hachage est préfixe de celle d'une autre classe, ces valeurs représentant donc un code préfixe.

Chaque classe est identifiée par un entier séquentiel. Nous associons à chacune de ces classes l'ensemble de ses membres. Un sous-arbre membre peut être explicité de deux manières :

1. Soit par un pointeur vers lui-même (explicitation directe). Il s'agit généralement d'expliciter l'identificateur de l'arbre englobant d'appartenance, ainsi que la place du sous-arbre membre lors d'un parcours en largeur de l'arbre englobant.

chaîne de sous-arbres.

2. Soit par l'identifiant de la classe d'équivalence de son arbre parent avec spécification de la place du sous-arbre pointé dans sa fratrie (explicitation par classe du parent). Cette solution est naturellement exclue pour les racines d'arbres n'ayant pas de parent.

Nous pouvons récupérer l'ensemble des membres d'une classe par obtention des pointeurs d'explicitation directe et en récupérant récursivement les parents membres dans le second cas pour en déduire par leur rang les enfants membres. Il est toutefois nécessaire de disposer d'une représentation de l'arbre englobant afin de déterminer l'identificateur du sous-arbre à partir de celui de son parent et de sa position dans la fratrie. En raison de l'indirection requise, nous réservons l'explicitation par classe du parent au cas où celle-ci permet une économie mémorielle : c'est uniquement le cas lorsque la classe d'équivalence du parent contient au moins deux membres, ce qui évite la spécification de deux pointeurs pour chacun de leurs sous-arbres respectifs.

10.1.4 Sérialisation des sous-arbres

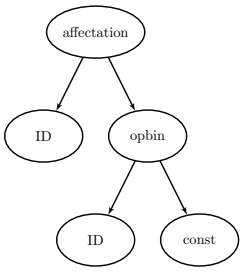
Il est utile de sérialiser les sous-arbres indexés, parallèlement aux tables de classes et de membres. L'objectif est de pouvoir récupérer l'intégralité des nœuds d'un sous-arbre représentatif d'une classe sans disposer d'un référentiel externe ou avoir à re-analyser syntaxiquement une unité de compilation. La forme sérialisée est utilisée pour obtenir explicitement des membres de classe désignés par leur parent et leur rang dans la fratrie ainsi que pour rehacher des sous-arbres en cas de collision accidentelle.

Deux types d'information sont essentielles pour la reconstitution d'un arbre de syntaxe : le type de chacun de ses nœuds (représentable par une valeur entière) et les relations de parenté entre eux. À cet effet, on pourra représenter un arbre sous la forme d'une liste de tuples où le tuple d'indice i explicite l'arbre englobant d'appartenance, le rang du parent ainsi que le rang et le type du nœud racine du sous-arbre de rang i. Le rang d'un nœud est défini par son ordre d'accès lors d'un parcours en largeur. Ainsi les nœuds d'une même fratrie possèdent des rangs consécutifs et le nœud parent d'un nœud est de rang plus bas.

Nous présentons par exemple en figure 10.3 une représentation sérialisée d'une instruction simple. Cette représentation autorise un accès direct aux informations du nœud de rang i ainsi qu'une récupération en temps logarithmique par dichotomie des indices des enfants d'un nœud (les tuples étant triés par rang croissant de sous-arbre parent). Dès lors, la désérialisation d'un sous-arbre peut être menée de façon paresseuse en accédant dans un premier temps à sa racine, puis en récupérant à la demande, niveau par niveau, ses descendants. Nous adjoignons aux informations de type et de parenté un lien de retour vers le code source original afin de pouvoir y localiser des correspondances définies par similarité de sous-arbre de syntaxe.

10.1.5 Algorithme d'indexation

L'algorithme 5 résume le processus d'indexation en base d'un arbre en utilisant les structures précédemment décrites. Celui-ci se décompose en deux étapes fondamentales. La première consiste à déterminer la classe d'équivalence d'appartenance des sous-arbres par leur valeur de hachage. Cette classe étant trouvée ou créée, s'il s'agit du premier exemplaire de ce sous-arbre, nous déterminons dans un second temps si le parent de ce sous-arbre existe déjà en plusieurs exemplaires en base pour adopter son type de spécification dans le classe d'équivalence.



Arbre	Rang racine	Rang parent	Type	Retour
A	1		affectation	k
	2	1	ID	k
	3	1	opbin	$[k..k+1]$
	4	3	ID	$k+1$
	5	3	const	$k+1$

FIG. 10.3 – Forme sérialisée du sous-arbre de syntaxe correspondant à l'instruction `somme = \n somme + 1 ;` (lignes $k \rightarrow k+1$ du code source)

Dans un souci d'économie mémorielle, chaque classe d'équivalence de la base est représentée par une valeur de hachage réduite de longueur variable. Lors de la création d'une nouvelle classe d'équivalence, cette longueur est choisie de telle sorte que la valeur de hachage réduite de la classe ne soit pas préfixe d'une valeur d'une autre classe.

Nous décrivons maintenant plus en détails le processus d'indexation d'un arbre A. Tout d'abord des valeurs de hachage longues sont calculées pour chacun des sous-arbres et sont associées à leur racine : celles-ci peuvent être obtenues en temps linéaire en nombre de nœuds en démarrant des feuilles en remontant vers la racine de l'arbre global comme décrit au chapitre précédent.

Détermination des classes d'équivalence de sous-arbres

Afin de pouvoir indexer chacun des sous-arbres, il est nécessaire de déterminer leur classe d'équivalence d'appartenance, voire d'en créer une nouvelle lorsqu'aucun exemplaire de ce sous-arbre n'existe en base. Pour déterminer la classe d'un sous-arbre $A[k]$ de valeur de hachage longue $\mathcal{H}(A[k])$, on recherche une valeur de hachage réduite dans la table des classes qui soit préfixe de $\mathcal{H}(A[k])$. Soit celle-ci n'est pas trouvée ce qui garantit qu'aucun exemplaire de ce sous-arbre n'a encore été indexé : la création d'une nouvelle classe sera donc nécessaire. Soit une classe est trouvée et il est nécessaire de vérifier l'appartenance réelle de $A[k]$ à celle-ci.

La détermination de classes d'équivalence est réalisée du plus petit sous-arbre de A (feuilles ou plus petits sous-arbres à indexer) au plus grand sous-arbre (A lui-même). Nous cherchons à spécifier pour chaque sous-arbre sa classe d'équivalence ou si celui-ci doit être ajouté dans une nouvelle classe. À cette fin, nous nous basons sur le fait que deux sous-arbres appartiennent à la même classe ssi les trois propriétés suivantes sont respectées :

- leur type de nœud est considéré comme similaire ;
- ils disposent du même nombre de sous-arbres enfants ;
- et les enfants de même rang appartiennent aux mêmes classes d'équivalence.

Ainsi, lorsqu'une classe d'équivalence candidate C_i pour accueillir $A[k]$ est trouvée, nous en extrayons un sous-arbre membre que nous notons r et déterminons si son type de nœud racine est similaire et si ses enfants appartiennent bien au mêmes classes d'équivalence que les enfants de $A[k]$. L'appartenance à une classe d'équivalence étant basée sur une relation de simi-

Données : Ensemble incrémental des classes d'équivalence et de leurs membres $\mathcal{C}$
Données : Arbre A à indexer
début
  Détermination des classes d'équivalence d'appartenance ;
  pour *chaque sous-arbre $A[k]$ de A du plus petit au plus grand (parcours en largeur inverse)* **faire**
    $\mathcal{H}(A[k]) \longleftarrow$ calcul de valeur de hachage longue de $A[k]$ (à partir des valeurs de ses enfants) ;
    $\mathcal{C}_i \longleftarrow$ classe d'équivalence de valeur de hachage préfixe de $\mathcal{H}(A)$;
    $a \longleftarrow$ faux ;
    si $\mathcal{C}_i \neq \emptyset$ **alors**
      $a \longleftarrow$ vérification de non-collision avec un membre indexé de $\mathcal{C}_i$;
      si $\neg a$ **alors**
        Augmentation de la longueur de la valeur de hachage de $\mathcal{C}_i$;
    si $\neg a$ **alors**
      $\mathcal{C}_i \longleftarrow$ nouvelle classe d'équivalence créée ;
  Ajout des sous-arbres comme membres des classes ;
  pour *chaque sous-arbre $A[k]$ de A du plus grand au plus petit (parcours en largeur)* **faire**
    ζ est la cardinalité de la classe d'équivalence $\mathcal{C}_j$ de $\mathcal{P}(A[k])$ ($\zeta \geq 1$) ou 0 si $A[k]$ est la racine ;
    si $\zeta = 0 \vee \zeta = 1$ **alors**
      Spécification explicite d'un pointeur vers α_k ;
    si $\zeta = 2$ **alors**
      Suppression du seul membre explicite de $\mathcal{C}_i$;
      Spécification de la classe du parent $\mathcal{C}_j$ et du rang ξ ;
    si $\zeta > 2$ **alors**
      La classe du parent $\mathcal{C}_j$ et le rang ξ sont déjà mentionnés ;
      Arrêt de l'indexation pour $A[k]$ et ses descendants ;
    $\zeta \longleftarrow \zeta + 1$;
fin

Algorithme 5 : Algorithme d'indexation

larité transitive, la vérification de similarité de $A[k]$ avec un membre quelconque représentant de C_i suffit à démontrer l'appartenance de $A[k]$ à C_i.

Si r est spécifié indirectement par référence à sa classe d'équivalence parent (r représente en fait plusieurs occurrences de sous-arbres similaires), il en va de même de ses enfants $r_1, \cdots, r_n$ mentionnés par lien vers la classe d'équivalence C_i : on recherche ces mentions sur les classes d'équivalence validées de $A[k]_1, \cdots, A[k]_n$.

Si r est spécifié explicitement par mention de son arbre d'appartenance (R) et de son rang (j), alors nous recherchons sur les classes d'équivalence de $A[k]_1, \cdots, A[k]_n$ les enfants de $R[j]$. Cela nécessite de connaître les rangs des enfants de $R[j]$ dans R par désérialisation paresseuse de R.

Résolution des collisions

Pour chacun des sous-arbres $A[k]$ distincts de A à indexer, l'étape précédente a permis de trouver ou non une classe d'équivalence candidate de valeur préfixe de $\mathcal{H}(A_i)$. Si une classe candidate a été trouvée, son adéquation à accueillir $A[k]$ a été vérifiée.

Si aucune classe n'est trouvée, une nouvelle classe doit être créée. Nous choisissons une valeur de hachage réduite pour la représenter de longueur au minimum égale à la plus longue valeur déjà présente en base (la longueur courante l) ; dans le cas contraire, cette valeur réduite pourrait être préfixe d'une autre valeur en base. Cette valeur est mise en correspondance avec un identificateur séquentiel s créé pour la classe et l'on indique l'appartenance de $A[k]$ à s.

Si une classe de valeur de hachage préfixe de $A[k]$ est trouvée et que ses membres (dont un membre r) sont égaux à $A[k]$, $A[k]$ peut intégrer cette classe d'équivalence. Dans le cas contraire, une nouvelle classe doit être créée avec $A[k]$ et la classe de r doit voir sa valeur de hachage représentative allongée pour éviter une collision avec cette nouvelle classe. À cet effet le membre représentatif r fait l'objet d'une opération de rehachage (après désérialisation) pour réobtenir sa valeur de hachage longue afin de compléter la valeur de hachage représentative de la classe.

La longueur des valeurs de hachage représentatives de $A[k]$ et r doit être :
- au minimum égale à la longueur courante l ;
- et plus grande que la longueur du préfixe commun à $\mathcal{H}(A[k])$ et $\mathcal{H}(r)$ afin de pouvoir distinguer les deux classes.

À nombre de classes constant, augmenter la longueur de valeur de hachage de α bits équivaut approximativement à diviser par $2^{\alpha/2}$ la probabilité de l'existence de collision. D'un autre point de vue si une collision accidentelle est rencontrée sur une valeur de hachage de longueur l avec C classes distinctes en bases, allonger sa valeur à $l + \alpha$ bits conduira à une nouvelle collision accidentelle pour un ordre de grandeur de $2^{\alpha/2}C$ classes présentes en base. Le choix de la longueur d'extension α conditionne la fréquence nécessaire de rehachage de représentants de classe.

Indexation

L'indexation des sous-arbres est réalisée par parcours en largeur du plus grand sous-arbre au plus petit sous-arbre de A. Afin de décider de mentionner explicitement le sous-arbre membre $A[k]$ ou plutôt un lien vers la classe d'équivalence C_j de son parent $\mathcal{P}(A[k])$, nous examinons la cardinalité de la classe de C_j (préalablement connue car $\mathcal{P}(A[k])$ a été antérieurement indexé) notée ζ. Quatre situations peuvent être rencontrées en fonction de la cardinalité de C_j :

1. Cas où $A[k]$ est la racine de l'arbre ($A[1]$) et ne possède donc pas de parent. Nous ajoutons alors explicitement un pointeur vers $A[k]$ dans la classe C_i.
2. Cas $\zeta = 1$. Dans cette situation la classe C_j ne contient que le parent $\mathcal{P}(A[k])$ antérieurement indexé. Comme pour le cas précédent, nous ajoutons un pointeur vers $A[k]$ dans C_i.
3. Cas $\zeta = 2$. Avant l'indexation de $\mathcal{P}(A[k])$, C_j était de cardinalité unitaire (avec la mention explicite d'un membre $\mathcal{P}(e)$) ce qui signifie que C_i contient au moins par référence explicite le sous-arbre e. Avec l'indexation de l'arbre A, C_j perd sa cardinalité unitaire et contient deux membres : il n'est donc plus utile de mentionner explicitement sur C_i les enfants de ses membres. Le sous-arbre e explicité sur C_i est donc supprimé et remplacé par la spécification de la classe d'équivalence parent C_j et du rang dans la fratrie, spécification englobant également le nouveau sous-arbre $A[k]$.
4. Cas $\zeta \geq 3$. Lors de l'indexation antérieure d'une occurrence de sous-arbre similaire à $A[k]$ appartenant à C_i, une mention vers la classe d'équivalence parent et le rang fraternel a déjà été réalisée. Aucune action n'est nécessaire pour indexer $A[k]$ déjà présent implicitement par la mention explicite en base de son plus proche ancêtre de parent non-dupliqué.

Complexité

Les opérations d'accès, d'ajout et de suppression d'une classe peuvent être menées en temps logarithmique du nombre de classe $\log |C|$ par un arbre d'indexation. Le hachage d'un arbre A nécessite de s'interroger sur l'appartenance de chacun de ses sous-arbres $A[k]$ à une classe d'équivalence. Dans le pire des cas, il peut être nécessaire de rechercher tous les enfants d'un représentant de la classe d'équivalence candidate dans les classes d'équivalence des enfants de $A[k]$ ce qui est réalisé en $\Theta(\log P)$ où P est le nombre maximal de pointeurs d'une classe d'équivalence. Il faut y ajouter le temps de désérialisation nécessaire à l'obtention du rang des enfants du représentant en $\Theta(\log N)$ où N est le nombre de sous-arbres indexés en base. Globalement, le processus d'indexation pour A requiert un temps en $\Theta(|A| \log(|C|PN))$.

Nous avons toutefois omis de discuter du coût lié au rehachage. Celui-ci peut être évité par l'emploi, dès la création de la base, d'une longueur de valeur de hachage suffisamment longue au prix d'un coût mémoriel plus important. Cette longueur initiale ainsi que le nombre de bits ajoutés par rehachage α conditionne le nombre de rehachage prévisible. Le coût asymptotique de rehachage amorti pour $|C|$ classes d'équivalence est équivalent à $\frac{|C| \log_2 |C|}{\alpha/2} t_d$ où t_d est le coût de désérialisation moyen d'un sous-arbre.

10.1.6 Exemple

Afin d'illustrer le principe d'indexation précédemment décrit, nous introduisons un petit exemple par le hachage et l'indexation des arbres A et B suivants où a et b sont deux types de nœuds utilisés :

$$\begin{aligned} A = &\quad a(b,a) \\ B = &\quad b\,(a(b,a), a(b,a), a) \end{aligned}$$

A est composé de trois sous-arbres b, a et l'arbre complet $a(b,a)$. Nous leur assignons des valeurs de hachage longues (arbitraires pour l'exemple) de 4 bits : $\mathcal{H}(b) = 1100$, $\mathcal{H}(a) = 0100$ et $\mathcal{H}(a(b,a)) = 1010$. La longueur initiale de valeur de hachage réduite est fixée à 2 : aucune collision n'est alors relevée pour les valeurs réduites des sous-arbres de A, le plus long préfixe commun étant de longueur 1. Nous obtenons la table de classes et la table de membres de la figure 10.4

Valeur de hachage	Classe	Sous-arbre	Membres explicites
10	1	$a(b,a)$	$A[1]$
11	2	b	$A[2]$
01	3	a	$A[3]$

FIG. 10.4 – Membres des classes après indexation de $A = a(b,a)$

Nous indexons maintenant B qui comporte 4 sous-arbres distincts dont nous spécifions les valeurs de hachage longues : $\mathcal{H}(a) = 0100$, $\mathcal{H}(b) = 1100$, $\mathcal{H}(a(b,a)) = 1010$ (nous utilisons une abstraction ignorant l'ordre des enfants d'un sous-arbre) et $\mathcal{H}(B = b(a(b,a), a(b,a), a)) = 1001$. Pour chacune de ces valeurs, nous vérifions s'il existe une classe sur la base de valeur de hachage préfixe : si celle-ci existe, nous déterminons si la classe est adaptée ou s'il s'agit d'une collision accidentelle. Pour a et b les classes de valeurs de hachage préfixes 11 et 01 existent et sont adéquates. Pour le sous-arbre $a(b,a)$ de valeur 1010, la classe de valeur préfixe trouvée est celle d'identifiant 1 avec le préfixe commun 10 : son unique membre a les mêmes sous-arbres enfants b et a. Nous pouvons ajouter les deux sous-arbres $a(b,a)$ de B à cette classe. Enfin pour $\mathcal{H}(\mathcal{B})$, nous relevons la classe d'équivalence 1 ayant également pour valeur de hachage préfixe 10. Les trois sous-arbres enfants de B ne correspondent cependant pas aux deux sous-arbres enfants b et a d'un membre de la classe 1, $A[1] = a(b,a)$. Une collision accidentelle est relevée : elle nécessite un rehachage de la valeur de hachage de la classe 1 en 101 et la création d'une nouvelle classe pour ajouter B avec pour valeur 100.

Lorsque nous ajoutons les occurrences de sous-arbres de B comme membres des classes d'équivalence adéquates de la base, nous explorons les sous-arbres de B du plus grand au plus petit. B est d'abord ajouté dans une classe nouvelle. Les deux occurrences de $a(b,a)$ ont pour parent commun le sur-arbre B présent en unique exemplaire dans la base : ils sont donc ajoutés explicitement.

Concernant le sous-arbre b, lors de l'ajout de sa première occurrence dans B nous constatons que la classe d'équivalence de son parent (A) contient déjà deux membres, $A[1]$ ainsi que le

premier sous-arbre $a(b, a)$ $B[2]$ de B. Nous supprimons donc la mention explicite vers l'enfant $A[2]$ de $A[1]$ et ajoutons une référence vers la classe d'équivalence parent 1 et le rang fraternel (1). Lors de la rencontre de la seconde occurrence de b dans B également enfant de $a(b, a)$ ($B[3]$) de classe d'équivalence parent de cardinalité désormais 3, aucune opération d'indexation n'est réalisée ; $B[3]$ étant référencé explicitement dans la classe d'équivalence 1.

Les occurrences du sous-arbre a dans B sont indexés analoguement aux occurrences de b. Nous obtenons finalement les tables de classe d'équivalence et de membres spécifiées ci-après en figure 10.5 :

Hachage	Classe	Sous-arbre	Membres explicites	Membres par classe du parent
101	1	$a(b, a)$	$A[1], B[2], B[3]$	
11	2	b		$(1, 1)$
01	3	a	$B[3]$	$(1, 2)$
100	4	$b(a(b, a), a(b, a), a)$	$B[1]$	

FIG. 10.5 – Membres des classes après indexation de $A = a(b, a)$ et $B = b(a(b, a), a(b, a), a)$

10.2 Familles de classe d'équivalence

Plutôt que d'être contraint à la recherche de similitudes avec un niveau immuable d'abstraction, il pourrait être utile d'indexer les sous-arbres des arbres de syntaxe selon plusieurs abstractions afin de proposer des critères de recherche de similarité plus flexibles. Une première méthode consiste à conserver des bases de valeurs de hachage indépendantes pour chaque abstraction. Dans cette optique, nous pourrions par exemple stocker pour chaque sous-arbre une empreinte pour un profil abstrayant les identificateurs, un second les types et les identificateurs, un troisième ajoutant une abstraction des sous-arbres d'expression de taille inférieure à un seuil... Cette première approche, bien que fonctionnelle, induit l'apparition d'informations redondantes et ne permet pas directement de déduire des relations d'inclusion entre classes d'équivalence de différents profils.

Proposer des méthodes de génération d'empreintes utilisant uniquement des propriétés caractéristiques ensemblistes de l'arbre manipulé pourrait s'avérer utile. Ces empreintes peuvent être obtenues par hachage de vecteurs caractéristiques de chaque sous-arbre. L'objectif n'est cependant ici pas de proposer des valeurs de hachage approchées dont la proximité signifierait une certaine similarité des sous-arbres qu'elles représentent. Il s'agit plutôt d'utiliser l'égalité exacte de telles valeurs afin de déduire des sous-classes d'équivalence plus précises entre arbres appartenant à une même classe d'équivalence plus générale. On pourra par exemple adjoindre à un profil abstrayant les types, identificateurs et commentaires des valeurs de hachage sur des vecteurs d'existence de types, d'identificateurs et de mots de commentaire.

10.2.1 Famille de classes d'équivalence

Définition 10.1. Famille de classes d'équivalence. Une famille de classes d'équivalence est un ensemble de classes d'équivalence contenant des sous-arbres définie par une fonction booléenne

transitive f d'égalité de sous-arbres. Pour tout couple de sous-arbres (A, B) d'une classe d'équivalence $\mathcal{C}_i$ de la famille, $f(A, B) =$ vrai alors que pour tout couple de sous-arbres (C, D) de classes distinctes, $f(A, B) =$ faux.

Afin d'indexer les sous-arbres, nous les classons dans un ensemble de familles de classes d'équivalence. Chaque famille correspond à un profil d'abstraction déterminé.

10.2.2 Graphe de familles de classe d'équivalence

Graphe

Afin de caractériser la similarité de sous-arbres selon des critères plus ou moins précis, nous utilisons plusieurs familles de classes d'équivalence. Il s'agit de créer une taxonomie hiérarchique des sous-arbres à l'aide de familles de classes. Nous organisons les familles de classes d'équivalence sous la forme d'un graphe orienté acyclique avec une relation de spécialisation des profils d'abstraction. Nous distinguons parmi ces familles plusieurs types selon leur arité entrante et sortante :

- les familles sources d'arité entrante nulle ;
- les familles feuilles d'arité sortante nulle ;
- et les familles composites d'arité entrante d'au moins deux.

Familles sources Les familles sources contiennent les classes d'équivalence racines des sous-arbres ; elles représentent ainsi les profils d'abstraction les plus généraux. Elles utilisent généralement une méthode de hachage pour le classement des sous-arbres en classe d'équivalence.

Familles feuilles Les familles feuilles représentent les profils les plus spécialisés. Elles sont les seules à accueillir des tables de sous-arbres membres. Lorsque l'énumération de l'ensemble des membres d'une classe d'équivalence c_1 d'une famille non feuille f_1 est requise, il est nécessaire de trouver une branche menant de f_1 vers une famille feuille f_n. Nous déterminons ensuite itérativement l'ensemble des sous-classes c_2 de c_1 sur f_2, des sous-classes de c_3 de c_2 sur f_3, ..., jusqu'à obtenir l'ensemble des sous-classes c_n sur la famille feuille f_n et déterminer ensuite les membres de c_n représentant les membres de la classe c_1.

Familles composites et index Les familles composites permettent l'obtention d'un profil d'abstraction plus spécialisé avec la prise en considération de plusieurs familles entrantes. La fonction d'égalité F d'une famille composite issue de k parents de fonctions f_1, f_2, ..., f_k est définie par $F = f_1 \wedge f_2 \wedge \cdots \wedge f_k$. Ainsi, deux sous-arbres A et B appartiennent à la même classe de la famille composite si leurs classes d'appartenance sont identiques pour toutes les familles entrantes. Une classe d'équivalence de c est donc définie par le k-uplet de ses classes d'équivalence entrantes.

Il est utile depuis une classe d'équivalence de c de f_i d'obtenir les sous-classes d'équivalence sur la famille composite F : cela nécessite l'indexation des classes de F par un critère de tri sur la classe de f_i.

Un exemple de graphe de familles de classes d'équivalence

Nous présentons ici un exemple de graphe de familles de classes d'équivalence adapté au langage Java. Nous définissons à cet effet les profils d'abstractions suivants correspondant chacun à une famille de classes d'équivalence que nous organisons en graphe de spécialisation :

- Le profil *funcAbstr* d'abstraction des corps de fonction. Seules les signatures des déclarations de méthodes (avec abstraction des identificateurs) ainsi que les autres membres des classes sont conservées. Ce profil permet de répérer des schémas de conception spécifiques. Il s'agit d'une famille source.
- Le profil *subAbstr* d'abstraction des petits sous-arbres de taille inférieure (en nombre de nœuds) à un seuil t fixé. Seuls les petits sous-arbres contenu dans le corps d'une méthode sont abstraits alors que les nœuds d'unités ultra-fonctionnels sont ignorés.
- Le profil *funcSubAbstr* est une famille composite issue de *funcAbstr* et *subAbstr*. Les sous-arbres infra-fonctionnels et fonctionnels similaires à de petits sous-arbres près abstraits sont regroupés dans les mêmes classes.
- Le profil *nodeAbstr* d'abstraction des types de nœuds. Seule la structure de l'arbre de syntaxe, sans des éléments sémantiquement inintéressants, est conservée.
- Le profil *nodeSubAbstr* est une famille composite de classes d'équivalence obtenue depuis *funcSubAbstr* et *nodeAbstr* : cette famille abstrait uniquement les types de nœud (et considère ainsi la structure uniquement) des petits sous-arbres.
- La famille source *contAbstr* permet l'obtention d'empreintes supprimant les nœuds de structures de contrôles et abstrayant types, identificateurs et éléments sémantiquement inintéressants (commentaires, modificateurs).
  On en déduit une famille composite *typeAbstr* prenant pour parents *contAbstr* mais aussi *nodeSubAbstr*. Cette famille abstrait également types, identificateurs et éléments sémantiquement inintéressants mais ne supprime plus les structures de contrôles. Toutefois si celles-ci étaient de taille inférieure au seuil d'abstraction de sous-arbres introduit par *subAbstr*, seule leur structure serait conservée.
- La famille source *typeSet* est basée sur la génération d'empreintes sur l'ensemble des types spécifiés dans un sous-arbre. Analoguement, la famille source *idSet* se base sur des empreintes d'ensemble d'identificateurs de sous-arbre tandis que *commentSet* hache des vecteurs de présence de mots dans les commentaires.
- Nous déduisons, à partir de *typeAbstr*, trois familles composites spécialisées utilisant pour autre parent pour l'une *typeSet* (*typeAbstr+typeSet*), pour l'autre *idSet* (*typeAbstr+idSet*) et enfin pour la troisième *commentSet* (*typeAbstr+commentSet*).
- Enfin une dernière famille *minAbstr* composite d'arité sortante nulle utilise pour parent *typeAbstr+{typeSet,idSet,commentSet}*. Les classes d'équivalence qu'elle contient sont les plus spécialisées et caractérisent des sous-arbres de structures et de types de nœuds identiques avec ensembles partagés de types, identificateurs et mots de commentaires. Nous notons que des nœuds de structures de contrôle racines de petits sous-arbres peuvent être ignorés.

Le graphe de ces familles de classes d'équivalence peut être ainsi exprimé :

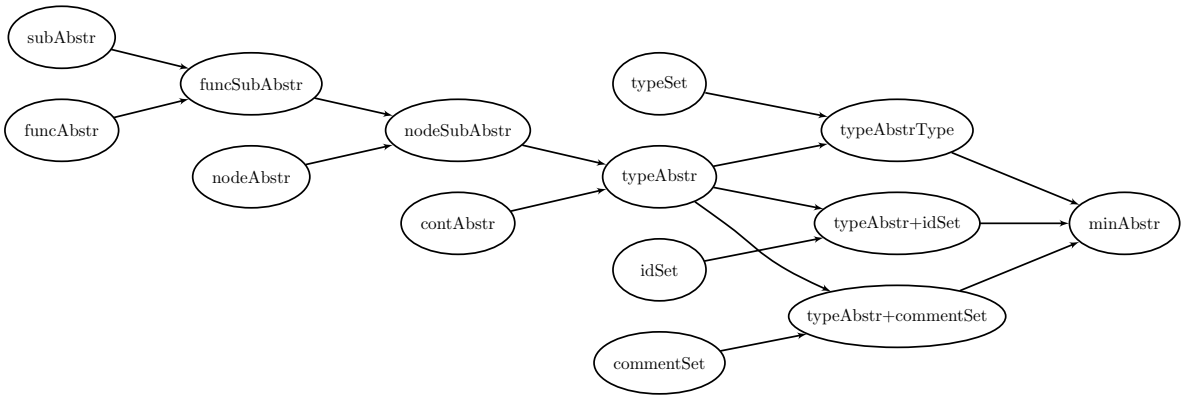


FIG. 10.6 – Un graphe de familles d'abstraction

10.3 Indexation selon un graphe de familles d'équivalence

10.3.1 Structures d'indexation

L'indexation d'arbres de syntaxe selon un graphe de familles nécessite la connaissance des classes des familles sources, la composition de chacune des classes des familles composites en terme de familles entrantes ainsi que les membres des classes des familles sources.

Famille source Comme décrit en 10.1.3, les familles sources, dont la définition des classes d'équivalence est basée sur des valeurs de hachage, maintiennent chacune une table associant valeur de hachage à l'identifiant de la classe.

Famille feuille Une famille feuille représente un profil le plus spécialisé. Nous lions à ces classes d'équivalence la liste de ses sous-arbres membres soit par spécification directe, soit par spécification de la classe d'équivalence du sur-arbre parent avec rang dans la fratrie comme discuté en 10.1.3.

Famille composite Une famille composite possède un index liant les k-uplets des identifiants de ses familles entrantes à un identifiant de classe sur la famille composite. Un index ne considère pour le tri des classes qu'une des permutations pour l'ordre de spécification des identifiants du k-uplet. Ainsi, si l'on souhaite connaître les sous-arbres membres d'une classe d'équivalence d'une famille entrante f_i sur la famille composite F, il est nécessaire que le premier critère de tri de l'index sur les k-uplets porte sur la famille f_i. Dans le cas contraire, la recherche nécessiterait le parcours exhaustif de toutes les classes d'équivalence de la famille composite pour y déceler celles de famille entrante f_i. Il faut donc maintenir au moins k tables de tri afin que chaque famille entrante fasse l'objet d'un premier critère de tri.

10.3.2 Procédure d'indexation

La procédure d'indexation sur un graphe de familles s'avère comparable à celle pour un profil unique décrite en 10.1.5. Les familles feuilles peuvent être ainsi assimilées à des profils uniques et la concaténation de classes d'équivalence des familles entrantes comme une valeur de hachage (sans problématique de collision). Quelques précisions spécifiques sont néanmoins

à apporter sur l'étape préliminaire de détermination de classe d'équivalence pour chaque sous-arbre d'un arbre A à indexer.

Classes d'équivalence sur les familles sources Les classes d'équivalence d'un sous-arbre donné pour toutes les familles sont définies sans ambiguïté par la connaissance des classes d'équivalence de ce sous-arbre selon les familles sources.

Pour chaque famille source, nous déterminons la classe d'appartenance candidate d'un sous-arbre $A[k]$ en calculant la valeur de hachage longue de ce sous-arbre pour l'abstraction considérée et en obtenant la classe dont la valeur de hachage représentative est préfixe de la valeur longue calculée pour le sous-arbre. Soit une classe candidate de valeur préfixe est trouvée : il faut alors valider l'appartenance du sous-arbre $A[k]$ à cette classe ; soit aucune classe n'est trouvée et le sous-arbre $A[k]$ appartient à une nouvelle classe.

Pour vérifier que la classe candidate est adaptée au sous-arbre $A[k]$, comme présenté en 10.1.5 nous analysons chaque sous-arbre de l'arbre à indexer du plus petit au plus grand afin de pouvoir sélectionner sur une classe candidate un sous-arbre représentant déjà indexé pour lequel nous vérifierons qu'il existe des enfants appartenant aux mêmes classes d'équivalence que les enfants de A_i.

Cette procédure nécessite de déterminer un membre représentatif d'une classe d'équivalence d'une famille, ce qui nécessite de suivre une branche menant à une feuille pour obtenir la hiérarchie des sous-familles ainsi que tous leurs membres par une famille feuille. Il est nécessaire de prendre en compte les opérations d'abstraction de la famille considérée : certains sous-arbres peuvent en effet être supprimés par le profil choisi voire une racine de sous-arbre supprimée. La complexité temporelle de recherche des sous-arbres enfants d'un arbre représentant d'une classe d'une famille est multiple du nombre de sous-classes de cette classe sur la famille feuille.

10.3.3 Implantation des structures d'indexation

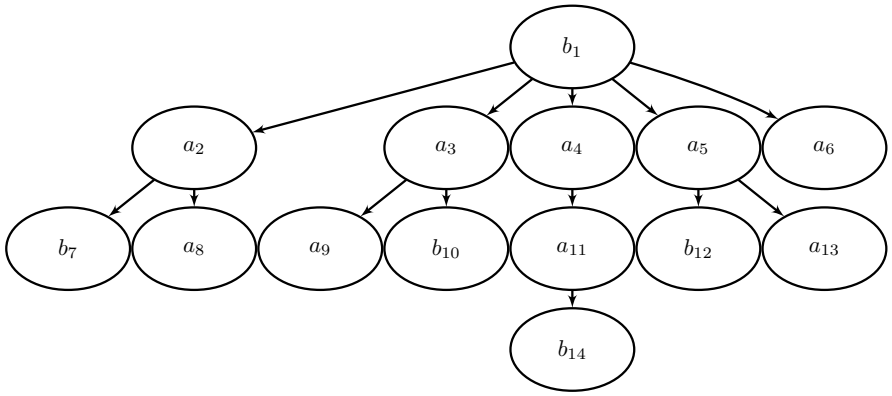
Arbre d'indexation adapté aux supports de masse L'implantation des tables classes d'équivalence pour chaque famille ainsi que les tables de membres pour les profils feuilles doit être réalisé en utilisant des structures d'indexation qui puissent être adaptées à l'usage de mémoire de masse. L'utilisation de structures d'arbres binaires équilibrés classiques peut nécessiter $N \log_2 N$ opérations de lecture ou écriture de bloc disque pour l'accès ou l'écriture d'un élément de l'index sans compter les opérations d'équilibrage. Nous pouvons plutôt opter pour l'usage de k-B+-tree qui sont des arbres d'arité k à $2k$ (avec k de valeur ajustée à la taille d'un bloc de disque) dont les nœuds internes contiennent des clés d'indexation et les feuilles les valeurs indexées. Cette structure permet de réduire le nombre d'accès disque (division par un facteur de $\frac{\ln k}{\ln 2}$) liés au parcours de l'arbre et à son équilibrage. Il est possible de déléguer la tâche d'indexation à un système de gestion de base de données généraliste (tel que PostgreSQL) avec un surcoût lié au traitement des requêtes SQL et aux opérations de communication.

Répartition sur plusieurs supports Les index peuvent aisément être répartis sur K supports distincts par l'usage d'un critère de répartition simple des valeurs de hachage et des identificateurs. Ainsi, les classes d'équivalence dont i est préfixe de la valeur de hachage peuvent être indexés sur le support i alors que les membres d'une classe d'équivalence d'identificateur i peuvent être stockés sur le support $i \mod K$. La répartition par famille est sans doute moins

avantageuse car offrant moins de garantie d'équilibrage du volume des données stockés sur chaque support : chaque famille comporte un nombre de classes d'équivalence hétérogène.

10.3.4 Exemple d'indexation

Nous présentons un exemple d'indexation d'un arbre de syntaxe, avec, dans un souci de simplification, l'utilisation de trois familles de classes d'équivalence dont deux familles sources et une famille composite feuille. La première famille ϕ_1 réalise une abstraction de tous les types de feuilles par une feuille représentante unique : seule la structure des sous-arbres est considérée. Une seconde famille ϕ_2 représente les sous-arbres par la suite brute[3] de ses nœuds sérialisés par un parcours en largeur. Enfin, ϕ_3 est une famille composite feuille ayant pour parents ϕ_1 et ϕ_2 et dont chacune des classes contient les occurrences d'un sous-arbre non-abstrait (structure et types sont pris en compte). Une branche d'intérêt $\phi_1 \longrightarrow \phi_3$ est définie par la présence d'un index sur ϕ_3 triant les couples de classes de ϕ_1 et ϕ_2 selon ϕ_1. Seuls deux types de nœuds a et b sont utilisés. Nous indexons l'arbre $A = b(a(b,a), a(a,b), a(a(b)), a(b,a), a)$. Nous présentons ci-dessous en figure 10.7 les classes d'équivalences de la famille feuille ϕ_3 avec leur composition en classes de ϕ_1 et ϕ_2. Cette table induit la présence de deux index (que nous n'explicitons pas), le premier trié selon la classe de la famille entrante ϕ_1 et le second selon ϕ_2.



Classes de ϕ_3	Composition en classes de ϕ_1 et ϕ_2	Sous-arbre	Membres explicites	Membres par classe du parent
1	$(\phi_1 : 1, \phi_2 : 1)$	b	$A[10], A[14]$	$(4, 1)$
2	$(\phi_1 : 1, \phi_2 : 2)$	a	$A[6], A[9]$	$(4, 2)$
3	$(\phi_1 : 2, \phi_2 : 3)$	$a(b)$	$A[11]$	
4	$(\phi_1 : 3, \phi_2 : 4)$	$a(b, a)$	$A[2], 4[5]$	
5	$(\phi_1 : 4, \phi_2 : 5)$	$a(a(b))$	$A[4]$	
6	$(\phi_1 : 5, \phi_2 : 5)$	$a(a, b)$	$A[3]$	
7	$(\phi_1 : 6, \phi_2 : 6)$	A	$A[1]$	

FIG. 10.7 – Classes d'équivalence d'une famille composite feuille pour l'indexation de l'arbre A

[3]Cette suite brute de nœuds sérialisés ne comporte pas d'information sur la structure de l'arbre : un nœud n'est pas relié à son parent.

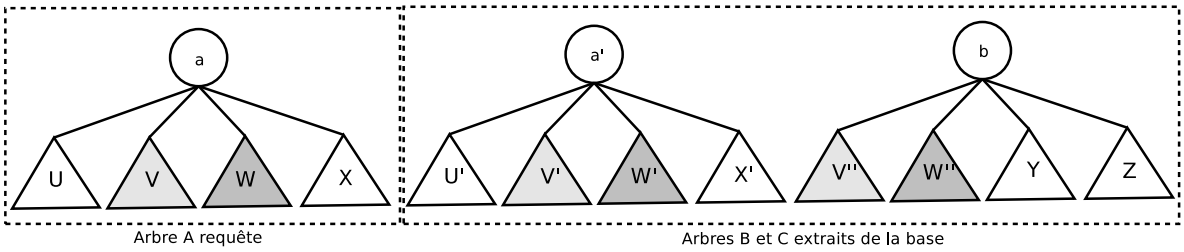


FIG. 10.8 – Un arbre requête A, son dupliqué B sur la base et un arbre de la base C comportant des similarités.

10.4 Recherche de similarité sur arbre requête

10.4.1 Problématique

De part l'organisation des index, la recherche de classes d'équivalence d'un sous-arbre requête sur l'ensemble des arbres indexés est presque immédiate pour une famille donnée. Il suffit de parcourir la branche des familles vers la famille feuille afin de récupérer l'arbre des classes d'équivalence ainsi que les sous-arbres membres directs et les sous-arbres membres par similarité de leur parent.

Nous présentons ici une méthode de recherche de chaînes de sous-arbres frères similaires appartenant à un arbre requête A présents dans la base indexée d'arbres $\mathcal{B}$. Une première étape de récupération de sous-arbres unitaires similaires est réalisée. Pour tout sous-arbre $A[k]$ de A, nous recherchons l'ensemble des sous-arbres de $\mathcal{B}$ similaires selon le profil considéré.

Nous imposons une condition de non-recouvrement des correspondances dans l'arbre. En d'autres termes si le sous-arbre $A[k]$ de A est égal au sous-arbre $B[k']$ de $B \in \mathcal{B}$, cette correspondance est reportée ssi $\mathcal{P}(A[k])$ et $\mathcal{P}(B[k'])$, les sur-arbres parents respectifs de $A[k]$ et $B[k']$ ne sont pas égaux. Pour l'exemple présenté en figure 10.8, nous reportons une correspondance entre les sous-arbre A et B mais pas entre leurs sous-arbres enfants égaux.

Notons que si nous trouvons dans $\mathcal{B}$ une classe d'équivalence de cardinalité non nulle C pour un sous-arbre $A[k]$ (ayant des enfants $A[k]_1, \cdots, A[k]_n$), cela ne signifie pas que la recherche de sous-arbres similaires doit être interrompue pour les descendants de $A[k]$. Il peut être possible de trouver, sur $\mathcal{B}$ des sous-arbres égaux à un descendant de $A[k]$ qui ne soient pas eux-mêmes des descendants des arbres membres de C. Par exemple sur les arbres de la figure 10.8, les classes d'équivalences contenant V, V', V'', et W, W', W'' doivent être prises en considération.

Dans un second temps, nous déterminons les facteurs répétés sur les séquences de sous-arbres. À cet effet, nous utilisons une structure d'indexation de suffixes (cf chapitre 6) pour obtenir un graphe des farmax des chaînes de nœuds frères similaires. Pour l'exemple traité, nous trouverions une correspondance entre les arbres A et B ainsi qu'une correspondance sur les chaînes de sous-arbres V, W, V', W' et V'', W''.

10.4.2 Recherche de sous-arbres individuels égaux

Pour rechercher des sous-arbres de A similaires à des sous-arbres indexés en base, nous procédons comme si nous souhaitions indexer A en base. Nous déterminons pour chacun des sous-arbres de A leur classe d'équivalence d'appartenance pour chacun des profils en s'assurant de l'absence de collision accidentelle entre la valeur de hachage de la classe préfixe binaire de la valeur de hachage longue du sous-arbre de A. Pour les besoins de la recherche, nous nous intéressons uniquement aux sous-arbres égaux selon une certaine famille f assez générale : nous notons les classes d'équivalence de cette famille auxquelles appartient chaque sous-arbre $A[k]$ de A.

Chaque sous-arbre $A[k]$ est ainsi lié à une classe d'équivalence de f avec une hiérarchie de sous-classes afférentes selon les familles plus spécialisées. À partir de cette hiérarchie, nous pouvons obtenir les sous-classes de famille feuille avec leurs membres. Pour chacun des membres de la classe sur la base, nous vérifions si son parent appartient à la même classe que le parent de $A[k]$: si c'est le cas, nous l'ignorons car il est déjà englobé par la similarité de son parent.

10.4.3 Recherche par hachage de chaînes de sous-arbres égaux

Disposant des correspondances entre sous-arbres unitaires de A et classes de correspondance sur la base $\mathcal{B}$, nous souhaitons déterminer les facteurs répétés maximaux de chaînes de sous-arbres tels que définis au chapitre 6. La recherche de facteurs répétés maximaux est intéressante pour les nœuds d'arité importante pour lesquels l'ordre des sous-arbres enfants possède une signification sémantique. Ainsi par exemple, en considérant le langage Java, il est inutile de rechercher des facteurs répétés sur les enfants de nœuds tels que les classes alors que la recherche s'avère incontournable pour les blocs d'instruction afin de localiser des instructions consécutives similaires. On notera toutefois que la recherche de facteurs répétés est inefficace contre des opérations d'obfuscation par insertion ou suppression de code inutile lorsque ces opérations segmentent les chaînes de sous-arbres frères similaires en sous-chaînes de trop faible longueur. Nous introduisons dans le chapitre suivant une méthode d'extension plus adaptée à ces situations permettant de réunir des similarités spatialement proches dans leur arbre de syntaxe.

Transformation des arbres n-aires en arbres binaires

Une première approche pourrait consister à transformer les arbres d'arité forte en arbres binaires avec des pointeurs vers le fils gauche et le frère droit (ou symétriquement vers le fils droit et le frère gauche). Ceci augmente le nombre de sous-arbres et donc la taille de la base pour l'indexation mais permet la recherche de préfixes (ou suffixes) communs sur la séquences des enfants de nœuds des arbres originaux d'arité forte. Toutefois la recherche de facteurs quelconques demeure impossible.

Hachage exhaustif de sous-chaînes

Une deuxième approche, introduite par Baxter et al. [63] pour leur outil de recherche de similitudes sur des arbres de syntaxe, utilise le hachage exhaustif des sous-chaînes de sous-arbres d'une même fratrie. Nous déterminons tout d'abord les fratries candidates susceptibles de participer à des correspondances. Une fratrie est candidate (aussi bien sur A que sur les

arbres de la base $\mathcal{B}$) ssi au moins deux arbres de la fratrie participent à des correspondances. Une base d'indexation est alors créée pour accueillir le résultat du hachage selon les différents profils d'abstraction de toutes les sous-chaînes de sous-arbres des fratries candidates. Nous pouvons généraliser les méthodes de hachage sur les arbres présentées au chapitre 9 pour les chaînes d'arbre en considérant une chaîne d'arbre $A_1 A_2 \cdots A_k$ tel un arbre de racine virtuelle comportant pour enfants les sous-arbres $A_1, A_2, \cdots, A_k$.

Les valeurs de hachage de chaque sous-chaîne d'arbres de la chaîne $A_1 A_2 \cdots A_k$ d'une fratrie étant précalculées, $\frac{k(k+1)}{2}$ empreintes doivent être déterminées correspondant chacune à une sous-chaîne. En calculant les empreintes par longueur croissante des sous-chaînes, il est possible d'exploiter la propriété d'incrémentalité de la fonction de hachage utilisée en calculant l'empreinte d'une sous-chaîne de longueur $i+1$ en utilisant l'empreinte de la sous-chaîne préfixe de longueur i. Chaque empreinte peut ainsi être obtenue en temps constant.

En parcourant les classes d'équivalence de la base temporaire des sous-chaînes de fratrie et en ne considérant que les classes contenant au moins une sous-chaîne de fratrie de A, nous obtenons l'ensemble des sous-chaînes de fratrie similaires. Toutefois, aucune information sur les recouvrements de ces sous-chaînes n'est exploitable. De plus, s'il existe de nombreuses fratries candidates d'arité importante, la complexité temporelle peut devenir rédhibitoire. Si nous considérons des fratries d'instructions au sein de blocs, sauf à effacer les instructions les plus fréquentes des arbres de syntaxe, le ratio de fratries d'instructions candidates sur le nombre total de fratries sur $\mathcal{B}$ peut approcher 1.

10.4.4 Détermination des farmax sur chaînes de sous-arbres frères

Principe général

Une structure d'indexation de suffixes telle qu'un arbre ou une table de suffixes est employée afin d'obtenir les facteurs répétés maximaux complets de sous-chaîne de fratrie.

Détermination de classes d'équivalence de nœuds La première étape consiste à obtenir les classes d'équivalence selon la famille considérée contenant les sous-arbres de l'arbre requête A ainsi que tous ceux de $\mathcal{B}$ correspondant à un sous-arbre de A. Chacune des classes prise en compte contient donc au moins une occurrence de sous-arbre de A. De plus, nous éliminons les classes dont tous les membres partagent un sous-arbre parent similaire.

Inventaire des chaînes de sous-arbres frères consécutifs Les chaînes de sous-arbres frères consécutifs sur A et $\mathcal{B}$ appartenant aux classes sélectionnées sont reportées. Ceci est réalisé en triant l'ensemble des sous-arbres α participant aux classes par couple (rang($\mathcal{P}(\alpha)$), rang(α)) : une chaîne de l sous-arbres frères consécutifs est modélisée par une suite de couples $(i,j),(i,j+1),\cdots,(i,j+l-1)$, sans possibilité d'extension sur la gauche ou la droite. Ces frères ont des identificateurs consécutifs par parcours en largeur et possèdent le même arbre parent d'identificateur i.

Génération de la table de suffixes Une table de suffixes pour toutes les chaînes de sous-arbres frères consécutifs est ensuite calculée. Deux sous-arbres sont considérés comme égaux

ssi ils appartiennent à la même classe d'équivalence selon la famille d'abstraction f considérée. On s'aide à cet effet d'un index liant chaque sous-arbre à sa classe d'équivalence.

Obtention des farmax La table de suffixes obtenue, nous pouvons calculer le graphe des farmax par l'intermédiaire de l'arbre des intervalles selon la méthode décrite en 6.4.4. Ce graphe peut être filtré en éliminant les facteurs ne comprenant aucune occurrence extraite de l'arbre requête. Ainsi, si l'arbre $a(e, c, d, b, f)$ est recherché sur la base constituée des arbres $a(b, c, d, e)$ et $a(f, b, c, d)$, les chaînes de sous-arbre frères consécutifs communs *cd*, *bcd* et *bcd* sont prises en comptes pour la calcul du graphe de farmax $cd \longrightarrow bcd$ (les nœuds correspondant à des chaînes de longueur inférieure à 2 sont ignorés). On notera que le farmax *bcd* ne comprenant aucune occurrence sur l'arbre requête peut être supprimé : seul le farmax *cd* subsiste.

Farmax avec relation de parenté Nous pouvons ajouter au graphe de farmax de chaînes de fratrie une information de lien de parenté lorsqu'un facteur de fratrie est l'enfant d'un autre facteur. Concrètement, pour une occurrence de facteurs de fratrie $a_1a_2 \cdots a_n$ au sein d'un facteur répété α, nous cherchons s'il existe une occurrence de facteur de fratrie $b_1b_2 \cdots b_n$ au sein d'un facteur répété β avec b_i parent de $a_1, a_2, \cdots, a_n$. Dans l'affirmative un lien parent de α vers β est créé. Si la cardinalité de α est égale à β, le facteur répété α peut être supprimé car toutes ses occurrences sont par relation de parenté comprises dans β.

À titre d'illustration, considérons l'arbre requête $A = a(a(a, b), b(b, a), a(b, a))$ et l'arbre de la base $B = b(a(a, b), b(b, a))$. Nous relevons les classes d'équivalences suivantes pour a, b, $a(a, b)$ et $b(b, a)$. Le farmax des chaînes de fratrie comporte les facteurs ab et ba ainsi que les facteurs de longueur 1 $a(a, b)$ et $b(b, a)$ comprenant chacun deux occurrences. ab peut être lié au parent $a(a, b)$: ces deux facteurs ayant le même cardinal d'occurrences, ab peut être supprimé. ba peut être lié au parent $b(b, a)$ comprenant moins d'occurrences : ba ne peut être supprimé.

Support d'opérations d'abstraction

Problématique La recherche de chaînes de sous-arbres frères ne pose pas de difficulté spécifique avec l'usage d'une famille d'abtraction f se limitant à des opérations d'abstraction sur les types des nœuds, la structure de l'arbre n'étant pas impactée. Nous discutons maintenant des opérations non-neutres sur la structure telle que la suppression d'un sous-arbre (décrite en 9.2.4) et la suppression d'une racine de sous-arbre (décrite en 9.2.4). La non-considération de ces opérations a pour conséquence de fractionner des correspondances qui n'auraient pas lieu de l'être. Par exemple les deux sous-arbres $A = a(b, c(b, a), a)$ et $B = d(b, a)$ avec un profil d'abstraction supprimant les sous-arbres de racine c voient deux correspondances reportées sur leur premier et dernier fils de profondeur 2 alors qu'une correspondance complète entre $b, c(b, a), a$ et b, a pourrait être reportée.

Extension de la consécutivité des nœuds par jointures Pour éviter le fractionnement de correspondances en présence d'opérations d'abstraction sensibles à la structure, nous mémorisons parallèlement à l'indexation, selon le profil, des jointures indiquant lorsque deux nœuds deviennent consécutifs suite à une opération d'abstraction. Pour l'exemple précédent, sur A une jointure est créée entre le sous-arbre b de rang 1 et le sous-arbre a de rang 3. Ainsi dans le cas présent, pour une opération de suppression de sous-arbre, la jointure lie des sous-arbres de même parent mais non consécutifs. Lorsqu'une racine de sous-arbre est supprimée,

ses enfants remontent d'un niveau : deux jointures sont créées pour relier le frère gauche de la racine supprimée avec le premier enfant de la racine et le dernier enfant de la racine avec le frère droit de la racine. La condition de consécutivité de deux nœuds est modifiée pour prendre en compte les jointures : deux nœuds sont consécutifs ssi ils partagent le même parent et sont de rang consécutif ou sont liés par une jointure.

Commutativité des sous-arbres d'une fratrie Certaines abstractions peuvent introduire une normalisation de l'ordre de sous-arbres enfants susceptibles d'être commutatifs. Dans cette situation, nous pouvons également normaliser l'ordre des sous-arbres frères consécutifs obtenus après inventaire. La recherche de chaînes de sous-arbres frères consécutifs répétés alors que ceux-ci sont commutatifs ne présente cependant pas un intérêt important dans la mesure où ces éléments (typiquement des membres de classes ou unité de compilation) sont généralement mutuellement indépendants et leur suite peuvent faire l'objet, outre d'opérations de transposition, d'opérations d'insertion et de suppression.

10.4.5 Quantification de l'exactitude de paires de chaînes d'arbres correspondantes

Chaque facteur répété de sous-arbres frères consécutifs selon une famille d'abstraction f obtenu par la méthode précédemment décrite contient un ensemble d'occurrences de chaînes de sous-arbres consécutifs égales selon f. Si f correspond au profil d'abstraction nul, l'égalité sur l'arbre de syntaxe original utilisé est exacte. Si un certain niveau d'abstraction est utilisé pour f, les chaînes de sous-arbres du facteur répété peuvent présenter des différences à des niveaux d'abstraction plus faibles. Nous proposons d'introduire une métrique d'exactitude $\mathcal{E}(u, v)$ afin de quantifier la similarité entre deux chaînes u et v de sous-arbres d'un même facteur répété.

Distance d'édition sur les arbres

Une distance d'édition peut être calculée entre les occurrences de chaînes de sous-arbres frères des facteurs répétés afin de quantifier leur exactitude. Nous utilisons à cet effet trois types d'opérations d'édition élémentaires symétriques, chacune associée à un coût spécifique :

1. La suppression $\delta(A)$ de la racine a (ou son ajout) de l'arbre $A = a(C_1, \cdots, C_l)$ de coût $C_\delta(a)$.
2. La suppression $\Delta(A)$ du sous-arbre complet A (ou son ajout) de coût $C_\Delta(A)$. Typiquement, le coût $C_\Delta(A)$ de suppression de A est égal à la somme des coûts $C_\delta(a[k])$ de suppression de chaque des nœuds $a[k]$ de A.
3. Le changement d'un type de nœud de a en a' $\kappa(a, a')$ (ou symétriquement de a' en a) de coût $C_\kappa(a, a') = C_\kappa(a', a)$. Nous choisissons C_κ tel que $C_\kappa(a, a') < C_\delta(a) + C_\delta(a')$: il est plus avantageux de changer le type de nœud que de le supprimer pour en ajouter un du nouveau type.

Pour deux occurrences $A = A_1 A_2 \cdots A_m$ et $B = B_1 B_2 \cdots B_m$ d'un facteur répété de sous-arbres frères, avec $A_1 = B_1, A_2 = B_2, \cdots, A_m = B_m$ selon la famille f utilisée, nous cherchons à calculer la distance d'édition minimale $D(A, B)$. Tout d'abord, nous notons que nous pouvons trouver entre deux sous-arbres enfants T_i et T_{i+1} de A ou B des opérations de suppression

intersticielles de sous-arbre complet (si T_i et T_{i+1} sont liés par une jointure et ont le même parent). Des opérations de suppression de racine de sous-arbre sont caractérisés par la présence d'une sous-chaînes de nœuds $T_i \cdots T_j$ ayant le même parent, différent du parent des nœuds environnants T_{i-1} et T_{j+1}.

Dans un premier temps, nous calculons le coût associé aux opérations de suppression intersticielles de sous-arbres ou racines entre A et B. Les fratries ou sous-arbres intersticiels entre A_i, A_{i+1} et B_i, B_{i+1} sont comparées. Si seul l'interstice sur A ou B comportent des éléments ignorés par la famille, le calcul de la distance d'édition est immédiat et correspond à la création (ou destruction) de ces éléments. Si les deux interstices sur A et B sont occupés par des éléments, ceux-ci sont comparés après désérialisation par une méthode d'alignement de forêt d'arbres (cf 5.5).

Dans un second temps, nous calculons récursivement les distances des sous-arbres enfants explicites $D(A_i, B_i)$.

Pour le calcul récursif de la distance d'édition sur les sous-arbres, nous comparons les types de racine afin d'éventuellement sommer un coût de changement de type. Pour les sous-arbres enfants, deux situations peuvent être rencontrées :

- Les sous-arbres enfants sont indexés selon la famille : nous pouvons donc déduire les opérations implicites de suppression de sous-arbre et de racine en utilisant les informations de jointure et la connaissance des rangs des sous-arbres enfants par désérialisation paresseuse. Nous pouvons ensuite procéder comme indiqué précédemment pour le calcul de la distance d'édition sur une chaîne de sous-arbres.
- Les sous-arbres enfants ne sont pas indexés selon la famille et aucune information de jointure n'est disponible. Ceci peut survenir, par exemple, pour des sous-arbres de petite taille. Il est alors nécessaire de désérialiser les sous-arbres afin de les comparer selon une méthode d'alignement d'arbre décrite antérieurement en 5.5.

Exactitude normalisée La distance d'édition $D(\alpha, \beta)$ entre les arbres α et β est bornée par $C_\delta(\alpha) + C_\delta(\beta)$ correspondant au coût de destruction de α et de construction complète de β. Nous obtenons ainsi une métrique d'exactitude normalisée : $\mathcal{E}(\alpha, \beta) = 1 - \frac{D(\alpha,\beta)}{C_\delta(\alpha)+C_\delta(\beta)}$.

Exemple

Nous considérons comme exemple deux fonctions de calcul de nombre de Fibonacci déjà évoquées en figure 4.4 afin d'illustrer différentes méthodes d'obfuscation : la fonction originale et celle avec réécriture d'expressions. Nous utilisons un profil d'abstraction adapté à cette obfuscation : les petits sous-arbres de trois nœuds ou moins qui ne sont pas des instructions sont abstraits (remplacés par le nœud ζ), les commentaires sont ignorés et les structures de boucle sont supprimées. Nous en déduisons donc, en figure 10.9, les deux chaînes f_1 (fonction originale) et f_2 (fonction avec réécriture d'expression) de sous-arbres instructions frères similaires selon la famille d'abstraction, ayant pour parent le bloc d'instructions de la fonction.

f_1	f_2	Abstraction
commentaire		ignoré
int k = l ;	int k = l ;	decl
int l = 1 ;	int l = 1 ;	decl
int m = 0 ;	int m = 0 ;	decl
commentaire		ignoré
if (n == 1) return k	if (n == 1) return k*1 ;	if(ζ, return ζ)
if (n == 2) return l	if (n == 2) return l+0	if(ζ, return ζ)
for	*for*	racine supprimée
int i = 3 ;	int i = 3 ;	decl
i < n ;	i < n ;	cond
i++ ;	i += 1 ;	aff
m = k + l	m = k + 2*l - l ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta + \zeta - \zeta)$
k = l	k = l/1 + 0 ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta + \zeta)$
l = m	l = m * m / (m*1) ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta/\zeta)$
return m	return pgcd(m, m) ;	return ζ

FIG. 10.9 – Abstractions de deux versions de fonctions de calcul de nombre de Fibonacci

Nous constatons que l'abstraction des sous-arbres permet de confondre les instructions conditionnelles initiales de f_1 et f_2, cependant la réécriture trop étendue des expressions opérées pour les affectations de la boucle ne permet pas de les classer dans une même classe d'équivalence. Nous obtenons ainsi pour plus grand facteur répété sur f_1 et f_2 (récupérable par la table des LCP) la sous-chaîne suivante :

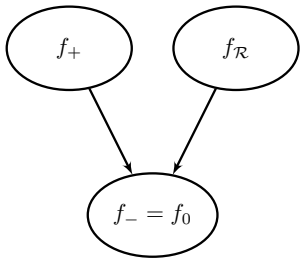
decl	decl	decl	if(ζ, return ζ)	if(ζ, return ζ)	decl	cond	aff

Nous calculons ensuite la distance d'édition $D(A, B)$ entre les deux occurrences A de f_1 et B de f_2 du plus long facteur répété. Celle-ci est composée des coûts de suppression des deux commentaires, ainsi que de la suppression de la racine de l'arbre de boucle *for*. Nous devons également comparer les sous-arbres abstraits par ζ : ainsi pour les instructions conditionnelles de retour k est transformé en $k * 1$ et l en $l + 0$, deux transformations nécessitant l'ajout d'une racine (opérateur binaire) et d'une opérande. Si toutes les opérations élémentaires d'édition ont un coût unitaire, $D(A, B) = 9$. En considérant $|A| = 32$ et $|B| = 35$, l'exactitude est estimée à $\mathcal{E}(A, B) = \frac{58}{67} \sim 0,87$.

10.5 Évaluation de quelques familles d'abstraction

Nous étudions expérimentalement des profils d'abstraction typiques à l'aide de graphes de famille de classes d'équivalence à trois nœuds. Une première famille de classe d'équivalence f_+ utilise le profil d'abstraction étudié tandis qu'une famille $f_\mathcal{R}$ est utilisée pour raffiner cette classe et produire depuis f_+ la famille $f_\mathcal{A} \cup f_\mathcal{R} = f_-$ de faible niveau d'abstraction.

Les cas étudiés de graphes de familles de classes d'équivalence sont introduits dans le tableau de la figure 10.10. Nous supposerons pour la suite que le niveau de plus faible abstraction des



Désignation	f_+
primTypeAbstr	f_0 avec abstraction des types primitifs
blockAbstr	f_0 avec abstraction des corps de méthodes et commutativité des membres
subAbstr4	f_0 avec abstraction des arbres de moins de 5 nœuds
nodeAbstr	f_0 avec abstraction de tous les types de nœuds
structAbstr	f_0 avec abstraction de la structure

FIG. 10.10 – Cas étudiés de graphes de familles de classes d'équivalence

arbres de syntaxe, noté f_0 prend en considération tous les types de nœuds et leurs propriétés sauf certains nœuds et sous-arbres sémantiquement non significatifs (commentaires, instructions d'importation et modificateurs) et réalise une abstraction des identificateurs (types non-primitifs et noms de variables) et constantes littérales ainsi qu'une normalisation de l'ordre des sous-arbres enfants de classes et des opérandes d'opérateurs infixes. Les tests sont réalisés sur des sous-arbres de volume d'au-moins 30 nœuds afin d'éviter de relever des similarités élémentaires.

Pour chacun des cas étudiés, nous supposons que le niveau d'abstraction f_- définit l'oracle de pertinence des clones : aucun couple de clones des classes d'équivalence de f_- n'est faux-positif.

La figure 10.11 présente des propriétés des classes d'équivalence des familles f_+ selon le nombre de sous-classes d'équivalence dont elles sont une généralisation. Deux projets développés en langage Java sont analysés : le paquetage de visualisation et d'édition de Javadoc pour l'éditeur Netbeans (environ 19K lignes de code dans 101 fichiers) ainsi que l'ensemble des paquetages spécifiques à la plate-forme de recherche de similarité Plade (environ 82K lignes de code dans 928 fichiers), présentés en annexe B.

Famille *primTypeAbstr* L'existence de classes à sous-classes multiples pour la famille abstrayant les types primitifs met en relief l'absence d'un mécanisme de généricité de code paramétré par des types primitifs en Java. La duplication de code pour différents types primitifs est ainsi notable dans l'API JDK de Java (méthodes de tri, de gestion de buffers dupliquées). Ce phénomène est cependant peu présent dans les projets étudiés (12 classes à 2 ou 3 sous-classes pour Plade, une pour Netbeans-Javadoc). Les clones différents par leur types primitifs sont de faible volume et concernent principalement pour Plade des constructeurs avec instructions répétées d'affectation ainsi que d'autres morceaux idiomatiques tel que celui présenté en figure 10.12 pour Netbeans-Javadoc présent en 18 exemplaires en 3 variantes de types (*boolean*, *long* et *int*).

Famille *blockAbstr* La famille *blockAbstr* permet de regrouper les méthodes de même signature ainsi que les classes équivalentes par les signatures de méthodes qu'elles contiennent. En particulier, les classes partageant un même ancêtre hiérarchique sans ajout de méthode

Profil ↓ / k →	1	2	3	4	5	[6..10]	[11..20]	> 20	Max
primTypeAbstr	209	0	1						3
	324	10	2						3
blockAbstr	8	24	9	1	4	9	3	2	55
	53	109	49	26	12	21	8	16	83
subAbstr4	194	22	2	3	0	0	0	0	4
	299	114	13	5	4	8	5	1	24
nodeAbstr	209	2	1						3
	322	22	2						3
structAbstr	211	1	2						3
	325	17	0						2

FIG. 10.11 – Nombre de classes de f_+ comportant k sous-classes de f_- pour chaque graphe de familles sur les projets Netbeans-Javadoc (1re ligne) et Plade (2e ligne)

```
109     public void setUse (boolean b) {
110         boolean old = use;
            use = b;
            firePropertyChange("use", new Boolean(old), new Boolean(use)); }
```

(a) StdDocletSettingsService

```
71      public void setMembers( long l ) {
            long old = members;
            members = l;
            firePropertyChange("members", new Long(old), new Long(members)); }
```

(b) ExternalJavadocSettingsService

FIG. 10.12 – Deux membres d'une même classe de f_+ = *primTypeAbstr* mais de sous-classes différentes de f_0 extraits de Netbeans-Javadoc

Cas	Classes concernées
Clones pertinents de code vivant factorisable	7
Clones pertinents avec un exemplaire de code mort	2
Clones idiomatiques difficilement factorisables	6
Clones non-pertinents	5

FIG. 10.13 – Évaluation de 20 classes de clones utilisant *subAbstr4*

additionnelle partagent la même valeur de hachage. Des classes de même valeur sans lien de parenté pourrait potentiellement bénéficier de la spécification d'un ancêtre commun. Concernant les méthodes de même signature, les plus nombreuses sont de prototype `void ()` pour Netbeans-Javadoc et Plade. Les types non-primitifs étant confondus, seul le nombre d'arguments est considéré : ce type d'abstraction n'a donc pas d'utilité réelle pour la recherche de schéma de conception. La prise en compte du type non-primitif déclaré ainsi que de la hiérarchie des types afin d'établir une relation de compatibilité entre signatures s'avère alors incontournable.

Famille *subAbstr4* La famille *subAbstr4* permet de regrouper des sous-arbres présentant des différences de nœuds à forte profondeur. Les sous-arbres comportant 4 nœuds ou moins sont abstraits ce qui induit une résistance à la réécriture simple d'expressions. Un effectif important de classes à deux sous-classes est présent pour Plade et dans une moindre mesure pour Netbeans-Javadoc. Pour Plade, la classe aux sous-classes les plus nombreuses concerne des interfaces comportant une ou plusieurs méthodes avec des paramètres abstraits car représentés par des sous-arbres de moins de 4 nœuds. Les classes à deux sous-classes comprenant un effectif réduit de clones s'avèrent les plus intéressantes dans une optique de factorisation de code comme le couple de clones de Plade présenté en figure 10.14. Nous pouvons être confronté également à des familles regroupant des clones apparemment faux-positifs de squelette structurel identique, avec quelquefois une similarité conceptuelle (pour 10.15 la conversion d'un tableau d'objets en un autre) sans possibilité de factorisation. Concernant Plade, nous avons évalué les clones de 20 classes parmi les 114 classes comportant 2 sous-classes avec un jugement subjectif humain sur la pertinence de ces familles. Le résultat de cette évaluation est résumé par le tableau en figure 10.13 (chaque classe contenant deux sous-classes d'un exemplaire de code). L'abstraction de petits sous-arbres montre son intérêt pour trouver des clones pertinents (9/20). La fréquence des clones idiomatiques trouvés pourrait être réduite par un filtrage supprimant les clones les plus fréquemment rencontrés sur une base importante de projets.

Famille *nodeAbstr* Concernant l'abstraction totale des types de nœuds (famille *nodeAbstr*), nous constatons que le nombre de classes à effectif élevé de sous-classes est particulièrement faible. Par exemple pour Netbeans-Javadoc, seules trois classes comportent plus d'une sous-classe. Les clones concernés diffèrent uniquement par des types utilisés (*primTypeAbstr* aurait pu suffire pour les regrouper) ou par l'emploi de littéraux à la place de types. Nous pouvons en conclure, pour les exemples considérés, que ne considérer que la structure améliore le rappel et se révèle peu générateur de faux-positifs. En effet, l'arité d'un nœud ainsi que l'arité des nœuds de son sous-arbre est assez prédictif de son type. Toutefois, la seule considération de la structure s'avère insuffisante pour une distinction fine des sous-arbres de faible volume.

```
66  if (mode.equals(OpeningMode.CREATE))
    {
     File dir = new File(path);
     boolean created = dir.mkdir();
70   if (! created) throw new IOException("Cannot␣create␣the␣directory␣'" + dir + "'");
    }
```

(a) DefaultTypeRepository

```
54  if (mode.equals(OpeningMode.CREATE))
55  {
     File f = new File(path);
     boolean created = f.mkdir();
     if (! created)
      throw new IOException("Cannot␣create␣directory␣" + path);
60  }
```

(b) LiveParsingRepresentationRepository

FIG. 10.14 – Deux exemplaires de code de Plade dans la même classe d'abstraction *subAbstr4* différant par une expression

```
108   String[] components = str.split(DEFAULT_ARRAY_SEPARATOR);
      Object[] data = new Object[components.length];
110   for (int k=0; k < components.length; k++)
       data[k] = fromString(components[k], cl.getComponentType(), dependencies);
```

(a) DefaultStringConverter

```
43  public Match[] getSelectedMatches()
    {
45   int[] selection = graphViewer.getSelection();
     Match[] matches = new Match[selection.length];
     for (int k=0; k < matches.length; k++)
      matches[k] = graphViewer.getGraph().getDataNode(selection[k]);
     return matches;
50  }
```

(b) MatchGraphViewer

FIG. 10.15 – Morceaux de code réunis par *subAbstr4* mais difficilement factorisables

Profil d'abstraction	Volume de couverture global
primTypeAbstr	39 790 85 928
blockAbstr	72 051 236 518
subAbstr4	47 330 158 530
nodeAbstr	39 928 99 355
structAbstr	40 440 86 142

FIG. 10.16 – Volume de couverture global pour les différents profils d'abstraction

Famille *structAbstr* La famille *structAbstr* réalise une abstraction de la structure des sous-arbres hachés en prenant en considération uniquement le types des nœuds : cela revient à considérer le sac (ensemble avec multiplicité) des nœuds d'un sous-arbre. Pour des sous-arbres de volume non négligeable, cette approche apparaît d'une bonne précision. Sur le projet Netbeans-Javadoc, ce profil d'abstraction, tout comme le précédent *nodeAbstr* se révèle pratiquement équivalent à f_0. Pour Plade, plus de classes multi-composées sont relevées. Après analyse exhaustive, l'ensemble de ces classes multi-composées apparaît comme pertinente même si dans certains cas l'intérêt d'une factorisation est discutable.

Rappel Concernant le rappel, la figure 10.16 peut servir de base comparative des différents profils d'abstraction. Le volume de couverture global des projets étudiés y est exposé ; ce volume est défini par la somme des volumes d'exemplaires de clone participant à une classe d'équivalence comportant au moins deux exemplaires de code. En faisant exception du cas particulier du profil *blockAbstr*, même si la précision exacte des exemplaires supplémentaires obtenus est inconnue, *subAbstr4* semble proposer le meilleur rappel. *nodeAbstr* et *structAbstr* présentent des couvertures disparates pour les deux projets et soulignent des caractéristiques différentes des clones de chacun des projets ; Plade présente plus de clones structurels creux que Netbeans-Javadoc. Ce dernier projet manipule intensément la bibliothèque d'affichage graphique Swing ce qui est susceptible de générer de nombreux clones d'un degré d'idiomaticité plus ou moins fort. On prendra pour exemple (figure 10.17) deux implantations de constructeur de composant graphique différant par des transpositions d'instructions rassemblées dans une même classe par *structAbstr* (un troisième exemplaire non présenté concerne le constructeur de `StandardTagPanel`). *structAbstr* permet également de regrouper des exemplaires dont des instructions auraient été remontées ou descendues dans l'arbre de syntaxe (changement de parent).

Quelques statistiques sur les classes À titre informatif afin de mieux appréhender la nature des clones des deux projets réunis par le profil d'abstraction minimal f_0, la figure 10.18 présente quelques statistiques de ses classes d'équivalence. Elle fait apparaître que les clones de Plade sont de volume moyen plus important, sans doute lié à la coexistence de code actif recopié depuis du code devenu mort et en instance de suppression. D'autre part les clones de Netbeans-Javadoc sont regroupés dans des classes d'effectif plus important confirmant l'hypothèse de clonage idiomatique lié à l'utilisation de Swing. Le ratio de couverture global par des clones

```
31      public ParamTagPanel( final JavaDocEditorPanel editorPanel ) {
            super( editorPanel );
            initComponents();
            initAccessibility();
35          jLabel2.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(ParamTagPanel.class).getString("
                CTL_ParamTagPanel.jLabel2.text_Mnemonic").charAt(0)); // NOI18N
            jLabel1.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(ParamTagPanel.class).getString("
                CTL_ParamTagPanel.jLabel1.text_Mnemonic").charAt(0)); // NOI18N
            addHTMLComponent( descriptionTextArea );
            editorPanel.registerComponent( descriptionTextArea );
            parameterComboBox.getEditor().getEditorComponent().addFocusListener(
40          new java.awt.event.FocusAdapter() {
                public void focusLost(java.awt.event.FocusEvent evt) {
                    commitTagChange(); }
            }); }
```

(a) ParamTagPanel

```
67      public ThrowsTagPanel( JavaDocEditorPanel editorPanel ) {
            super( editorPanel );
            initComponents ();
70          jLabel2.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(StandardTagPanel.class).getString("
                CTL_ThrowsTagPanel.jLabel2.text_Mnemonic").charAt(0)); // NOI18N
            jLabel1.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(StandardTagPanel.class).getString("
                CTL_ThrowsTagPanel.jLabel1.text_Mnemonic").charAt(0)); // NOI18N
            editorPanel.registerComponent( descriptionTextArea );
            addHTMLComponent( descriptionTextArea );
            exceptionComboBox.getEditor().getEditorComponent().addFocusListener(
75          new java.awt.event.FocusAdapter () {
                public void focusLost (java.awt.event.FocusEvent evt) {
                    commitTagChange(); }
            });
            initAccessibility(); }
```

(b) ThrowsTagPanel

FIG. 10.17 – Deux constructeurs de composants graphiques avec transposition d'instructions de Netbeans-Javadoc réunis par *structAbstr* (volume des clones : 91 nœuds)

Propriété	Min	Q1	Médiane	Moyenne	Q3	Max
Volume moyen des classes ($\mathcal{V}$)	29,50 17,86	36,00 35,00	49,00 51,00	73,14 120,0	85,38 93,0	525,0 3 434
Cardinalité des classes (c)	2,000 2,000	2,000 2,000	2,000 2,000	3,024 2,151	3,000 2,000	48,00 7,000
Couverture des classes ($\mathcal{C} = \mathcal{V}c$)	59,00 48,00	90,00 74,00	125,0 114,0	187,5 246,5	214,0 192,0	1 584,0 6 868,0
Paires dans chaque classe ($\frac{c(c-1)}{2}$)	1,000 1,000	1,000 1,000	1,000 1,000	13,58 1,39	3,000 1,0	1 128 21,00
Nombre de classes	212 344					
Volume de couverture global ($\mathcal{C}_{\text{moyenne}} * C$)	39 757 84 790					

FIG. 10.18 – Valeurs aux quartiles (avec moyenne) pour les propriétés des classes de f_0 (volume, cardinalité des classes et proximité des exemplaires de clone) pour les projets Netbeans-Javadoc (1re ligne) et Plade (2e ligne)

est deux fois plus faible pour Plade que pour Netbeans-Javadoc. Il faut toutefois nuancer les résultats obtenus en rappelant que seuls les sous-arbres unitaires de l'arbre de syntaxe de plus de 30 nœuds sont considérés : le rappel pourraît être favorisé par l'utilisation d'un seuil plus faible pour collection des correspondances servant de germes à une consolidation puis en ignorant ensuite les germes de volume trop faibles non consolidés. Dans cette optique, nous tentons d'étudier dans l'annexe suivante quelques expérimentations de consolidation de germes.

10.6 Limitations de l'indexation par profil

La méthode d'indexation par famille de profils permet de constituer des bases d'empreintes permettant de déterminer des k-correspondances approchées entre arbres de la base et arbre requête. Cette méthode ne permet cependant que de gérer certains types d'abstraction courants bien délimités induisant un niveau de normalisation de l'arbre indexé. S'il est possible d'ignorer certains mots-clés méta-informatifs, de supprimer des commentaires, des racines de structures ou abstraire types ou commentaires, voire d'ignorer l'ordre de sous-arbres, il est impossible de gérer des opérations d'édition arbitraires consistant à ajouter ou supprimer des sous-arbres quelconques. Nous présentons au chapitre suivant une heuristique d'extension se basant sur un ensemble de k-correspondances trouvées sur une base d'indexation. Les 2-correspondances (paires de clones) en sont extraites et nous cherchons à les consolider suivant un critère de proximité dans l'arbre de syntaxe. Nous obtenons ainsi des paires d'arbres présentant une duplication approchée avec insertion, suppression ou transposition d'arbres sur une fratrie ou entre cousins de niveau hétérogène.

11

Consolidation de correspondances

Sommaire

11.1 De l'intérêt de la consolidation de correspondances

La recherche de facteurs répétés de sous-arbres frères telle que présentée dans le chapitre précédent montre en pratique ses limites, même avec l'usage de certains profils d'abstraction. Si certains profils permettent de gérer des cas d'obfuscation comme la réécriture d'expressions (abstraction des petits sous-arbres) ou la modification de structures de contrôle (suppression des nœuds représentant ces structures), d'autres opérations d'éditions conduisent à l'obtention de valeurs de hachage différentes pour un sous-arbre et son clone obfusqué.

11.1.1 Ajout, suppression ou transposition de code

En particulier, l'ajout ou la suppression de code inutile n'est pas systématiquement décelable lors d'une étape de normalisation de l'arbre de syntaxe. Il en est de même pour certaines opérations de transposition de code. Ces opérations conduisent à l'ajout, la suppression ou le déplacement de sous-arbres de volume plus ou moins important.

Ajout/suppression de code Ainsi, par exemple, pour une fratrie de sous-arbres $C_1C_2\cdots C_n$ et son homologue clone $C'_1C'_2\cdots C'_{i-1}DC'_{i+1}\cdots C'_n$ auquel est ajouté un sous-arbre D, deux facteurs répétés distincts $\{C_1\cdots C_{i-1}, C'_1\cdots C'_{i-1}\}$ et $\{C_{i+1}\cdots C_n, C'_{i+1}\cdots C'_n\}$ seront relevés. Il apparaîtrait plus légitime, pour un évaluateur humain, de consolider ces deux correspondances en une seule en mentionnant l'ajout du sous-arbre D si celui-ci est de volume négligeable comparé à celui de l'ensemble de la fratrie.

Transposition de code Considérons maintenant le cas de la structure conditionnelle basée sur l'arbre de syntaxe $A = \text{si}(\text{cond}, \text{alors}, \text{sinon})$. Une version obfusquée A' est réalisée en négativant la condition et intervertissant les sous-arbres alors et sinon : $A' = \text{si}(\neg\text{cond}, \text{sinon}, \text{alors})$. Même l'utilisation d'un profil d'abstraction éliminant les structures conditionnelles ($\text{abstr}(A) = (\text{cond}, \text{alors}, \text{sinon}$, $\text{abstr}(A') = (\neg\text{cond}, \text{sinon}, \text{alors})$ ne permet pas de consolider les sous-arbres alors et sinon en une unique correspondance à cause de la transposition des deux sous-arbres et de la négation de la condition.

11.1.2 Factorisation ou développement de fonctions

Un ensemble d'instructions inter-dépendantes d'une fonction peut être externalisée dans une nouvelle fonction qui sera appelée depuis la fonction source. L'externalisation (ou factorisation) se matérialise sur l'arbre de syntaxe par la migration des instructions de la fonction source vers la fonction externalisée avec l'introduction de paramètres de fonction correspondant aux

variables locales utilisées dans la fonction source ; l'ajout d'instructions de retour est également probable.

L'opération réciproque de développement consiste à remplacer l'appel d'une fonction par le corps de ladite fonction. Le corps de la fonction nécessite généralement quelques adaptations au contexte local : changement de noms de variables locales, remplacement des instructions de retour par des affectations, remplacement des identificateurs des paramètres par leur valeur d'argument (cf figure 7.1)... Ces petites modifications ne permettent souvent pas d'obtenir une correspondance exacte entre le corps développé et le corps original : des fossés de non-correspondance sont présents. Il s'agit de regrouper ces correspondances séparées par des fossés en une unique correspondance.

11.1.3 Travaux antérieurs

Des méthodes d'extension de correspondances ont déjà été proposées pour le raccordement de facteurs de séquences de lexèmes : celles-ci ont été abordées brièvement en 5.4.3.

L'extension de correspondances sur des arbres de syntaxe est quant à elle un sujet qui été peu exploré. On pourra citer la méthode de propagation de correspondances utilisée par CloneDr [91]. Celle-ci consiste, lorsque deux arbres (ou chaînes d'arbres frères) ont été reportés comme similaires, à tenter de prolonger systématiquement la correspondance à leurs parents. Les sous-arbres ayant pour racine respective chacun des deux parents sont donc comparés en utilisant une méthode d'alignement sur arbres comme celles décrites en section 5.5. Si cette approche permet la détection en tant que clones de deux arbres possédant au moins un sous-arbre identique avec des modifications sur les autres sous-arbres frères, plusieurs niveaux de propagation pourraient être nécessaires pour permettre la mise en valeur de certains clones. D'autre part, le coût de détermination de distance d'édition entre deux arbres peut devenir prohibitif pour de grands sous-arbres.

Cobena [42] a également utilisé des méthodes de propagation à des nœuds parent pour la recherche de paires de sous-arbres correspondants dans des arbres de documents XML. La problématique demeure néanmoins assez différente de la notre car consistant à déterminer un delta entre différentes versions d'un même document XML, chaque sous-arbre ne pouvant correspondre au plus qu'avec un seul sous-arbre de l'autre version. Dans le cadre de la recherche de correspondances sur des sous-arbres de syntaxe provenant d'analyse de code source, l'utilisation de clones chevauchants n'est pas sans intérêt, une même zone de code pouvant être dupliquée au sein d'un même projet, voire d'une même unité de compilation.

11.1.4 Aperçu de la méthode de consolidation

Nous proposons dans ce chapitre une heuristique de consolidation de paires de correspondances en s'aidant de l'arbre de syntaxe des unités étudiées. Étant donné un arbre de syntaxe requête et une base d'arbres indexés, nous recherchons dans un premier temps les classes d'équivalence de sous-arbres, selon un profil d'abstraction donné, contenant au moins un exemplaire d'un sous-arbre de l'arbre requête. Nous en déduisons l'ensemble des paires de sous-arbres similaires dont un exemplaire provient de l'arbre requête et l'autre d'un arbre de la base. Ces

paires constituent des germes qui serviront de base à l'extension des correspondances. L'heuristique de consolidation de correspondances utilise alors deux étapes principales. L'étape de fusion consiste à regrouper les correspondances dont l'exemplaire sur l'arbre requête est proche. La seconde, l'étape de propagation, consiste à étendre une correspondance couvrant un volume important d'une fratrie de sous-arbres d'un arbre de la base à son sur-arbre parent. Ces deux étapes sont menées itérativement depuis les germes de forte profondeur des arbres de la base vers les germes de faible profondeur : elles permettent d'agglutiner des germes au sein de correspondances de volume croissant. Les étapes de fusion et propagation permettent non seulement la consolidation de sous-arbres frères, ce qui était déjà proposé d'une certaine manière par [91] ou en 10.4.4 lorsque les sous-arbres étaient consécutifs dans la fratrie, mais également le regroupement de sous-arbres cousins plus ou moins éloignés.

Nous nous intéressons ensuite à la consolidation des correspondances à travers les graphes d'appels afin de gérer des opérations d'obfuscation par développement ou factorisation de fonctions. Ainsi, si une correspondance occupe un volume conséquent d'une fonction, celle-ci pourra être propagée à l'ensemble de la fonction et par la suite à ses fonctions appelantes.

11.2 Extension à travers les arbres de syntaxe

11.2.1 Quelques définitions préalables

Définition 11.1. *(Correspondance élémentaire)* Une correspondance élémentaire (U, V) entre un arbre requête A et un arbre B de la base $\mathcal{B}$ met en relation une chaîne de sous-arbres (consécutifs) de la même fratrie de A (U) avec une chaîne de sous-arbres de la même fratrie de B (V). U et V sont égales selon un profil d'abstraction p donné.

Définition 11.2. *(Correspondance consolidée)* Une correspondance consolidée (U, V) entre un arbre requête A et un arbre B de la base $\mathcal{B}$ met en relation un ensemble de sous-arbres d'une même fratrie de A (U) avec un ensemble de sous-arbres d'une même fratrie de B (V). Cette correspondance est caractérisée par les propriétés suivantes :

- L'opération de consolidation ayant conduit à l'obtention de la correspondance.
- Les correspondances C (correspondance consolidée ou élémentaire) ayant servi de base à la création de la correspondance à l'aide de l'opération de consolidation.
- Le volume de U et de V ($\mathcal{V}(U)$ et $\mathcal{V}(V)$)[1].

Nous notons qu'il est possible, à partir d'une correspondance consolidée racine, d'en déduire un arbre de consolidation, chaque correspondance ayant pour correspondances enfants les correspondances C ayant été utilisées pour sa création. Les correspondances feuilles de l'arbre sont les correspondances élémentaires (ou germes).

Chaque correspondance (U, V) est, pour les besoins de l'algorithme, rattachée au nœud parent de V dans la base. Ceci nous permet de connaître les correspondances auxquelles participent les sous-arbres enfants d'un sous-arbre de la base, un sous-arbre enfant pouvant participer à plusieurs correspondances liées à des sous-arbres distincts de l'arbre requête mais présentant une similarité entre-eux.

[1]Dans un premier temps, à des fins de simplification, nous confondons le nombre de nœuds de l'arbre et son volume.

La connaissance des volumes des sous-arbres manipulés sur l'arbre de requête et chacun des sous-arbres de la base est nécessaire afin de tester les conditions des opérations de fusion et propagation. Les volumes de tous les sous-arbres de l'arbre requête sont calculables en temps linéaire si ils s'assimilent au nombre de nœuds. Ceux des sous-arbres de la base peuvent être enregistrés lors de leur indexation.

11.2.2 Récupération des paires de correspondances

Préalablement à l'extension, il est nécessaire de déterminer l'ensemble des correspondances élémentaires (germes) qui seront utilisées pour la constitution de correspondances consolidées. Ainsi par un parcours en largeur, nous examinons chacun des sous-arbres $A[t]$ de l'arbre requête A afin de déterminer s'il existe une classe d'équivalence sur la base avec le profil d'abstraction considéré correspondant à $A[t]$. On se reportera au chapitre 10 pour plus de détails sur cette opération. Nous obtenons pour chaque sous-arbre $A[t]$ l'ensemble c des sous-arbres de la classe d'équivalence correspondante sur la base. Nous notons que si un autre arbre β appartient à c, ses sous-arbres enfants appartiennent aux même classes d'équivalence que les enfants $A[t]_i$ de $A[t]$ sans toutefois être référencés explicitement dans ces classes.

Le processus de consolidation utilise des 2-correspondances et non des groupes de cardinalité supérieure à 2 : à partir d'une k-correspondance comportant k' composantes sur A ($k' \geq 1$), nous extrayons les $k'(k-k')$ 2-correspondances composées exactement d'une composante de A.

11.2.3 Fusion

Objectif

L'opération de fusion de correspondances consiste à regrouper plusieurs correspondances dont les composantes sur la base sont frères et dont les composantes sur l'arbre requête sont suffisamment proches. Son objectif principal vise à rassembler dans l'arbre requête, que l'on suppose être une version obfusquée d'une portion d'arbre de la base, des sous-arbres qui auraient été frères (dans la base) mais dispersés dans l'arbre requête. Cette situation peut être rencontrée, par exemple, dans un cas d'obfuscation où une portion de code serait extraite d'une structure de contrôle, et donc séparée de ses frères, pour être remontée dans l'arbre de syntaxe. Pour la fusion, nous utilisons une heuristique gloutonne avec validation.

Création d'une correspondance par fusion

Lorsque k correspondances $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$ sont fusionnées, une nouvelle correspondance (α, β) est créée. β est constituée de l'ensemble des composantes sur la base : $\{V_1, V_2, \cdots, V_k\}$. Nous distinguons deux types de fusion :

1. La fusion entre composantes frères et de même ordonnancement dans l'arbre requête et l'arbre de la base. Un ensemble de correspondances $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$ (triées par position de leur composante sur A) satisfait cette condition ssi pour tout j, U_j et U_{j+1} sont contigüs (chaînes ou arbres unitaires) ainsi que leur homologues sur la base V_j et V_{j+1}. Une nouvelle correspondance $(U_1U_2 \cdots U_k, V_1V_2 \cdots V_k)$ est alors créée.
2. La fusion entre composantes non-frères ou d'ordonnancement différent. Dans ce cas, pour les composantes sur l'arbre requête, nous recherchons leur plus petit ancêtre commun

(Lowest Common Ancestor : LCA) sur l'arbre requête A. Il s'agit concrètement de déterminer le plus petit sous-arbre $L = lca(\{U_1, U_2, \cdots, U_k\})$ de A contenant $U_1, U_2, \cdots, U_k$. Une nouvelle correspondance $(U = L, V = \{B_1, B_2, \cdots, B_k\})$ est alors créée issue de la fusion des correspondances. Pour pouvoir être fusionnées, les correspondances doivent concerner des composantes de la base (composantes frères) totalement distinctes[2].

Calcul du *LCA* Le calcul du LCA de deux nœuds s'effectue naïvement par récupération du premier nœud commun sur leurs branches ascendantes menant à la racine en temps linéaire en la profondeur cumulée des deux nœuds ($\Theta(h)$ pour un arbre de hauteur h). De nombreux calculs de LCA étant requis lors des opérations de fusion, un pré-traitement de l'arbre afin de pouvoir déterminer le LCA en $o(h)$ est envisageable. En utilisant la méthode de Berkman et Vishkin [5], l'arbre requête fait d'abord l'objet d'un parcours en profondeur. Nous obtenons alors la séquence des nœuds parcourus (deux fois pour chaque nœud) associés à leur profondeur dans l'arbre : trouver le LCA de deux nœuds équivaut alors à déterminer le nœud de plus faible profondeur entre les deux nœuds de la séquence de parcours. Ceci revient à trouver l'entier minimal sur un intervalle d'une suite d'entiers (*Range Minimum Query* : RMQ). Le problème du RMQ est résolvable en temps constant après un pré-traitement linéaire. Fisher et Heun proposent une implantation [9] nécessitant seulement $2n + o(n)$ bits pour le pré-traitement d'une suite de n entiers. Le LCA de k nœuds peut être calculé récursivement en déterminant le LCA de $k-1$ de ces nœuds avec le nœud restant : après pré-traitement, cette opération est réalisée en $O(k)$.

Problématique Nous souhaitons déterminer sur les κ 2-correspondances dont les composantes de base sont frères une partition de ces 2-correspondances dont chaque groupe représenterait une correspondance issue de la fusion de ses membres. Il s'agit de minimiser le nombre de groupes tout en optimisant le ratio de couverture du sous-arbre LCA de chaque groupe par ses composantes germes constituantes sur l'arbre requête. Nous notons ce ratio ρ ; en supposons des correspondances non-chevauchantes $(U_1, V_1), \cdots, (U_k, V_k)$ envisagées pour la fusion, nous avons $\rho = \frac{\mathcal{V}(U_1)+\cdots+\mathcal{V}(U_k)}{\mathcal{V}(\text{lca}(U_1,\cdots,U_k))}$. Les deux objectifs de minimisation des groupes et d'optimisation des ratios de couverture étant antagonistes, un compromis doit être réalisé : nous proposons à cet effet une heuristique de partition en deux étapes décrite ci-après.

Heuristique de sélection des correspondances à préfusionner Nous nous proposons de fusionner progressivement les κ correspondances par une heuristique gloutonne en commençant par les fusions de ratio de couverture ρ les plus élevés. Nous maintenons un tas comportant l'ensemble des 2-correspondances consolidées avec priorité aux correspondances de plus fort ratio de couverture ρ. Il est initialisé par les κ 2-correspondances germes frères sur un arbre de la base avec $\rho = 1$. À chaque itération, nous associons la correspondance c de valeur ρ maximale avec une autre correspondance du tas telle que la correspondance issue des deux précédentes soit de ratio de couverture ρ maximal. Ceci est accompli en calculant les ratios de couverture de c avec chacune des autres correspondance du tas. Les deux correspondances germes sont supprimées du tas et la nouvelle correspondance créée y est insérée. L'itération de ce processus $k-1$ fois conduirait à l'obtention d'une large correspondance issue de toutes les correspondances germes initiales et dont le LCA global représenterait un sous-arbre de l'arbre

[2]Il est possible qu'un nœud de la base soit en correspondance avec plusieurs nœuds de l'arbre requête : ceux-ci ne sont donc pas fusionnés en leur LCA

requête de volume potentiellement important. Le nombre de correspondances obtenues serait minimisé mais le ratio de couverture serait faible. Nous introduisons donc une condition de préfusion : deux correspondances peuvent être associées si la correspondance résultante est de ratio de couverture $\rho \geq t_m$ où t_m est un seuil fixé. La condition d'arrêt des itérations est caractérisée par l'inexistance de couple de correspondances du tas dont l'association serait de ratio de couverture $\rho \geq t_m$.

Validation des correspondances préfusionnées Chaque correspondance préfusionnée est de la forme $(\{U_1, U_2, \cdots\}, \{V_1, V_2, \cdots\})$ issue de la préfusion d'au moins deux correspondances (U_1, V_1) et (U_2, V_2). Nous avons $L = lca(U_1, U_2, \cdots)$ avec le ratio de couverture ρ associé. Si $\rho \geq t_M$, valeur seuil pour la fusion définitive (avec $t_M \geq t_m$), la correspondance préfusionnée est définitivement fusionnée. Si la correspondance préfusionnée ne satisfait pas la condition de fusion définitive, l'opération de préfusion la plus récente est annulée : nous obtenons les deux correspondances préalablement à la préfusion sur lesquelles nous testons la condition de fusion définitive. L'opération est récursivement répétée sur chaque correspondance issue d'une préfusion si celle-ci ne peut être fusionnée définitivement ou alors lorsqu'une correspondance simple non issue de préfusion est obtenue.

Intérêt de la préfusion L'utilisation d'un seuil de préfusion t_m plus faible que le seuil de fusion définitive t_M réside dans la possibilité de préfusionner dans l'arbre requête des structures de proximité importante dont le ratio de couverture par rapport au *lca* est trop faible pour atteindre t_M, mais qui pourraient, après plusieurs préfusions successives atteindre ce ratio. Considérons par exemple un sous-arbre $B = a(\alpha, \beta, \gamma)$ de la base avec $\mathcal{V}(\alpha) = \mathcal{V}(\beta) = \mathcal{V}(\gamma) = 1$. Imaginons une version obfusquée de cet arbre où chaque sous-arbre aurait été encapsulé dans une structure conditionnelle systématiquement exécutée (que nous noterons c) avec effet neutre sur le volume : $A = a(c(\alpha'), c(\beta'), c(\gamma'))$. Si $t_M > \frac{2}{3}$, la fusion directe des correspondances (α', α) et (β', β) de LCA B serait impossible ($\rho = \frac{2}{3} < t_M$) : cependant avec $t_m \leq \frac{2}{3}$, une préfusion serait possible. Après deux préfusions, nous obtenons $(A, \{\alpha, \beta, \gamma\})$ avec $\rho = 1$: la fusion peut être validée.

Choix du sur-arbre du *LCA* Afin de gérer certains cas d'obfuscation consistant à entourer un bloc de code par un ou plusieurs niveaux de structures inutiles (entourage par une structure conditionnelle invariablement vraie, une boucle d'itération unique, ...), plutôt que de choisir le LCA des composantes de l'arbre requête pour la fusion de correspondances, nous sélectionnons le parent direct ou indirect. Ainsi, si le ratio $\rho' = \frac{\mathcal{V}(L)}{\mathcal{V}(\mathcal{P}(L))}$ du volume de l'arbre L conservé par la fusion (initialement L est le LCA des composantes de l'arbre de requête) sur le volume de son sur-arbre parent est supérieur à un seuil $1 - \epsilon$, L est substitué par $\mathcal{P}(L)$. Le procédé est itéré jusqu'à ce que la condition de sélection du parent ne soit plus remplie.

11.2.4 Propagation

Une fois réalisées les fusions de correspondances associées à une fratrie de sous-arbres d'un arbre de la base se pose la question du devenir de ces correspondances fusionnées et de celles restantes n'ayant pas fait l'objet de fusion. Il peut être intéressant de les associer à leur grand parent sur l'arbre de la base afin de pouvoir les fusionner avec des correspondances tantes issues elle-mêmes de correspondances cousines. L'opération de propagation permet de *remonter* une correspondance dans l'arbre de syntaxe de la base.

Données : Correspondances à fusionner : $(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)$
Données : t_m : ratio minimal de couverture pour la préfusion
Données : $t_M > t_m$: ratio minimal de couverture pour la fusion
Résultat : Ensemble des correspondances fusionnées R

début
  Initialisation de la file de priorité des paires de correspondances candidates à la fusion ;
  $F \longleftarrow \emptyset$;
  pour $(U_i, V_i) \in$ *correspondances germes considérées pour la fusion* **faire**
    $F \longleftarrow F + (A_i, B_i)$;
  Itérations de préfusion ;
  $G \longleftarrow \emptyset$ *(Correspondances non-associables)* ;
  tant que $F \neq \emptyset$ **faire**
    $c \longleftarrow \max_\rho(F)$;
    Initialisation de la correspondance à associer ;
    $c' \longleftarrow \emptyset$;
    pour $d \in F / d \neq c$ **faire**
      si $\rho(\text{fusion}(c, d)) > \rho(\text{fusion}(c, c')) > t_m$ **alors**
        $c' \longleftarrow d$
    si $c' \neq \emptyset$ **alors**
      $F \longleftarrow F - c - c' + \text{fusion}(c, c') + G$;
      $G \longleftarrow \emptyset$;
    sinon
      $G \longleftarrow G + c$;
  Validation des préfusions ;
  $F' \longleftarrow \emptyset$;
  tant que $F \neq \emptyset$ **faire**
    $c \longleftarrow F[1]$;
    $F \longleftarrow F - c$;
    si $\rho(c) \geq t_M$ **alors**
      $F' \longleftarrow F' + c$;
    sinon
      $(c_1, c_2) \longleftarrow$ défusion(c) ;
      $F \longleftarrow F + c_1 + c_2$;
  retourne *les correspondances fusionnées* F'
fin

Algorithme 6 : Heuristique gloutonne de fusion de correspondances

L'opération de propagation sur une correspondance (U, V) associée à son nœud parent de la base $\mathcal{P}(V)$ transforme ladite correspondance en $(U, \mathcal{P}(V))$ désormais associée à $\mathcal{P}(\mathcal{P}(V))$.

Pour chaque correspondance (U, V) associée au sous-arbre B de la base, la propagation est décidée si le ratio $\rho'' = \frac{\mathcal{V}(U)}{\mathcal{V}(\mathcal{P}(V))}$ est supérieur à un seuil de propagation t_p. Nous notons que si B concerne des enfants de la racine d'un arbre de la base, la correspondance est remontée à la racine et associée à une sur-racine virtuelle : celle-ci ne peut plus être propagée dans l'arbre de la base.

11.2.5 Algorithme général

Nous présentons maintenant le squelette de l'algorithme général de consolidation à travers les arbres de syntaxes (algorithme 7). Initialement les correspondances élémentaires sont trouvées et associées au parent de leur composante sur la base. Chaque parent de la base ayant au moins une correspondance pour enfant est ajouté dans une file de priorité permettant d'obtenir le plus petit sous-arbre de la base (en terme de nombre de nœuds) dont les correspondances enfants n'ont pas encore fait l'objet de procédé de fusion puis de propagation. À chaque itération, un sous-arbre de la base est ainsi sélectionné. Nous appliquons le processus de fusion sur ses correspondances enfant. Chaque correspondance enfant créée ou subsistant après la fusion est considérée pour son éventuelle propagation d'un niveau dans l'arbre de la base. Si la condition de propagation est remplie, la correspondance transformée est associée à son grand-parent de l'arbre de la base qui est ajouté, s'il n'était pas encore présent, dans la file de sélection des sous-arbres de la base. L'algorithme s'achève lorsque la file des sous-arbres de la base est soit vide, soit constituée uniquement de sur-racines virtuelles.

Les correspondances conservées sont les correspondances racines dans l'arbre des correspondances, i.e. les correspondances n'ayant pas été utilisées comme base à une opération de fusion ou de propagation.

11.2.6 Exemple

Nous traitons ici d'un petit exemple de consolidation de correspondances à travers une fonction en langage Java ayant pour fonction d'obtenir les caractères d'un fichier texte dans leur ordre inverse. Une version obfusquée est réalisée en changeant la structure de la fonction. Le bloc try-finally est supprimé avec l'entourage de l'instruction du finally par une structure conditionnelle complétée d'un bloc *sinon* inutile, tandis que, à un niveau supérieur, le code de lecture de fichier est entouré par une boucle inutile à itération unique. La version originale du code, la version obfusquée ainsi que les squelettes des arbres de syntaxe correspondants sont spécifiés en figure 11.1. À des fins de simplification, les déclarations initiales ainsi que l'instruction de retour sont ignorées. D'autre part, toujours pour simplifier, toute correspondance préfusionnée est automatiquement fusionnée définitivement : $t_m = t_M$.

Les correspondances élémentaires sont d'abord cherchées pour chaque sous-arbre de la fonction f de la base : nous trouvons (A', A) (lecture du fichier), (B', B) (fermeture du fichier) et (C', C) (inversion de la chaîne). Pour simplifier, nous convenons des volumes suivants pour les sous-arbres manipulés : $\mathcal{V}(A) = \mathcal{V}(A') = 2$, $\mathcal{V}(B) = \mathcal{V}(B') = 1$, $\mathcal{V}(C) = \mathcal{V}(C') = 2$ et $\mathcal{V}(\text{else}) = 2$. Nous examinons le plus petit sous-arbre de f (base) dont les enfants participent

Données : Correspondances élémentaires (U_i, V_i) associées à leur sous-arbre parent de la base $\mathcal{P}(V_i)$
Résultat : Correspondances consolidées racines R

début
  $F \longleftarrow \{\cdots \mathcal{P}(V_i) \cdots\}$;
  tant que $F \neq \emptyset$ **faire**
    $\beta \longleftarrow$ plus petit arbre de F ;
    $F \longleftarrow F - \{\beta\}$;
    $\text{corr}(\beta) = \{(U_1, V_1), (U_2, V_2), \cdots, (U_k, V_k)\} \longleftarrow$ correspondances telles que V_i soit un ensemble de sous-arbres enfants de β ;
    $\text{corr'}(\beta) = \{(U'_1, V'_1), (U'_2, V'_2), \cdots, (U'_{k'}, V'_{k'})\} \longleftarrow \text{fusion}(\text{corr}(\beta))$ $k' \leq k$;
    pour $(U'_i, V'_i) \in \text{corr'}(\beta)$ **faire**
      si V'_i *est propageable* $(\frac{\mathcal{V}(V'_i)}{\mathcal{V}(\mathcal{P}(V'_i))} \geq t_p)$ **alors**
        $(U'_i, V'_i) \longleftarrow (U'_i, \beta)$;
        si β *n'est pas racine d'un arbre* **alors**
          Association de (U'_i, β) avec $\mathcal{P}(\beta)$;
          $F \longleftarrow F \cup \{\mathcal{P}(\beta)\}$;
          $\lambda \longleftarrow$ faux ;
        sinon
          $\lambda \longleftarrow$ vrai ;
      sinon
        $\lambda \longleftarrow$ vrai ;
      si λ **alors**
        $R \longleftarrow R \cup \{(U'_i, V'_i)\}$;
  retourne R
fin

Algorithme 7 : Consolidation de correspondances par arbres de syntaxe

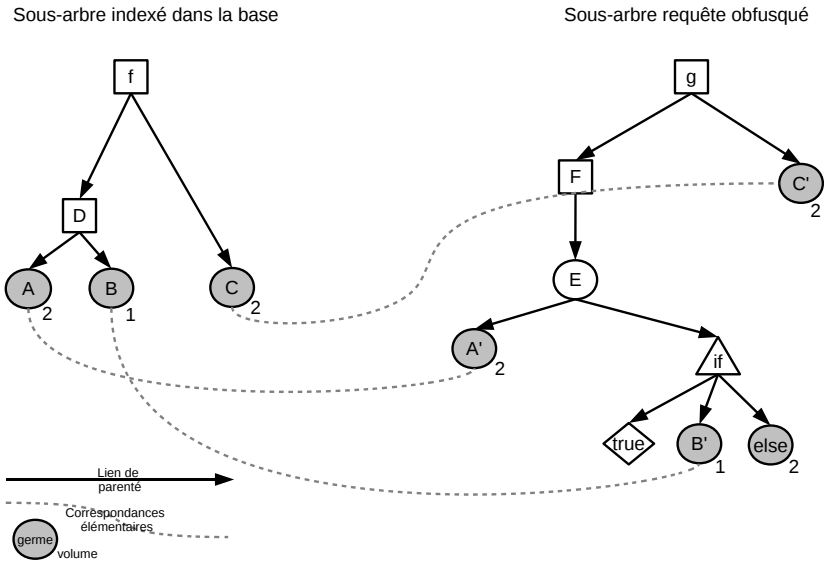
```
1  String reverseFile(String filename) throws
       IOException
   {
    StringBuilder sb = null;
    Reader r = null;
    try {
6     r = new FileReader(filename);
     sb = new StringBuilder();
     while (int read = r.read() != -1)
      sb.appendCodePoint(read);
    } finally
11  {
     r.close();
     System.err.println("Reading the file was
          successful");
    }
    for (int k=0; k < sb.length(); k++)
16  {
     // Swap the characters k and len-1-k
     char tmp = sb.charAt(k);
     sb.setCharAt(k, sb.charAt(sb.length()-1-k));
     sb.setCharAt(sb.length()-1-k, tmp);
21  }
    return sb.toString();
   }
```

(a) Fonction originale indexée dans la base

```
1  String reverseFileContent(String f) throws
       IOException
2  {
    StringBuilder sb = null;
    Reader reader = null;
    for (int i=0; i % 2 == 0; i+=1)
    {
7    reader = new FileReader(f);
     sb = new StringBuilder();
     while (int read = reader.read() != -1)
      sb.appendCodePoint(read);

12   if (i % 3 <= 1)
     {
      r.close();
      System.err.println("Success of the operation"
           );
     } else
17   {
      for (int j=0; j < sb.length(); j++)
       for (int k=0; k < j; k++)
       {
        int s = 0;
22      while (s == 0)
         s += 1;
        sb.setCharAt(k, '\0');
       }
     }
27  }
    {
     for (int j=0; j < sb.length(); k++)
     {
      char tmp = sb.charAt(j);
32    sb.setCharAt(j, sb.charAt(sb.length()-1-j));
      sb.setCharAt(sb.length()-1-j, tmp);
     }
    }
    return sb.toString();
37 }
```

(b) Fonction obfusquée requête



(c) Arbres de syntaxes

FIG. 11.1 – Un exemple de deux fonctions avec leurs squelettes d'arbres de syntaxe formant une correspondance approchée déterminée par consolidation

à au moins une correspondance : il s'agit de D. Pouvons nous fusionner les correspondances (A', A) et (B', B) ? Le LCA de A' et B' est E dans l'arbre requête. En supposant les volumes liés au nœud *if* et à la condition invariablement vrai nuls, $\rho = \frac{\mathcal{V}(A')+\mathcal{V}(B')}{\mathcal{V}(D)=\mathcal{V}(A')+\mathcal{V}(B')+\mathcal{V}(\text{else})} = \frac{3}{5}$. Selon la valeur du seuil de fusion, deux situations sont rencontrées :

1. Si $\rho \geq t_m$, la fusion est réalisée pour obtenir la correspondance $(E, \{A, B\})$. Toutefois comme $\frac{\mathcal{V}(E)}{\mathcal{V}(\mathcal{P}(E)=F)} = 1$, nous choisissons le parent du LCA E, soit F, d'où la correspondance $(F, \{A, B\})$. Celle-ci est ensuite propagée à $\mathcal{P}(\{A, B\})$, i.e. D. La correspondance devient (F, D). f possède donc deux correspondances sur ces enfants : la correspondance consolidée (F, D) et la correspondance élémentaire (C', C). Celles-ci sont fusionnées en $(lca(C', F) = g, \{D, C\})$ car $\rho = 1$: par propagation nous obtenons la correspondance (g, f).
2. Si $\rho < t_m$, la fusion n'est pas réalisée. (A, A') ou (B, B') peuvent néanmoins se voir propagés dans le sous-arbre f de la base si $\frac{\mathcal{V}(A)}{\mathcal{V}(D)} \geq t_p$ et $\frac{\mathcal{V}(B)}{\mathcal{V}(D)}$ respectivement. Si $\frac{1}{3} < t_p \leq \frac{2}{3}$, seule (A, A') est propagée pour devenir (D, A'). (D, A') et (C, C') ne peuvent ensuite pas être fusionnés car $\rho = \frac{\mathcal{V}(A')+\mathcal{V}(C')}{\mathcal{V}(lca(A',C')=g)} = \frac{4}{7} < \frac{3}{5} < t_m$.

Nous noterons que dans tous les cas, si $\mathcal{V}(\text{else})$ tend vers l'infini, aucune fusion sur A' et B' ne peut être réalisée : inonder le code source par du code inutile limite les possibilités de consolidation de correspondances. Toutefois, adopter un seuil de fusion t_m très bas peut conduire, dans certains conditions, à des fusions non désirées. Trouver un compromis sur les seuils t_m et t_p est donc délicat et nécessite des tests expérimentaux.

Il est possible d'introduire une externalisation du code d'inversion de la chaîne dans la version obfusquée de la fonction : ce code serait présent dans une fonction externe appelée depuis la fonction principale. Les graphes d'appels de la fonction originale et obfusquée sont alors nécessaires afin de consolider le code de la fonction principale avec celui externalisé (cf 11.3).

11.2.7 Complexité

La complexité spatiale du processus de consolidation est linéaire en nombre de nœuds de l'arbre requête (pour le stocker et conserver la table de détermination de *lca*) ainsi qu'en nombre de correspondances élémentaires. La complexité temporelle dépend directement du nombre de paires de correspondances candidates pour chaque processus de fusion, du nombre d'opérations de fusion réalisées ainsi que du nombre d'opérations de propagation.

Dans le pire des cas ($t_p = 0$), toutes les correspondances sont propagées à leur parent dans l'arbre de la base : une correspondance entre un sous-arbre de la base et un sous-arbre de l'arbre requête sera étendue à la racine de l'arbre concerné de la base. Le coût temporel est en $O(h)$ où h est la hauteur maximale d'un arbre de la base.

Concernant les opérations de fusion, nous devons considérer le nombre de correspondances $\gamma = |\text{corr}(B)|$ associée à un sous-arbre B de la base. Dans le pire des cas, les correspondances sont fusionnées dans l'ordre inverse de celui de leur présence dans la pile. Cela nécessite $O(\gamma^2)$ opérations de calcul de LCA et de ratio ρ de couverture pour chacune des paires testées

pour chaque préfusion réussie, soit globalement $O(\gamma^3)$. Dans le pire des cas, en supposant le nombre de duplications internes de chaque sous-arbre dans l'arbre de requête borné par d, jusqu'à dk correspondances peuvent être associées à un sous-arbre B d'arité k de la base (soit une complexité pour l'étape de fusion de $O((dk)^3)$). Le coût peut ainsi devenir important pour, par exemple, un bloc d'instructions de la base contenant de nombreuses instructions de squelette typique aussi présentes en grand nombre dans l'arbre de requête (e.g. instructions de déclaration, d'incrémentation...). Il peut donc être intéressant d'ignorer l'existence de tels sous-arbres fréquents pour le procédé de fusion ainsi que d'extension : les correspondances les concernant sont ignorées et la détermination de la métrique de volume est révisée en conséquence.

11.3 Extension à travers les graphes d'appels

L'extension des correspondances à travers les graphes d'appels afin de gérer les opérations de factorisation et/ou développement de fonctions est assez similaire à la procédure d'extension par les arbres de syntaxe. Il s'agit, à partir de correspondances infra-fonctionnelles, de réaliser des opérations de fusion de composantes sur le graphe d'appels requête et de propagation sur le graphe d'appels de la base.

Dans un premier temps, nous clarifions la notion de correspondances infra-fonctionnelles. La connaissance des graphes d'appels étant par nature indispensable pour la procédure d'extension, nous nous intéressons à leur obtention ainsi que leur conservation pour les projets de la base.

11.3.1 Correspondances infra-fonctionnelles

Après la consolidation des correspondances élémentaires par les arbres de syntaxe, nous obtenons un jeu de correspondances consolidées racines ne pouvant être ni fusionnées ni propagées. Ces correspondances peuvent mettre en relation, aussi bien pour la composante sur l'arbre de requête que celle sur la base, des sous-arbres infra-fonctionnels (appartenant à un arbre représentant une fonction) ou ultra-fonctionnels (représentant une structure contenant des fonctions telle qu'une unité de compilation ou une classe). Nous supposerons, dans un souci de simplification, qu'il n'existe pas d'implantations de fonctions imbriquées[3]. Si des fonctions imbriquées sont présentes, celles-ci peuvent être désencapsulées lors de l'étape de normalisation de l'arbre de syntaxe au prix de la communication de paramètres complémentaires pour la communication de variables de contexte local.

Lors du processus de consolidation via les arbres de syntaxe décrit dans la section précédente, nous conservons les correspondances portant sur des paires d'arbres infra-fonctionnels lorsqu'une fusion entraîne l'obtention d'une composante ultra-fonctionnelle (survenant par exemple pour deux unités de compilation similaires) ou lorsque ceci est le fait d'une propagation dans un arbre de la base (par exemple si l'arbre de la base est une unité de compilation

[3]Ceci est vrai pour certains langages procéduraux (C) et orientés objet (Java, C++, C#...) dans une certaine mesure (si l'on exclut par exemple les classes anonymes en Java). D'autres langages tels que les langages fonctionnels (Haskell, Caml...), des langages procéduraux (Fortran, Pascal...) et des langages orientés objet (Ada, Python, JavaScript...) supportent les fonctions imbriquées (*closures*).

ne contenant qu'une fonction similaire avec une fonction de l'arbre requête). Les correspondances infra-fonctionnelles sont, pour la consolidation par graphes d'appels, homologues aux correspondances élémentaires pour la consolidation par arbre de syntaxe.

Les opérations d'obfuscation par développement ou factorisation de fonctions sont susceptibles d'être réalisées sur plusieurs unités de compilation représentées par des arbres de syntaxe distincts. Les correspondances infra-fonctionnelles doivent donc préalablement être recherchées pour chacun des arbres de syntaxe du projet requête.

11.3.2 DAG d'appels

Comme explicité en section 3.4, l'obtention du graphe d'appels pour un projet nécessite la résolution des liens d'appels. Nous notons que cette résolution n'est pas toujours réalisable statiquement pour certains langages permettant l'usage de pointeurs de fonction ou de fonctions homonymes.

Nous calculons à partir du graphe d'appels le DAG d'appels (cf sous-section 3.4.4), graphe acyclique reliant les composantes fortement connexes du graphe d'appel. Chaque composante fortement connexe est associée à un volume de couverture, mesurant la quantité de code potentiellement accessible depuis une fonction de la composante fortement connexe.

11.3.3 Fusion et propagation des correspondances

Les processus de fusion et de propagation de correspondances décrits en 11.2.3 et en 11.2.4 pour les arbres de syntaxe sont appliqués à partir des correspondances infra-fonctionnelles avec l'aide des DAG d'appels du projet requête et des projets de la base. Quelques remarques doivent néanmoins être réalisées.

Fusion

Considérons deux correspondances (U_1, V_1) et (U_2, V_2) pour illustrer la fusion. Chacune des composantes, du projet requête et d'un des projets de la base, est représentée par un ensemble de structures infra-fonctionnelles et/ou de fonctions complètes. Fusionner les deux composantes du projet requête U_1 et U_2 est réalisé par le calcul du LCA des composantes U_1 et U_2 du projet requête ainsi que du ratio de couverture ρ.

Si les deux composantes U_1 et U_2 représentent des structures infra-fonctionnelles de la même fonction, leur LCA est calculé par l'utilisation de l'arbre de syntaxe de l'unité concernée. Dans le cas contraire, parmi U_1 et U_2, il existe au moins une fonction complète. En supposant U_1 une fonction complète, la métrique de volume utilisée correspond à son volume de couverture tel que défini 3.4.4, i.e. la somme des volumes propres des fonctions de sa clôture transitive. En revanche, si U_2 est une composante infra-fonctionnelle, son volume est défini par son volume propre, ne prenant donc pas en compte les fonctions appelées en son sein. Les LCA de U_1 et U_2 sont alors des fonctions complètes déterminées par le DAG d'appel. Il est utile de rappeler que le volume de couverture d'un ensemble de fonctions n'est pas égal à la somme des volumes de couverture de chacune des fonctions : il est donc nécessaire de conserver, pour chaque fonction, l'ensemble de sa clôture transitive afin de pouvoir calculer la couverture d'un ensemble de fonctions par la somme des volumes propres de l'union des clôtures transitives.

LCA dans un graphe acyclique

Il existe un unique LCA pour deux éléments a et b quelconques d'un arbre : il s'agit de l'ancêtre commun à a et b n'admettant pas parmi ses descendants un ancêtre commun à a et b. Pour un DAG, chaque paire d'éléments (a, b) peut admettre aucun, un ou plusieurs LCA. Calculer les LCA de (a, b), peut nécessiter, dans le pire des cas, de parcourir l'ensemble des prédécesseurs de a et de b en suivant jusqu'à $O(m)$ arêtes pour un DAG à n nœuds et m arêtes. Kowaluk et Lingas [13] proposent une méthode de détermination des LCA de toutes les paires de nœuds de complexité temporelle $\Theta(mn)$ et nécessitant $\Theta(n^2)$ en espace. Elle consiste à calculer, pour chaque nœud du DAG son ensemble de descendants puis l'ensemble des nœuds avec lequel il partage un ancêtre commun (l'union des ensemble des descendants de ses prédécesseurs directs) avec un pointeur vers cet ancêtre. Nous pouvons ensuite déduire toutes les paires de nœuds admettant au moins un LCA avec les pointeurs vers leurs LCA. Si le DAG considéré est dense, ce qui est rarement le cas pour les DAG d'appel, des méthodes plus rapides permettent le calcul d'un des LCA de chaque paire de nœuds (s'il existe). On peut ainsi rechercher [13] les témoins maximaux du produit des matrices booléennes en $O(n^{2.575})$ (complexité liée au produit matriciel) de la clôture transitive du graphe et de sa contraposée, les nœuds étant ordonnés selon un tri topologique.

La préfusion de composantes donne lieu à l'obtention d'autant de correspondances préfusionnées que de LCA, les correspondances de ratio de couverture inférieur au seuil de préfusion t_m étant ignorées.

Pour chaque LCA $\mathcal{L}$ sélectionné sur les composantes de la base pour la préfusion, nous vérifions si le choix d'un des parents du LCA ne modifie pas notoirement ρ : pour l'ensemble des parents $\mathcal{P}(\mathcal{L})$, nous sélectionnons le parent p tel que $\rho' = \frac{\mathcal{V}^+(\mathcal{L})}{\mathcal{V}^+(p)}$ soit maximal avec $\rho' > 1-\epsilon$.

Propagation

Lorsque toutes les correspondances associées à leur composante parent dans la base ont été soumises à l'heuristique de préfusion et de validation de fusion, il est nécessaire de déterminer si celles-ci seront propagées à leur parent. Si la composante de la base est infra-fonctionnelle, celle-ci est testée pour sa propagation à son unique parent. Dans le cas où la composante de la base est une fonction, plusieurs parents, qui sont les composantes connexes appelant cette fonction dans le DAG, peuvent coexister. La condition de propagation est donc testée pour chacun des parents.

La condition de propagation est validée si le ratio de couverture des composantes enfants de la base par rapport à la composante parent considérée est au moins égal au seuil t_p.

Algorithme général

L'algorithme général de consolidation à travers le DAG suit le même schéma directeur que celui de consolidation à travers les arbres de syntaxe individuels. Il s'agit de traiter tout d'abord les fonctions de projets de la base dont le volume de couverture de la composante connexe est le plus faible : nous débutons ainsi le traitement par les composantes fortement connexes feuilles du DAG. Pour chaque composante fortement connexe du DAG d'un projet de

la base, nous tentons de fusionner les correspondances impliquant une des composantes enfant (composante fortement connexe ou composante infra-fonctionnelle) d'une des fonctions de la composante. La fusion réalisée, nous tentons de propager les composantes vers les composantes connexes parents.

11.3.4 Exemple

Nous nous proposons d'appliquer l'heuristique de consolidation sur un petit exemple impliquant une opération d'obfuscation par développement de code. Nous considérons l'ensemble des fonctions originales suivantes indexées dans la base ($C_1, \cdots, C_5$ étant des morceaux de code sans sites d'appel, de volume 1 chacun) :

$$\begin{aligned} f_1 = &\ C_1@f_2C_2 \\ f_2 = &\ C_3@f_1C_4 \\ f_3 = &\ C_5@f_1 \end{aligned}$$

Ces fonctions constituent le DAG d'appels suivant contenant deux composantes fortement connexes :

$$\mathrm{DAG}\left(\{f_1, f_2, f_3\}\right) = \{f_3\} \rightarrow \{f_1, f_2\}$$

Nous avons $\mathcal{V}^+(\{f_1, f_2\}) = \mathcal{V}(C_1)+\mathcal{V}(C_2)+\mathcal{V}(C_3)+\mathcal{V}(C_4) = 4$ et $\mathcal{V}^+\left(\{f_3\}\right) = \mathcal{V}^+(\{f_1, f_2\})+\mathcal{V}(C_5) = 5$. Ces trois fonctions sont réécrites en une unique fonction f_q émulant manuellement une pile d'appels (code C_7 de volume 1) pour gérer la récursivité mutuelle de f_1 et f_2, les morceaux $C'_1, \cdots, C'_5$ étant déjà identifiés en tant que clone de $C_1, \cdots, C_5$:

$$f_q = C'_5 \ \text{ while } \left(C_7 \ \text{ switch}(C'_1, C'_2, C'_3, C'_4)\right)$$

Concernant les composantes fortement connexes de la base, $\{f_1, f_2\}$ dispose des correspondances enfants (C'_1, C_1), (C'_2, C_2), (C'_3, C_3), (C'_4, C_4) et $\{f_3\}$ de (C'_5, C_5).

La composante fortement connexe $\{f_1, f_2\}$, de plus faible volume couverture, est traitée en priorité. La préfusion des composantes C'_1 et C'_2 conduit à l'obtention de la fonction LCA f_q : celle-ci est réalisée si $\rho = \frac{\mathcal{V}(C'_1)+\mathcal{V}(C'_2)}{\mathcal{V}(f_q)} = \frac{2}{6} > t_m$. La préfusion peut continuer avec l'ajout de (C'_3, C_3) et (C'_4, C_4) pour parvenir à un ratio de couverture sur f_q de $\frac{4}{7}$. La fusion est ensuite validée si $t_M \leq \frac{4}{7}$.

La correspondance obtenue par fusion $(f_q, \{C'_1, C'_2, C'_3, C'_4\})$ peut-elle être propagée à la composante connexe $\{f_3\}$ parente de $\{f_1, f_2\}$? Oui car $\rho'' = \frac{\mathcal{V}(C'_1)+\mathcal{V}(C'_2)+\mathcal{V}(C'_3)+\mathcal{V}(C'_4)}{\mathcal{V}^+(\{f_1,f_2\})} = 1 \geq t_p$. Nous obtenons ainsi la correspondance $(f_q, \{f_1, f_2\})$ associée à $\{f_3\}$. La fusion de (C'_5, C_5) avec celle-ci permet d'obtenir $(f_q, \{f_3\})$: les fonctions f_q et f_3 sont donc considérées comme similaires après consolidation.

11.4 Métriques d'exactitude

11.4.1 Définition

Afin de quantifier l'importance relative des correspondances élémentaires, issues d'une similarité de sous-arbres par valeur de hachage pour un certain profil, par rapport aux correspondances consolidées obtenues, nous introduisons une métrique d'exactitude. Pour une

correspondance consolidée $C = (c_1, c_2)$ issue des correspondances élémentaires (germes) non-chevauchantes $D_1 = (d_{11}, d_{12}), D_2 = (d_{21}, d_{22}), \cdots, D_l = (d_{l1}, d_{l2})$, nous définissons deux métriques d'exactitude $\mathcal{E}_1$ et $\mathcal{E}_2$:

$$\mathcal{E}_i = \frac{\mathcal{V}(c_i)}{\sum_{k\in[1..l]} \mathcal{V}(d_{ki})}$$

La métrique d'exactitude $\mathcal{E}_1$ quantifie la participation des germes du projet requête sur l'entité structurelle requête consolidée par fusion. $\mathcal{E}_2$, quant à elle, représente la participation des germes de la base sur l'entité structurelle consolidée de la base obtenue par propagations successives. Il est possible également d'intégrer dans les volumes des composantes des germes un facteur d'exactitude tel que défini en 10.4.5.

11.4.2 Utilité

Les métriques d'exactitude ne sauraient être utilisées pour quantifier la pertinence précise dans une optique de réorganisation du code ou de détection de plagiat d'une correspondance consolidée. Il est possible qu'une correspondance intéressante soit associée à une valeur d'exactitude faible. Ainsi, un bloc de code camouflé par un niveau important d'imbrication dans des multiples structures conditionnelles *si* avec clause *sinon* inutile voit sa valeur d'exactitude $\mathcal{E}_1$ tendre vers 0. La correspondance demeure néanmoins intéressante. Inversement, une correspondance peu pertinente peut être associée à une valeur d'exactitude forte ; cela peut survenir lorsque malgré la proximité des correspondances élémentaires, leur fusion ou propagation n'était pas judicieuse ou simplement lorsque le code support est idiomatique.

Les métriques d'exactitude peuvent être comparées afin d'en déduire d'éventuels scénarios d'édition pour une correspondance (A, B) :

1. Si $\mathcal{E}_1((A,B)) << \mathcal{E}_2((A,B))$, il est plausible que A soit une copie de B avec ajout de code inutile. Malgré cet ajout de code, lors des fusions le LCA englobant les germes sur le projet requête est choisi avec un ratio global de couverture des germes par rapport au LCA faible ($\mathcal{E}_1((A,B))$) : ce ratio peut être potentiellement inférieur à t_M avec les fusions successives.
2. Si $\mathcal{E}_1((A,B)) >> \mathcal{E}_2((A,B))$, une déduction inverse peut être réalisée. Il est néanmoins nécessaire de rappeler que pour des correspondances cousines sur un arbre de la base, seule une propagation des correspondances jusqu'à leur LCA pourrait permettre de les réunir : les correspondances frères doivent à ce titre posséder un ratio de couverture dépassant t_p pour être propagées.
3. Si $\mathcal{E}_1((A,B)) \sim \mathcal{E}_2((A,B))$, les composantes sur le projet requête et la base sont de volume proche. Toutefois, les opérations de fusion étant réalisées sans considération d'ordre sur les sous-arbres, il est possible que les sous-arbres germes soient organisés différemment dans A et B. Cela nous permet notamment de relever des cas de copie avec transposition de code mais peut également générer des cas de fausse positivité lorsque les germes représentent des sous-arbres d'occurrences trop fréquentes (ce qui est généralement le cas pour les petits sous-arbres idiomatiques).

11.5 Étude expérimentale de l'extension de correspondances sur arbres de syntaxe

À titre d'évaluation empirique de l'heuristique d'extension de correspondances proposée, nous nous proposons d'étudier les correspondances étendues relevées sur deux jeux de codes sources à caractéristiques différentes. Dans un premier temps, nous évaluons le projet en C Weltab comportant un fort taux de duplication et déjà très étudié dans des publications traitant de réingénierie de code. Ensuite, nous nous intéressons au paquetage Ant d'Eclipse comportant une densité de duplication plus faible.

11.5.1 Étude de Weltab

Nous évaluons tout d'abord l'heuristique d'extension de correspondances sur le projet Weltab (environ 11K lignes de code). Ce projet est un système de gestion de résultats d'élections porté de Fortran vers C au début des années 1980 et utilisé dans l'État du Michigan. Il présente un style de programmation très linéaire peu structuré (utilisation d'étiquettes de branchement et de sauts) avec la présence de nombreux clones originels. Nous étudions la version utilisée par le comparatif de Bellon et al [86] qui comporte en outre de nouveaux clones artificiellement ajoutés. Les paramètres de l'heuristique d'extension sont fixés aux seuils de $\frac{1}{4}$ pour la pré-fusion, $\frac{1}{2}$ pour la fusion et $\frac{1}{2}$ pour la propagation. Préalablement à l'étape d'extension, nous identifions l'ensemble des paires de morceaux de code dont les arbres de syntaxe respectifs sont exactement similaires et comportent au moins 30 nœuds (volume assimilé au nombre de nœuds) avec une technique d'indexation sur fenêtre[4]. Nous relevons 1282 classes d'équivalence totalisant 30921 paires de clones (ceux-ci pouvant s'intersecter en raison de l'indexation sur fenêtre) qui serviront de germes lors de l'extension. Celle-ci est réalisée en ordonnant arbitrairement les unités de compilation $u_1, u_2, \cdots, u_n$ puis en recherchant pour chaque unité requête u_k des similarités sur la base des unités $u_1, u_2, \cdots, u_{k-1}$. Le processus d'extension étant asymétrique, le choix de l'ordre des unités est susceptible de modifier le résultat obtenu.

Après l'extension, on relève un nombre important de correspondances consolidées racines (1680). Celles dont le volume sur l'arbre requête ou l'arbre de la base est le plus important mettent en relation de larges fonctions ou des unités complètes. Nous constatons en particulier que les unités de compilation `r01tmp.c`, `r11tmp.c`, `r26tmp.c`, `r51tmp.c`, `r101tmp.c` et `rsum.c` forment un groupe d'unités quasiment similaires dont chacune des paires est liée à une correspondance de racine d'arbre de syntaxe. Ils ne diffèrent que par l'initialisation et la condition de continuation d'une boucle itérative. Les unités `rsumxx.c`, `lans.c` et `ejcn88.c` complètent ce groupe d'unités similaires : celles-ci présentent cependant plus d'opérations d'édition. Par exemple, si nous comparons `rsum.c` et `ejcn88.c` par alignement global sur les lignes brutes, outre les renommages de variables ou modifications de littéraux (modifications insensibles par la manipulation de lexèmes abstraits), une quinzaine de lignes d'instructions ainsi que la fonction `setdatetime` sont insérées dans `ejcn88.c` (`rsum.c` comptabilise 387 lignes et `ejcn88.c` 415). Selon la méthode exposée en section 12.3.1, ces deux unités présentent une similarité normalisée par rapport à `ejcn88.c` (ou leur union) de $0,942$ et de 1 par rapport à `rsum.c` (entièrement contenu dans `ejcn88.c`).

[4] L'indexation sur fenêtre est ici particulièrement nécessaire en raison de la présence de longs blocs d'instructions élémentaires.

Les autres paires d'unités présentent une similarité normalisée sur l'union inférieure à $\frac{9}{10}$ (avec 9 paires de similarité comprise entre $\frac{5}{10}$ et $\frac{9}{10}$).

Clones ajoutés Nous examinons la détection des clones injectés au projet Weltab. Outre les clones exacts et variant par des modifications d'identificateurs ou types (de type 1 et 2 selon la taxonomie de Bellon) naturellement relevés, 4 clones de type 3 inter-unités sont présents. Si des structures dupliquées comportant 6 à 7 champs (avec variantes avec ajout et suppression de champs) présentées en figure 11.2 ne sont pas détectées en raison du seuil de volume d'indexation trop faible (leur pertinence en tant que clone pouvant être questionnée), les clones comprenant un germe assez volumineux le sont, notamment pour le fichier `linecoun.c` comprenant une boucle légèrement modifiée de comptage de lignes d'une fonction de `spol.c`

```
24 typedef struct {
25   int vect;
     char* ptr;
     void* next;
     char flags[16];
     unsigned int err;
30   char* ptrs[4];
     void* parent;
   } cnv1_t;
```

(a) cnv1.c

```
23 typedef struct {
     int vect;
25   float val;
     char* ptr;
     void* next;
     char flags[16];
     unsigned int err;
30 } canv1_t;
   typedef struct {
     int type;
     float amount;
     char* ptr;
35   void* next;
     char passwd[16];
     unsigned int flags;
   } canv2_t;
```

(b) canv.c

FIG. 11.2 – Copies de structures

Correspondances d'exactitudes faibles Afin de repérer l'éventuelle existence d'opérations de consolidation peu pertinentes, nous examinons maintenant les correspondances présentant les exactitudes les plus faibles sur arbre requête ou sur la base. Parmi celles-ci certaines présentent une exactitude sur arbre requête forte avec une exactitude sur arbre de la base modeste. À titre d'exemple nous présentons en figure 11.3 une séquence de quatre instructions de `wellib.c` (arbre requête) de volume de 37 nœuds en correspondance avec deux copies dans `vfix.c` (arbre de la base) : l'exemplaire le plus profond de l'arbre de la base est propagé quatre fois (à travers une conditionnelle et une boucle) avec un ratio minimal de $0,66$ jusqu'à atteindre le bloc principal où une fusion a lieu avec l'autre exemplaire. On notera que si les rôles d'arbre requête et d'arbre de la base avaient été inversés, la consolidation n'aurait pu avoir lieu qu'en une unique étape de fusion entre les deux exemplaires de `vfix.c` ; leur ratio de volume cumulé par rapport au volume du LCA (bloc entier) aurait été insuffisant (ici $t_M = \frac{1}{2}$). Une rapide analyse des correspondances présentant un ratio minimal de fusion (minimum : 0,536) ou de propagation (minimum : 0,502) confirme la pertinence des extensions réalisées.

```
        trimlen = itrim(80,report);
        /* printf("trimlen = %d\n",trimlen); */
        report[trimlen] = '\0';
        fprintf(fileid,"%s\n",report);
        blkbuf(80,report);
```

(a) wellib.c

```
x525: sclear();
x526: gtoffi(j,report,0);
            len = itrim(50,report);
            report[len] = '\0';
            fprintf(stderr,"%s\n\n",report);
            blkbuf(80,report);
            askchange("this office",&change,&quit,TRUE,FALSE);
            /* askchange("this office",&change,&quit,TRUE,TRUE); */
            if (quit == 999) goto x535;
            if (quit) goto x600;
            if (! change) continue;
            /* office okay for precinct, proceed with query */
x530: fprintf(stderr,"\nExisting results are:\n");
            totoff = 0L;
            if (offinf[CANDCOUNT] > 0) goto x535;
            /* if (offinf[WRITEINCOUNT] > 0) goto x535; */
            fprintf(stderr,"*** Office has no candidates\n");
            goto x570;
            /* election has ballot candidates */
x535: istart = offinf[CANDSTART];
            iend = istart + offinf[CANDCOUNT] - 1;
            iscat = 0;
            /* iscat = iend + 1; */
            /* if (offinf[WRITEINCOUNT] < 1) iscat = 0; */
            /* if (offinf[WRITEINCOUNT] >= 1)
                  iend += offinf[WRITEINCOUNT]; */
            if (istart <= 0 || iend < istart) failure(512);
            if (quit == 999) goto x238;
            for (k=istart;k<(iend+1);k++) {
                  if (((k-istart+1) > 1) &&
                        (((k-istart+1) % 10) == 1) &&
                      (k < (iend - 1))) {
                        pause();
                        sclear();
                        gtoffi(j,report,0);
                        len = itrim(50,report);
                        report[len] = '\0';
                        fprintf(stderr,"%s (continued)\n\n",report);
                        blkbuf(80,report);
                        };
                  gtcand(k,report,2);
                  cvicl(vote[k],report,28,10);
```

(b) vfix.c

FIG. 11.3 – Utilisation multiple d'un germe de l'arbre requête ($\mathcal{E}_1 = 0,411$, $\mathcal{E}_2 = 1$)

11.5.2 Étude d'Eclipse Ant

Apache Ant est un logiciel en langage Java utilisé pour la construction de projets à partir de la spécification de règles de dépendances entre leurs composantes. La version étudiée est le paquetage livré avec l'IDE Eclipse et utilisé pour le comparatif de Bellon : celle-ci comporte 178 unités de compilation avec environ 19K lignes de code utiles. Nous réalisons tout d'abord une recherche de classes de portions de code similaires d'au moins 20 nœuds utilisant une abstraction ignorant les sous-arbres comportant 4 nœuds ou moins : 1917 classes d'équivalence sont trouvées. Parmi ces classes, nous écartons celles comprenant un nombre de membres trop élevés (plus de 10 membres) car représentant des similarités potentiellement idiomatiques (successions de déclaration de membres de classes ou de constantes, d'affectations simples, de définition de *getters* ou *setters* basiques...). Au sein des classes sélectionnées nous relevons 1087 paires en correspondance. Sont dérivées de ces paires germes, par extension, 481 correspondances consolidées racines.

Classement des correspondances racines Parmi les correspondances racines (non utilisées comme composante pour la fusion ou la propagation), nous distinguons trois ensembles de correspondances :

1. Les correspondances germes non-consolidées (272 paires) dites *solitaires*. Il s'agit de correspondances n'ayant pu être regroupées avec d'autres correspondances germes proches. Elles représentent environ 57% de la totalité des germes.
2. Les correspondances consolidées dont les exactitudes sur l'arbre requête et la base sont de 1 (173 correspondances). Elles sont le résultat du regroupement de correspondances frères consécutives aussi bien sur l'arbre requête que l'arbre de la base. On notera que ces correspondances auraient pu être obtenues également par l'usage d'un farmax sur les chaînes de nœuds consécutifs appartenant à une correspondance comme expliqué en section 10.4.4. L'ensemble des correspondances obtenues ici est néanmoins plus étendu en raison de la méthode de fusion utilisée qui est agnostique sur l'ordre des composantes frères regroupées.
3. Les autres correspondances consolidées dites *approchées* (au nombre de 36). Cette catégorie mérite d'être étudiée plus en détails afin d'évaluer la pertinence des consolidations réalisées.

Correspondances approchées Parmi les correspondances approchée obtenues, deux sous-catégories sont distinguées : la première concerne des correspondances composée d'un unique germe (19 correspondances) ayant subi une ou plusieurs opérations de propagation dans leur arbre de la base. Les correspondances consolidées sont de volume moyen compris entre 31 et 80. Les opérations de propagation sont généralement réalisées à l'échelle locale lorsqu'un germe occupe la majorité du volume d'un bloc d'une boucle ou d'une structure conditionnelle ; elles se limitent dans la quasi-totalité des cas à deux propagations successives. Un seul cas impliquant plus de 2 opérations de propagation (6) a été relevé ; il se base sur un germe non pertinent qui est propagé ainsi inutilement à une fonction entière. Il est exposé en figure 11.4.

Nous choisissons de nous intéresser à la seconde sous-catégorie des correspondances comprenant plusieurs germes (au nombre de 17). Elles comprennent des similarités à un niveau local voire fonctionnel avec des instructions insérées, supprimées ou des expressions réécrites.

```
201     public synchronized static IntrospectionHelper getHelper(Class c) {
            IntrospectionHelper ih = (IntrospectionHelper) helpers.get(c);
            if (ih == null) {
                ih = new IntrospectionHelper(c);
205             helpers.put(c, ih); }
            return ih; }
```

(a) ant.IntrospectionHelper

```
465     private InputStream getResourceStream(File file, String resourceName) {
            try {
                if (!file.exists()) {
                    return null; }
                if (file.isDirectory()) {
470                 File resource = new File(file, resourceName);
                    if (resource.exists()) {
                        return new FileInputStream(resource); } }
                else {
                    // is the zip file in the cache
475                 ZipFile zipFile = (ZipFile)zipFiles.get(file);
                    if (zipFile == null) {
                        zipFile = new ZipFile(file);
                        zipFiles.put(file, zipFile); }
                    ZipEntry entry = zipFile.getEntry(resourceName);
480                 if (entry != null) {
                        return zipFile.getInputStream(entry); } } }
            catch (Exception e) {
                log("Ignoring␣Exception␣" + e.getClass().getName() + ":␣" + e.getMessage() +
                    "␣reading␣resource␣" + resourceName + "␣from␣" + file, Project.MSG_VERBOSE); }
485         return null; }
```

(b) ant.AntClassLoader

FIG. 11.4 – Propagation d'un germe de pertinence faible

Quelques cas présentent une similarité à l'échelle d'une classe complète de faible volume comprenant quelques fonctions.

Parmi les correspondances consolidées approchées multi-germes examinées, une peut être considérée non-pertinentes à cause de ses germes qui ne le sont pas par le choix d'une abstraction effaçant les petits sous-arbres. Une autre est étendue inutilement par un germe inadéquat. Les correspondances restantes apparaissent pertinentes et pourraient faire l'objet d'une factorisation ; certains commentaires des développeurs associés à des duplications prévoient cette perspective. L'utilité d'employer une méthode de regroupement de 2-correspondances similaires est soulignée par la présence des classes KaffeRmic, Javac12 et SunRmic regroupables en une 3-correspondance. On notera également l'imbrication sur ZipLong de deux correspondances.

En conclusion, nous pouvons constater sur cet exemple que la consolidation permet d'augmenter le volume des correspondances par le regroupement de plusieurs germes (ici 52 germes ont été regroupés pour former 17 correspondances). Lorsque les germes sont pertinents, la consolidation apparaît indiquée. La principale difficulté réside dans le choix de la méthode d'obtention de germes. Si le choix d'une abstraction forte permet un meilleur rappel de germes, elle introduit de nombreuses correspondances non-pertinentes. Il est possible de s'en prémunir en ignorant comme ici les classes d'équivalence de forte cardinalité. Toutefois, cela pourrait être préjudiciable lorsque des germes issues de classes peuplées et d'autres de classes de faible cardinalité se cotoient dans un arbre de syntaxe requête : les premiers sont ignorés et peuvent compromettre la fusion des seconds. Une solution serait de considérer l'ensemble des classes d'équivalence et d'introduire, à des fins de filtrage à l'issue de l'extension, une métrique de singularité. Lors de la détermination des germes, nous utilisons plusieurs familles d'abstraction et associons à chaque germe une métrique de singularité. Celle-ci serait basée sur la cardinalité de la classe d'équivalence la plus spécialisée contenant les deux exemplaires du germe. Pour toute correspondance consolidée, nous associons la singularité du germe le plus singulier. Cette métrique permettrait d'estimer l'idiomaticité de la correspondance consolidée et de prioritiser l'examen des correspondances pour un évaluateur humain.

11.6 Quelques améliorations envisageables

Outre l'introduction d'une métrique de singularité associée au germes et correspondances consolidées évoquée précédemment, quelques pistes peuvent être explorées afin d'améliorer l'heuristique de consolidation de correspondances sur arbre de syntaxe et DAG d'appel présenté.

11.6.1 Favoritisme des fusions et propagations sur composantes d'ordonnancement concordant

En premier lieu, nous notons que la méthode de fusion est ensembliste : le rassemblement de correspondances frères sur la base et la fusion, avec choix de LCA commun, de correspondances proches sur le projet requête, sont réalisés sans aucune considération d'ordre. Ainsi, par exemple, ce procédé permet d'obtenir une correspondance consolidée par fusion $(T_q, \{A_2, A_1\})$ pour les deux arbres $T_q = a(A'_1, A'_2, A'_3)$ et $T_{\mathcal{B}} = a(A_2, A_1, A_4)$. Ceci permet de gérer les cas de transposition de code mais est susceptible d'introduire des cas faux-positifs. En particulier, il serait souhaitable qu'à ratio de couverture égal légèrement inférieur aux seuils ($\rho + \epsilon = t_m$,

Correspondance	Volume requête (exactitude)	Volume base (exactitude)	Germes	Niveau	Observations	Pertinence
Move(106-113), Copy(275-284)	60 (1)	60 (0,98)	4	Local	Réécriture d'expression	Oui
Replace(69-84), RuntimeConfigurable(59-79)	85 (0,75)	65 (1)	2	Multi-fonctionnel	Réécriture d'expression	Non
Rmic[115-168], Javadoc[91-445]	134 (1)	302 (0,83)	8	Multi-fonctionnel	Transposition, duplication	Partiellement
Touch(153-157), Copy(196-202)	48 (1)	69 (0,70)	2	Local	Insertion d'instruction	Oui
War, Ear	180 (1)	224 (0,80)	7	Classe	Insertion d'instruction	Oui
DefaultCompilerAdapter(310-326), Jikes(92-107) (11.6)	121 (0,65)	105 (0,74)	3	Local	Réécriture d'expression, insertion d'instruction	Oui
Kjc(104-112), DefaultCompilerAdapter(210-222)	45 (1)	46 (0,98)	2	Local	Similarité exacte propagée au bloc parent	Oui
DefaultRmicAdapter(117-162), DefaultCompilerAdapter(137-175)	226 (0,81)	177 (0,81)	4	Local	Blocs conditionnels avec ajout d'un cas *else if*	Oui
KaffeRmic, Javac12 (2.3)	125 (0,70)	182 (0,46)	3	Classe	Réécriture d'expression, insertion d'instruction	Oui
SunRmic, KaffeRmic	133 (0,67)	151 (0,58)	3	Classe	Réécriture d'expression, insertion d'instruction	Oui
SunRmic, Kjc	133 (0,64)	113 (0,69)	2	Classe	Insertion d'instruction	Oui
CommandLine(229-233), Javadoc(471-477)	41 (1)	51 (0,82)	2	Fonctionnel	Réécriture d'expression	Oui
Jdk14RegexMatcher(94-98), JakartaRegexMatcher(85-97)	38 (1)	49 (0,78)	2	Fonctionnel	Réécriture d'expression	Oui
Jdk14RegexMatcher(1-99), JakartaOroMatcher(66-98)	92 (0,53)	122 (0,71)	3	Classe	Réécriture d'expression	Oui
ZipEntry(160-177), MailMessage(246-267)	106 (1)	130 (0,815)	5	Fonctionnel	Insertion d'instruction, clone ajouté	Oui
ZipOutputStream(532-536), ZipLong(88-99)	63 (1)	70 (0,9)	3	Local, fonctionnel	Propagation fonctionnelle de similarité exacte	Oui
ZipShort(86-111), ZipLong(88-115) (11.7)	79 (0,91)	108 (0,68)	3	Multi-fonctionnel	Insertion d'instruction	Oui

FIG. 11.5 – Correspondances consolidées approchées à plusieurs germes sur Eclipse-Ant

```
310             if (Commandline.toString(args).length() > 4096) {
                    PrintWriter out = null;
                    try {
                        tmpFile = new File("jikes"+(new Random(System.currentTimeMillis())).nextLong());
                        out = new PrintWriter(new FileWriter(tmpFile));
315                     for (int i = firstFileName; i < args.length; i++) {
                            out.println(args[i]); }
                        out.flush();
                        commandArray = new String[firstFileName+1];
                        System.arraycopy(args, 0, commandArray, 0, firstFileName);
320                     commandArray[firstFileName] = "@" + tmpFile.getAbsolutePath();
                    } catch (IOException e) {
                        throw new BuildException("Error creating temporary file", e, location);
                    } finally {
                        if (out != null) {
325                         try {out.close();} catch (Throwable t) {} } }
                } else {
```

(a) ant.taskdefs.compilers.DefaultCompilerAdapter

```
 91             if (myos.toLowerCase().indexOf("windows") >= 0
                    && args.length > 250) {
                    PrintWriter out = null;
                    try {
 95                     tmpFile = new File("jikes"+(new Random(System.currentTimeMillis())).nextLong());
                        out = new PrintWriter(new FileWriter(tmpFile));
                        for (int i = 0; i < args.length; i++) {
                            out.println(args[i]); }
                        out.flush();
100                     commandArray = new String[] { command,
                                                      "@" + tmpFile.getAbsolutePath()};
                    } catch (IOException e) {
                        throw new BuildException("Error creating temporary file", e);
                    } finally {
105                     if (out != null) {
                            try {out.close();} catch (Throwable t) {} } }
                } else {
```

(b) ant.taskdefs.Jikes

FIG. 11.6 – Correspondance consolidée approchée pertinente d'exactitude homogène sur Eclipse-Ant ($\mathcal{E}_0$ 0,65, $\mathcal{E}_1$ 0,74)

```
532         byte[] result = new byte[4];
            result[0] = (byte) ((value & 0xFF));
            result[1] = (byte) ((value & 0xFF00) >> 8);
535         result[2] = (byte) ((value & 0xFF0000) >> 16);
            result[3] = (byte) ((value & 0xFF000000l) >> 24);
```

(a) zip.ZipOutputStream

```
88      /**
         * Get value as two bytes in big endian byte order.
90       *
         * @since 1.1
         */
        public byte[] getBytes() {
            byte[] result = new byte[4];
95          result[0] = (byte) ((value & 0xFF));
            result[1] = (byte) ((value & 0xFF00) >> 8);
            result[2] = (byte) ((value & 0xFF0000) >> 16);
            result[3] = (byte) ((value & 0xFF000000l) >> 24);
            return result; }
100     /**
         * Get value as Java int.
         *
         * @since 1.1
         */
105     public long getValue() {
            return value; }
        /**
         * Override to make two instances with same value equal.
         *
110      * @since 1.1
         */
        public boolean equals(Object o) {
            if (o == null || !(o instanceof ZipLong)) {
                return false; }
115         return value == ((ZipLong) o).getValue(); }
```

(b) zip.ZipLong

```
86      /**
         * Get value as two bytes in big endian byte order.
         *
         * @since 1.1
90       */
        public byte[] getBytes() {
            byte[] result = new byte[2];
            result[0] = (byte) (value & 0xFF);
            result[1] = (byte) ((value & 0xFF00) >> 8);
95          return result; }
        /**
         * Get value as Java int.
         *
         * @since 1.1
100      */
        public int getValue() {
            return value; }
        /**
         * Override to make two instances with same value equal.
         *
         * @since 1.1
         */
        public boolean equals(Object o) {
            if (o == null || !(o instanceof ZipShort)) {
110             return false; }
            return value == ((ZipShort) o).getValue(); }
```

(c) zip.ZipShort

FIG. 11.7 – Correspondances imbriquées utilisant une conversion d'entier en octets sur Eclipse-Ant

$\rho' + \epsilon = t_M$, ou $\rho'' + \epsilon = t_p$), que ce soit par rapport à un LCA (fusion) ou à un parent (propagation), des ensembles de composantes, dont l'ordonnancement concorde, soit acceptées pour la fusion ou la propagation, alors que des mêmes composantes, avec un ordonnancement non concordant soit refusées. Ainsi, si $t_M = \frac{1}{2} + \epsilon$, avec $\mathcal{V}(A_i) = \mathcal{V}(A'_i) = 1$, la fusion de A'_1 et A'_2 ne serait pas possible, alors que pour $T_r = a(A'_2, A'_1, A'_3)$, celle-ci aurait été possible, car d'ordre concordant avec la fratrie enfant de $T_{\mathcal{B}}$.

Afin de privilégier les fusions et propagations sur composantes d'ordonnancement concordant, une piste envisagée serait l'utilisation d'une métrique de volume corrigée sur le numérateur pour calculer les ratios de couverture. Celle-ci pourrait être une fonction hyperlinéaire du volume, tel que $x \longrightarrow x^{1+\alpha}$ (α étant un petit réel). Nous obtiendrions ainsi le volume corrigé $\mathcal{V}' = \mathcal{V}^{1+\alpha}$.

11.6.2 Considération sémantique

La méthode présentée utilise des seuils de préfusion, fusion et propagation fixes sans prise en compte de la nature des sous-arbres fusionnés ou propagés. Il pourrait être intéressant, à la lumière des résultats expérimentaux, de proposer des seuils personnalisés en fonction des nœuds considérés afin d'améliorer le rappel et la précision.

Quatrième partie

Problématiques connexes

Une étape essentielle de la recherche de similarité réside dans la mise en correspondance de morceaux de code similaires. Le travail d'un outil de recherche de code cloné ne saurait cependant pas se limiter à cette étape : il reste à mettre en perspective les correspondances trouvées pour une présentation pratique à destination d'un utilisateur humain. Nous nous intéressons donc dans cette partie à des méthodes généralistes de post-traitement de correspondances.

Des métriques de similarité entre séquences de lexèmes ou arbres de syntaxes basées sur le calcul de distance d'édition en utilisant des opérations élémentaires de transformation ont déjà été abordées dans un processus décisionnel de choix de correspondances pertinentes ou d'évaluation d'exactitude de correspondances. Les correspondances obtenues, nous nous orientons vers le calcul de métriques de similarité entre unités syntaxiques de différents niveaux. Ces métriques servent à la constitution de matrice de similarité permettant une vision plus globale de la localisation des clones. Ces matrices peuvent être employées comme bases à des représentations graphiques des similarités comme nous pourrons l'explorer au chapitre 14. Elles peuvent aussi être le moyen de consolider des correspondances au sein de même unités structurelles de haut niveau (unités de compilation, paquetage).

Certaines méthodes de recherche de similarité (approchée) conduisent à l'obtention de groupes de 2 -correspondances : ceux-ci peuvent être extensibles par fusion de groupes. Nous étudions ainsi au chapitre 13 quelques techniques de regroupement de correspondances basées sur des matrices de similarité entre correspondances ou alors sur un regroupement direct avec constitution d'un représentant abstrait pour chaque groupe comme pour l'anti-unification.

Il y a une mesure en toute chose.
Horace (Satires)

12
Métriques de similarité

Sommaire

La recherche de similarité sur du code source n'implique pas uniquement la localisation de zones potentiellement clonées : quantifier pour chaque paire d'unité structurelle la notion de similarité est également important préalablement ou postérieurement à la recherche de correspondances. Nous examinons dans ce chapitre deux types de métriques de quantification de similarité.

Le premier type est relatif aux métriques permettant d'obtenir, à partir de chaînes de lexèmes ou d'arbre de syntaxe, une valeur de similarité. Ce type de métrique a pour vocation d'être implantable par des heuristiques de complexité temporelle avantageuse au prix d'une

certaine dégradation de la qualité, l'objectif principal étant de filtrer approximativement les zones à examiner de façon plus approfondie pour une recherche, plus minutieuse de correspondance.

Le second type de métriques est d'utilisation postérieure à la recherche de correspondances. Il s'agit alors d'estimer une valeur de similarité entre des macro-unités structurelles à partir de la donnée de leurs correspondances partagées. Le calcul de matrice de similarité de macro-unités peut alors être utilisé au sein d'un processus d'extension de correspondances.

Pour la suite de ce chapitre, nous discuterons de métriques de similarité définies sur des paires de morceaux de code à valeurs normalisées dans [0..1], 0 étant synonyme de similarité minimale, 1 de similarité maximale. Il est possible d'associer à une telle métrique s une métrique de distance $d = 1 - s$.

12.1 Métriques directes pour la comparaison d'arbres de syntaxe

Les arbres de syntaxe, comme explicité en 9.1.1, peuvent faire l'objet d'une quantification vectorielle de certaines propriétés syntaxiques et sémantiques. Ces métriques sont facilement modifiables par certains procédés d'obfuscation. En revanche, lorsque du code copié sans volonté obfuscatoire est recherché, celles-ci peuvent présenter un intérêt afin de constituer des groupements préalables dont les paires pourront être analysées avec plus d'attention.

La détermination de distance d'édition sur les arbres (cf 5.5) permet parallèlement la détermination de correspondances approchées avec fossés au prix d'une complexité temporelle quadratique en lexèmes traités. Le calcul de distance d'édition est donc utilisable plutôt dans une optique d'approfondissement de la comparaison d'unités que pour une préselection de celles-ci.

12.2 Métriques directes pour la comparaison de chaînes de lexèmes

Quantifier la similarité entre deux chaînes de lexèmes peut être réalisé en utilisant plusieurs types de distance dont leur distance de Hamming ou leur distance d'édition. La distance de Hamming [12] nécessite la simple comparaison de lexèmes de même indice alors que la distance d'édition requiert l'utilisation de méthodes d'alignement par programmation dynamique comme décrites dans le chapitre 5, de complexité temporelle quadratique. Lorsque la détermination de la similarité deux à deux de k chaînes de n lexèmes est requise, calculer la distance d'édition pour toutes les paires nécessite $O(k^2n^2)$ opérations. La distance de Hamming n'est, elle, pas adapté à la comparaison de chaînes pouvant faire l'objet d'insertion ou de suppression de lexèmes.

La table de suffixes d'un ensemble de chaînes peut être mise à profit pour quantifier leur similarité. Ainsi, lorsque deux suffixes de chaînes différentes sont proches dans la table de suffixes et partagent un préfixe commun de longueur $\mathcal{L}$, les chaînes sous-jacentes partagent au moins un facteur de longueur $\mathcal{L}$. Des chaînes partageant de multiples suffixes proches

dans la table de suffixes partagent autant de facteurs communs. De cette observation, nous déduisons deux heuristiques de calcul de matrice de similarité sur un ensemble de k chaînes. La première, inspirée de [17] quantifie les alternances dans la table de suffixes. Quant à la seconde, elle permet le calcul d'une matrice de similarité creuse, seules les paires de chaînes de similarité notable étant considérées.

12.2.1 Quantification des alternances sur table de suffixes

Pourquoi quantifier l'alternance ?

Considérons deux chaînes de lexèmes u et v dont la table de suffixes $t = \mathrm{SA}(\{u, v\})$ est calculée. Si u et v possèdent des suffixes communs de longueur non nulle, ceux-ci sont ordonnés dans t selon l'ordre lexicographique de u et v. Si $u = v$, par convention, les suffixes de u précèdent ceux de v dans la table t. Dans cette situation, il y a alternance parfaite de suffixes égaux de u et de v. Si l'on substitue à chaque suffixe sa chaîne d'appartenance dans la table, nous obtenons $t' = (u)(v)(u)(v)\cdots(u)(v)$. De manière générale, t' est de la forme $(u)^{i_1}(v)^{j_1}(u)^{i_2}(v)^{j_2}\cdots(u)^{i_z}(v)^{j_z}$ avec tous les i_k et j_k non nuls pour $k \in [1..z]$, sauf éventuellement pour i_1 et j_z. Nous pouvons alors quantifier le degré d'alternance $\mathrm{alter}(t)$ de la table t par la formule suivante :

$$\mathrm{alter}(t) = \sum_{k\in[1..z]/i_k>0} (i_k - 1) + \sum_{k\in[1..z]/j_k>0} (j_k - 1)$$

En particulier, si $u = v$, $\mathrm{alter}(t) = 0$. Ceci est également le cas si $|u| = |v|$ et que pour tout lexème d'indice i de u, $u[i] < v[i]$. En supposant $|u| = |v|$ avec u et v générés aléatoirement, la proabilité d'obtention d'une séquence de taille i_k est de $\frac{1}{2^{i_k}}$. La valeur moyenne de $\mathrm{alter}(t)$ est donc de $|u|/2$ et sa valeur maximale de $|u| + |v| - 2$. Afin de gérer les cas où $|u| < |v|$ et obtenir une valeur inférieure à 1 nous normalisons $\mathrm{alter}(t)$ et déduisons une métrique de similarité s_{alter} :

$$\begin{aligned} \mathrm{alter}_n(t) &= \tfrac{1}{|u|+|v|-2}\left(\mathrm{alter}(t) - (|v| - |u|)\right) \\ s_{\mathrm{alter}}(t) &= 1 - \mathrm{alter}_n(t) \end{aligned}$$

Ainsi, si u est un facteur de v, $\mathrm{alter}_n(t) = 0$ et $s_{\mathrm{alter}}(t) = 1$.

Intuitivement, $\mathrm{alter}(t)$ peut donc être considérée comme une métrique de similarité entre u et v, calculable en temps $O(|u| + |v|)$ par l'emploi d'un algorithme de construction de table de suffixes linéaire. Celle-ci pourrait donc être utilisée afin de calculer une matrice de similarité sur un ensemble de chaînes de lexèmes, issues par exemple, de la lexémisation de projets à comparer. Les paires de chaînes de similarité importante pourraient ensuite faire l'objet d'une comparaison plus coûteuse comme par la détermination d'une séquence d'opérations d'édition par alignement.

Exemple Nous reprenons les chaînes $\alpha = abcdef, \beta = gbcdehef, \gamma = cijkcdeh$ dont la table de suffixe a été exposée en figure 6.1. Nous obtenons la séquence de chaînes $t' = \alpha^2\beta\alpha\beta\gamma^2\alpha\beta\gamma\alpha\beta\gamma\beta\alpha\beta^3\gamma^4$. Nous filtrons t' sur α et β pour obtenir $\mathrm{alter}(t < \alpha, \beta >) : t' < \alpha, \beta >= \alpha^2\beta\alpha\beta\alpha\beta\alpha\beta^2\alpha\beta^3$, soit une valeur de 4. Nous faisons de même pour les couples (α, γ) et (β, γ) pour obtenir finalement la matrice de valeurs suivante (les valeurs normalisées sont

parenthésées) :

	α	β	γ
β	4(1/6)		
γ	6(1/3)	8(1/2)	

Calcul de la métrique d'alternance pour chaque paire de chaînes d'un ensemble

Nous proposons deux méthodes de calcul de la métrique d'alternance pour chacune des paires d'un ensemble de chaînes. La première méthode est la plus naturelle : elle consiste à établir une table de suffixes pour chaque paire et à y quantifier les alternances. Pour k chaînes de n lexèmes, la matrice de valeurs de métrique d'alternance est donc calculée en temps $\frac{k(k-1)}{2}(T(n)+O(n))$ avec $T(n)$ le temps de calcul de chaque table de suffixes (généralement en $\Theta(n)$). Pour la seconde méthode, nous supposons qu'une table de suffixes pour l'ensemble des chaînes est déjà calculée (en $T(kn)$) : une méthode de filtrage dichotomique peut alors être employée afin de déterminer la métrique d'alternance pour chaque paire en un temps global équivalent.

12.2.2 Quantification des suffixes sur fenêtre

Déterminer exhaustivement les valeurs de similarité d'une matrice pour un ensemble conséquent de chaînes est particulièrement coûteux : il pourrait être préférable de ne s'intéresser qu'aux valeurs de similarité pour les paires de chaînes présentant le plus haut niveau de ressemblance. À cet effet nous introduisons une métrique quantifiant la similitude entre deux chaînes par la proximité, bornée par une fenêtre, de leurs suffixes dans la table de suffixes.

Découpage de deux chaînes en facteurs communs

Afin d'introduire la métrique, nous considérons le problème du découpage de chaînes en facteurs commun. Il s'agit pour deux chaînes u et v de trouver une décomposition en facteurs communs maximisant la couverture. Celle-ci n'est pas unique : une stratégie exposée par l'algorithme de Greedy String Tiling exposé en 8.4 consiste à identifier les facteurs communs du plus long au plus court. Plus qu'aux facteurs eux-mêmes, nous nous intéressons à leur couverture en nombre de lexèmes des deux chaînes. À cet effet nous proposons la méthode suivante.

Dans un premier temps, nous établissons la table de suffixes de $T = \{u, v\}$. Chaque $u[i]$ est en position $p = \mathrm{rSA}\,(T)\,[(1, i)]$. Pour tout suffixe de v, nous déterminons la valeur de lcp $\mathcal{L}$ avec $u[i..]$: si celle-ci n'est pas nulle et que l'occurrence de facteur sur u et v n'intersecte pas avec des occurrences déjà prises en compte, nous incrémentons la valeur de similarité $s(u, v)$ de $\mathcal{L}$. $s(u, v)$ correspond à la couverture maximale par facteurs non-recouvrants communs de u et de v.

La métrique précédemment définie ne présente cependant pas un réel intérêt pratique : une unité syntaxique peut contenir l'ensemble des lexèmes d'une autre unité, avec des facteurs communs courts avec $s(u, v)$. Seule la considération de facteurs assez longs est intéressante.

Détermination de la métrique sur un ensemble de chaînes

Nous introduisons une métrique variante de la couverture maximale en facteurs définie par les modalités de calcul suivantes pour un ensemble de chaînes $T = \{t_1, t_2, \cdots, t_k\}$. Nous calculons la table de suffixes pour T. Pour chaque chaîne t_i nous parcourons la table de suffixes de l'occurrence de suffixe $t_i[1]$ à $t_i[n]$. Pour le suffixe $t_i[j]$ de position p dans la table, nous considérons l'ensemble des suffixes de la table dans l'intervalle $[p..p+\zeta]$ où ζ est la taille fixée de la fenêtre afin de rechercher les préfixes de suffixes communs. Pour chaque $l \in [1..\zeta]$, il s'agit de déterminer le LCP de $t_i[j]$ et SA (T) $[p+l]$, suffixe de la chaîne $t_{i'}$ (par exemple en déterminant le LCA des intervalles parent de ces deux occurrences de suffixe). Nous incrémentons $s(t_i, t_{i'})$ de la valeur du LCP, en supposant que les occurrences de facteur ne sont pas en intersection avec d'autres facteurs déjà considérés pour le calcul.

La limitation de la taille de la fenêtre permet d'éviter la prise en compte des facteurs courts. Elle rend également la métrique sensible à l'environnement des chaînes : deux chaînes ne possèdent pas la même valeur de similarité selon les autres chaînes incluses dans l'ensemble T considéré. La présence d'environ ζ chaînes identiques dans T avec une fenêtre d'examen de ζ camoufle la similarité éventuelle pour des paires comprenant une des ces chaînes et une autre chaîne de l'ensemble.

Normalisation La valeur de similarité $s(t_i, t_{i'})$ peut être normalisée en la rapportant sur la valeur maximale potentielle $\min(|t_i|, |t_{i'}|)$.

Complexité Cette heuristique permet le calcul d'une matrice creuse comprenant au maximum ζN valeurs (où $N = |t_1| + |t_2| + \cdots + |t_k|$). Si $\zeta << k$, cette méthode prend tout son intérêt en évitant le calcul exhaustif des valeurs de similarité pour chacun des couples des chaînes.

12.3 Métriques consolidées

Postérieurement à la recherche de correspondances, il est utile de mesurer la similarité entre des paires d'unités structurelles d'un certain niveau (classes, unités de compilation, paquetages, projets). Selon la finalité de la recherche de similitudes, les paires d'unités comparées varient. Ainsi par exemple, pour une refactorisation de projet, nous privilégions la comparaison d'unités de compilation d'un même projet alors que la recherche de plagiat entre différents projets implique la comparaison de paires de projets entiers, ou pour une meilleure granularité, de paires d'unités de compilation de projets différents.

Le calcul de matrices de similarité est également utilisable pour la consolidation de correspondances. On pourra par exemple déterminer dans un premier temps une matrice creuse de similarité entre unités structurelles fonctionnelles. Les paires de fonctions dont la valeur de similarité dépasse un certain seuil sont alors considérées comme correspondances consolidées. Le processus peut ensuite être répété à un niveau structurel supérieur pour la comparaison des unités de compilation. Une variante de cette méthode est proposée par l'outil Clone Miner [62, 61] pour trouver des clones structurels. On note que contrairement à la méthode

de consolidation présentée au chapitre 11, cette technique est difficilement adaptable à la recherche de correspondances infra-fonctionnelle mais peut se contenter d'une analyse syntaxique légère.

Nous divisons la détermination d'une métrique consolidée à partir d'un jeu de correspondances en deux sous-problématiques. La première consiste à calculer, pour une paire d'unités structurelles, leur volume d'intersection, i.e. une quantification du code source commun partagé entre les deux unités. Cette quantité doit cependant être rapportée ensuite aux volumes respectifs de chacune des deux unités. À cet effet, la seconde sous-problématique consiste à normaliser le volume d'intersection selon différentes méthodes.

12.3.1 Détermination du volume d'intersection de deux unités

Introduction

Préalablement à la détermination du volume d'intersection entre les unités structurelles U_1 et U_2, nous supposons disposer d'un jeu de 2-correspondances non-chevauchantes $M = \{m_{11} \to m_{12}, m_{21} \to m_{22}, \cdots, m_{1n} \to m_{2n}\}$ où m_{i1} et m_{i2} sont des sous-unités structurelles respectivement de U_1 et U_2.

Nous utilisons une fonction de volume $\mathcal{V}$ définie sur les unités structurelles afin de quantifier approximativement leur volume dans le code source. Le volume peut par exemple être représenté par le nombre de nœuds de l'arbre syntaxique représentant l'unité. Il est également possible d'assimiler le volume d'une unité à la quantité d'information qu'elle représente par une approximation de sa complexité de Kolmogorov [15]. Cette définition du volume est sensiblement différente dans la mesure où les redondances informationnelles de l'unité sont considérées. Celles-ci peuvent être déterminées par une recherche de similarité intra-unité.

Nous introduisons également une fonction d'exactitude $\mathcal{E}_{U_k}$ évaluant, pour chaque composante d'une correspondance, son degré d'approximation par rapport à son unité. Nous supposons que pour une correspondance $m_{i1} \to m_{i2}$, $\mathcal{E}_{U_1}\mathcal{V}(m_{i1}) = \mathcal{E}_{U_2}\mathcal{V}(m_{i2})$. Ceci est notamment vrai pour la définition de la fonction d'exactitude utilisée pour les correspondances étendues (cf 11.4)[1], ou plus prosaïquement pour des correspondances exactes où l'exactitude est de 1.

Le volume d'intersection est donc ainsi défini (identique selon le calcul sur U_1 avec $k = 1$ ou sur U_2 avec $k = 2$) :

$$\mathcal{V}(U_1 \cap U_2) = \sum_{i \in [1..n]} \mathcal{E}_{U_k}\mathcal{V}(m_{ki})$$

Nous introduisons également des volumes d'intersection relatifs à une des deux unités considérées sans pondération par l'exactitude :

$$\mathcal{V}'_{U_k}(U_1 \cap U_2) = \sum_{i \in [1..n]} \mathcal{V}(m_{ki})$$

Il reste ensuite à déterminer, à partir d'un jeu de k-correspondances brutes, les 2-correspondances non-chevauchantes dont l'une des composantes appartient à U_1 et l'autre à U_2.

[1] $\mathcal{E}_{U_k}\mathcal{V}(m_{ik})$ correspond au volume des germes utilisés pour la consolidation.

Calcul du volume d'intersection sans chevauchement des correspondances

Lorsque les correspondances ne se chevauchent pas, ce qui est notamment le cas après l'application d'une méthode de recherche de facteurs non-chevauchants par tuilage de k-grams de taille variable, le calcul du volume d'intersection pour chaque paire d'unité est aisé. Les correspondances sont parcourues dans un ordre quelconque, et pour chacune des $\frac{1}{2}k(k-1)$ paires $(m_{ij}, m_{ij'})$ de composantes distinctes d'une k-correspondance, la valeur $M[\alpha][\beta]$ où α et β sont les indices des unités d'appartenance de m_{ij} et $m_{ij'}$ respectivement. La matrice symétrique M contient finalement les volumes d'intersection pour chaque paire d'unités. L'usage d'une matrice creuse est particulièrement adaptée si le nombre d'unités comparées est important alors que peu de paires d'unités sont potentiellement similaires.

Calcul du volume d'intersection avec chevauchement potentiel des correspondances

Cas de chevauchement L'utilisation d'une structure d'indexation de suffixes afin de récupérer les facteurs répétés de projets lexèmisés est susceptible d'induire l'existence de correspondances chevauchantes. Ainsi, les chaînes $u = xy$ et $u' = x'y'x''$ (avec $x = x' = x''$, $y = y'$) induisent deux facteurs répétés $(xy, x'y')$ et (x, x', x''). Si les volumes des sous-chaînes sont unitaires, la non prise en compte du chevauchement conduit à l'obtention d'un volume d'intersection de 5 (pour les 2-correspondances $(xy, x'y'), (x, x'), (x', x''), (x, x'')$) alors que celui-ci ne devrait pas être supérieur à $\mathcal{V}(u) = 2$. Concernant les 2-correspondances issues de l'extension de germes exacts sur des arbres de syntaxe, l'existence de chevauchements est également possible.

Méthode Nous considérons le cas où les correspondances ont pour composantes des chaînes d'arbres frères[2]. Les k-correspondances sont d'abord décomposées en 2-correspondances dont chacune des composantes concerne une unité différente. Chaque 2-correspondances est de la forme $(i[a..b], i'[a'..b'])$ où i est l'identifiant de l'arbre de syntaxe contenant la première composante (ou son unité structurelle) et $[a..b]$ la chaîne de sous-arbres commençant par le sous-arbre de racine d'identifiant a et se terminant par celui d'identifiant b, les identifiants étant attribués par un parcours en largeur[3]. Il en est de même pour i', a' et b'. Par convention $i > i'$. Nous trions l'ensemble des 2-correspondances selon les critères (i, i', a, a') pour la mise à jour de la matrice de volume d'intersection. Toutes les 2-correspondances concernant une paire d'unité (i, i') sont traitées consécutivement : à chaque nouvelle paire (i, i') abordée nous initialisons une structure permettant de conserver les sous-arbres déjà pris en compte pour le calcul du volume. Pour chaque correspondance $([a..b], [a'..b'])$ (i et i' sont fixés), nous déterminons les portions de $i[a..b]$ et $i'[a'..b']$ déjà prises en compte pour le volume d'intersection. Ainsi, soit $[a..b]$ (ou $[a'..b']$) intersecte avec une composante déjà considérée représentée par l'intervalle de sous-arbre frères $[c..d]$: seul le volume de $[a..b] - [c..d]$ est ajouté au volume d'intersection en cours de calcul de (i, i'). Il est également possible que $[a..b]$ soit une chaîne de sous-arbres enfants d'un des arbres d'une chaîne $[c..d]$ déjà prise en compte : dans cette situation $[a..b]$ est ignorée.

[2]La méthode présentée est immédiatement adaptable à des correspondances de chaînes de lexèmes.

[3]Si A et B sont deux sous-arbres d'identifiant a et b respectivement avec $a < b$, alors soit A et B ne partagent aucun nœud commun, soit A est parent de B.

Structures d'indexation Afin de déterminer pour chaque paire d'unités si une chaîne de sous-arbres $[a..b]$ intersecte avec une chaîne déjà traitée $[c..d]$, et si l'intersection est non-vide d'obtenir la différence $[a..b] - [c..d]$, nous créons, de façon paresseuse, un arbre binaire de segments pour chaque sous-arbre interne A de i et i' spécifiant l'étendue des sous-arbres enfants déjà considérés. Si les sous-arbres marqués $[c..d]$ intersectent $[a..b]$ et sont de la même fratrie, l'arbre de segments permet directement de déterminer $[a..b] - [c..d]$. En revanche, si $[a..b]$ est une fratrie enfant du nœud c déjà marqué, il est nécessaire de parcourir de façon ascendante la branche de $a \longrightarrow c$ pour le déterminer. Au passage, on pourra marquer tous les nœuds de cette branche comme participant à une correspondance. Déterminer $[a..b]$ privé des sous-arbres marqués intersectant est donc réalisé en temps $O(h \log \alpha)$ où α est l'arité maximale de l'arbre et h sa hauteur. La complexité spatiale est en $O(|A|\,|B|)$.

Exemple Nous présentons un petit exemple illustratif de calcul de similarité avec correspondances chevauchantes en figure 12.1. Lorsque l'on considère les correspondances sur le couple de structures (A_1, A_2), on remarque que les correspondances exactes (d, d') et (b, b'') sont chevauchantes avec la correspondance approchée $(a(bcd), a'(b'ed'))$. La correspondance approchée étant de plus faible profondeur, elle est relevée en premier. Lorsque (d, d') est traitée, nous constatons le chevauchement par le marquage de leurs parents respectifs a et a'. Pour la correspondance (b, b''), le chevauchement n'est constaté que sur A_1. En supposant que chaque nœud possède un volume 1 et qu'un sous-arbre a pour volume la somme de ses nœuds, nous pouvons calculer les volumes d'intersection :

$$\begin{array}{lcl} \mathcal{V}(A_1 \cap A_2) & = & \mathcal{V}(a(bcd)) = \mathcal{V}(a'(b'c'd')) = 4 \\ \mathcal{V}(A_1 \cap A_3) & = & \mathcal{V}(d) = \mathcal{V}(d'') = 1 \\ \mathcal{V}(A_2 \cap A_3) & = & \mathcal{V}(d') = \mathcal{V}(d'') = 1 \end{array}$$

Si la non prise en compte de la 2-correspondance (d, d') issue de la 3-correspondance (d, d', d'') paraît légitime car intersectant des structures participant préalablement à des correspondances sur A_1 et A_2, la mise à l'écart de (b, b'') le paraît moins car n'intersectant qu'une structure marquée sur A_1 et non également sur A_2 : il pourrait ainsi être intéressant de distinguer, en plus du volume d'intersection précédemment décrit un volume d'intersection propre $\mathcal{V}_p(U_1, U_2)$ non symétrique. Pour le calcul de $\mathcal{V}_p(U_1, U_2)$ seraient ignorées uniquement les correspondances dont la composante sur U_1 intersecte une composante déjà marquée sur U_1. Ainsi pour l'exemple décrit, nous obtiendrons $\mathcal{V}(A_1 \cap A_2) = \mathcal{V}_p(A_1, A_2) = 4$, mais $\mathcal{V}_p(A_2, A_1) = 5$, (b'', b) étant prise en compte.

12.3.2 Normalisation du volume d'intersection

Plusieurs méthodes de normalisation du volume d'intersection peuvent être employées selon que l'on souhaite le mettre en perspective par rapport au volume d'une des unités structurelles comparées ou à une combinaison de ces deux unités. Ces méthodes nécessitent la connaissance du volume de chacune des unités structurelles, celles-ci étant déjà préalablement calculées (lors de l'indexation par exemple). Nous présentons ici trois méthodes de normalisation induisant une métrique de similarité symétrique :

1. Normalisation par rapport à l'union des unités : $s_{\cup}(U_1, U_2) = \frac{\mathcal{V}(U_1 \cap U_2)}{\mathcal{V}(U_1 \cup U_2)}$. Le dénominateur représentant le volume de l'union des deux unités peut être calculé ainsi : $\mathcal{V}(U_1 \cup U_2) = \mathcal{V}(U_1) + \mathcal{V}(U_2) - \mathcal{V}(U_1 \cap U_2)$.

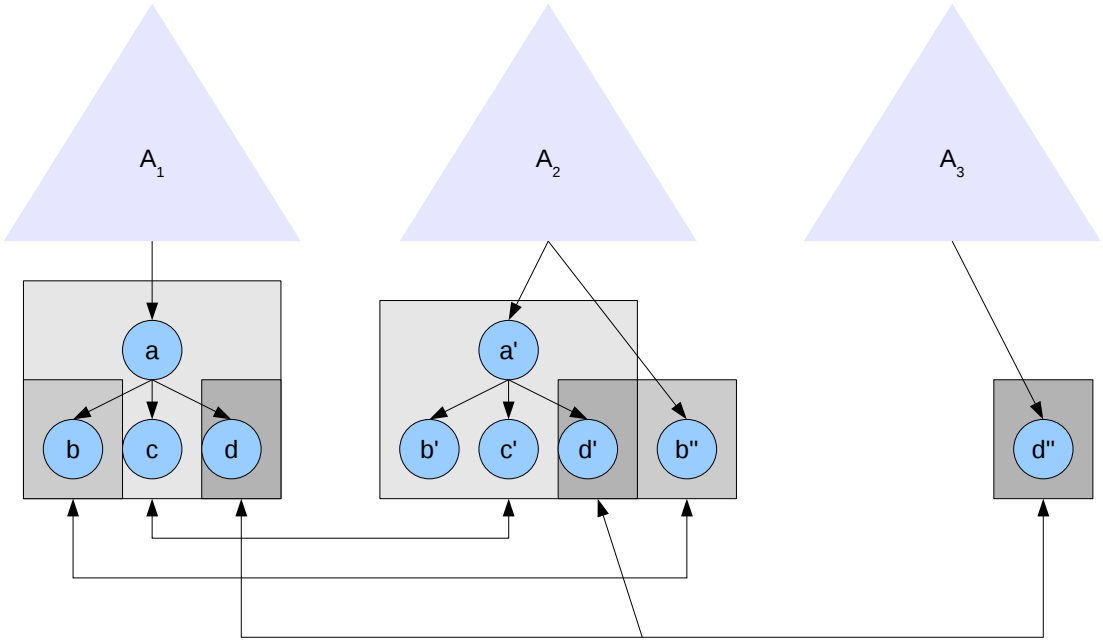


FIG. 12.1 – Correspondances chevauchantes sur trois arbres de syntaxe

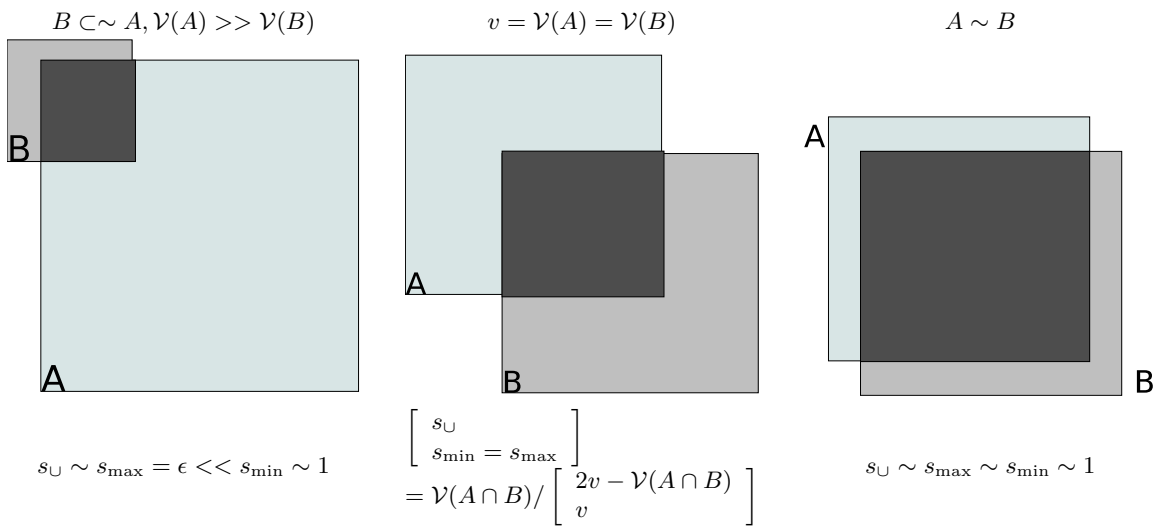


FIG. 12.2 – Trois cas typiques de similarité entre paires de structures

2. Normalisation par rapport à l'unité de plus faible volume : $s_{\min}(U_1, U_2) = \frac{\mathcal{V}(U_1 \cap U_2)}{\min(\mathcal{V}(U_1), \mathcal{V}(U_2))}$. Ce mode de normalisation est particulièrement adapté pour quantifier un éventuel plagiat entre U_1 et U_2 : on suppose qu'une des unités (par exemple U_2) a été dupliquée pour produire l'autre avec l'ajout de code inutile. Ainsi $\mathcal{V}(U_2) >> \mathcal{V}(U_1)$ et $\mathcal{V}(U_1) \sim \mathcal{V}(U_1 \cap U_2)$. La métrique obtenue peut être qualifiée de métrique d'inclusion : $s_{\min}$ est élevée ssi une des unités est incluse dans l'autre.
3. Normalisation par rapport à l'unité la plus volumineuse : $s_{\max}(U_1, U_2) = \frac{\mathcal{V}(U_1 \cap U_2)}{\max(\mathcal{V}(U_1), \mathcal{V}(U_2))}$. Il s'agit ici de quantifier la couverture de l'unité la plus volumineuse par le code commun trouvé sur l'unité la moins volumineuse.

Les deux précédentes métriques de normalisation par rapport à une unité peuvent être résumées par une métrique non-symétrique $\bar{s}(U_1/U_2) = \frac{\mathcal{V}(U_1 \cap U_2)}{\mathcal{V}(U_2)}$. Elle peut notamment être utilisée lorsque le volume d'intersection employé n'est pas symétrique.

De ces méthodes de normalisation peuvent être dérivées des méthodes composites, comme par exemple une moyenne pondérée de $s_\cup$, $s_{\min}$, et $s_{\max}$. Il est toutefois en pratique préférable de présenter à un évaluateur humain un vecteur incluant comme composantes différentes méthodes de normalisation tel que $(s_\cup, s_{\min}, s_{\max})$. Ceci est d'autant plus utile que les unités comparées sont de volume hétérogène. Ainsi en pratique le service web SID [95] utilise la normalisation par rapport à l'union en utilisant une approximation de la complexité de Kolmogorov en guise de volume ; quant à JPlag [76], il emploie une normalisation par rapport à l'unité la moins volumineuse ainsi qu'une normalisation moyenne $\frac{s_{\min}+s_{\max}}{2}$ (également utilisée par SMAT [84]) tandis que Moss [94] réalise une normalisation par rapport au volume des deux unités comparées.

Qui se ressemble s'assemble...

Proverbe populaire français

...Tout ce qui se ressemble n'est pas identique.

William Shakespeare (Jules César)

13

Méthodes de groupement de correspondances

Sommaire

Idéalement, un procédé de recherche de similitude devrait proposer nativement un groupement des morceaux de code présentant un certain degré de similarité. C'est le cas, dans une certaine mesure, lors de la recherche de facteurs répétés sur des chaînes de lexèmes à l'aide d'une structure d'indexation de suffixes (cf chapitre 6) et de graphe de farmax : un facteur répété est complet et comprend toutes les occurrences de ce facteur (le groupe formé n'est ni peuplable, ni dépeuplable). La même remarque s'applique à la recherche de chaînes de sous-arbres frères répétés (cf chapitre 10). Cependant la méthode de consolidation de correspondances présentée au chapitre 11 se contente d'obtenir des 2-correspondances approchées. Sachant que ces 2-correspondances comprennent une composante sur un arbre requête ainsi qu'une autre sur un arbre de la base, il est possible de calculer des empreintes par une technique de hachage pour chacune des composantes de l'arbre requête participant à une correspondance : si (A_1, B_1) et (A_2, B_2) possèdent la même empreinte pour A_1 et A_2, nous pouvons regrouper ces deux 2-correspondances en une 4-correspondance (A_1, A_2, B_1, B_2).

Si le regroupement par hachage des composantes demeure simple, cette méthode ne permet de regrouper des composantes (et ainsi les correspondances où elles sont abritées) que sur la base d'une similarité plus ou moins exacte (selon un profil d'abstraction utilisé lors du hachage). Des métriques vectorielles peuvent être utilisées avec l'emploi ensuite d'une fonction de hachage localement sensible [52] comme l'utilise Deckard [70]. Une vérification des éléments de chaque groupe demeure néanmoins indispensable afin de gérer les cas de fausse-positivité.

Nous examinons ici deux techniques de regroupement de correspondances de similarité approximative. La première utilise directement une représentation par arbre de syntaxe (ou chaînes de lexèmes) et réalise un regroupement par anti-unification : il s'agit de trouver un squelette commun à plusieurs arbres (ou chaînes) avec abstraction de leurs petits sous-arbres discordants (ou petits facteurs discordants) pour les regrouper. Cette méthode présente l'avantage de conserver un représentant abstrait pour chaque groupe. Pour la seconde méthode, nous calculons préalablement une matrice de similarité entre les composantes des correspondances. Cette matrice sert de base au regoupement.

13.1 Groupement direct de chaînes de lexèmes ou d'arbres de syntaxe par anti-unification

13.1.1 Anti-unification

Introduction

L'opération d'anti-unification consiste à générer, pour plusieurs éléments $e_1, e_2, \cdots, e_n$, un élément représentatif plus général E permettant de retrouver chacun des éléments originels par une opération de substitution de ses termes. Il existe plusieurs éléments E plus généraux que $e_1, e_2, \cdots, e_n$ dont par exemple l'élément le plus général $E = \Sigma$ où le symbole Σ peut être remplacé par chacun des éléments originels. L'objectif est alors de trouver un des éléments E minimisant pour chacun des éléments originels les coûts de substitution. Il reste ainsi à définir la fonction de minimisation des coûts de substitution sur les éléments originels. Pour une première approche, si E est l'élément anti-unificateur avec les symboles substitutifs $\sigma_1, \sigma_2, \cdots, \sigma_k$ et e_i dérivable de E en substituant σ_1 par $\sigma_1(e_i)$, ..., σ_k par $\sigma_k(e_i)$, nous cherchons à minimiser la fonction f suivante de somme de coûts de substitution :

$$f : E \longrightarrow \sum_{i \in [0..n]} \sum_{j \in [0..k]} |\sigma_j(e_i)|$$

L'anti-unification peut être réalisée sur des éléments tels que des chaînes de lexèmes, les termes substitués étant des facteurs, ou alors sur des arbres de syntaxe avec pour termes susbtitués des séquences de sous-arbres frères.

Exemple

Nous traitons pour exemple les deux fonctions f_1 et f_2 suivantes réalisant pour l'une la moyenne arithmétique des valeurs d'un tableau et pour l'autre la moyenne quadratique :

Après lexèmisation des fonctions f_1 et f_2, nous déterminons sa chaîne de lexèmes anti-unifiante :

1 **double** moyenneA(**double**[] tab)
{
double somme = 0.0;
4 **for** (**int** i=0; i < tab.length(); i++)
somme = somme + tab[i];
return somme / tab.length();
}
	1 **double** moyenneA(**double**[] c)
	{
	3 **double** addition = 0.0;
	for (**int** i=0; i < c.length(); i++)
	addition = addition + c[i] ∗ c[i];
	return somme / c.length();
	}

f_1 : moyenne arithmétique	f_2 : moyenne quadratique

FIG. 13.1 – Deux fonctions de calcul de moyenne

```
double ID1(tab [] ID2) {
double ID3 = 0.0;
for (int ID4=0; ID4 < ID2 . length; ID4++) ID3 = ID3 + EXPR;
return ID3 / ID2 . length ;
}
```

FIG. 13.2 – Chaîne anti-unifiante des deux fonctions de calcul de moyenne

Les variables identiques sont substituées par les symboles IDk, tandis que l'expression ajoutée à la somme (élément de tableau ou carré de l'élément) est représentée par EXPR.

De façon analogue, nous pouvons déterminer l'arbre de syntaxe anti-unifiant de f_1 et f_2 dont nous spécifions ici le sous-arbre sérialisé concernant l'affectation de sommation :
A = affectation(ID3, binop(+, ID3, EXPR))

13.1.2 Algorithme d'anti-unification de deux éléments

Nous proposons ici une méthode calculant l'anti-unifié en minimisant la somme des coûts de substitution qui repose sur la détermination de la plus longue sous-séquence commune (LCS) entre deux chaînes de lexèmes ou chaînes de sous-arbres frères.

Anti-unification de deux chaînes de lexèmes

Déterminer un anti-unifié de deux chaînes u et v consiste à calculer une chaîne $w = \alpha_1\sigma_1\alpha_2\sigma_2\cdots\alpha_k\sigma_k$ telle que l'on puisse obtenir u et v à partir de w en substituant σ_1, σ_2, ..., et σ_k par des facteurs de u ($\sigma_1(u)$) et v ($\sigma_1(v)$) respectivement. Nous nous intéressons à la recherche d'un anti-unifié minimisant la somme des coûts de substitution (i.e. $|\sigma_1(u)| + \cdots + |\sigma_k(u)|$ et $|\sigma_1(v)| + \cdots + |\sigma_k(v)|$). Cela revient à maximiser la longueur de la séquence de facteurs communs $\alpha_1, \alpha_2, \cdots, \alpha_k$, une des séquences satisfaisant cette condition peut être obtenue par le calcul d'un LCS entre u et v en temps $\Theta(|u|\,|v|)$.

La séquence des facteurs communs $\alpha_1, \cdots, \alpha_k$ étant calculée, nous pouvons déduire les substitutions pour chaque σ_i intersticiels pour u et v. Nous disposons alors d'un jeu de k symboles substitutifs potentiellement réductible en identifiant les substitutions similaires sur u et v. Ainsi si $\sigma_i(u) = \sigma_j(u)$ et $\sigma_i(v) = \sigma_j(v)$, nous remplaçons le symbole substitutif de σ_j par une occurrence supplémentaire de σ_i.

Anti-unification de deux arbres de syntaxe

Nous proposons une méthode récursive d'anti-unification de deux arbres de syntaxe s'inspirant de celle décrite pour les chaînes. Étant donnés deux arbres A et B dont l'on souhaite trouver un anti-unifié, nous comparons tout d'abord les racines de A et de B. N'admettant que des opérations de substitution de sous-arbres entiers et non de nœuds, si les racines sont différentes, A et B n'admettent que l'arbre anti-unifié le plus général σ. Si les racines concordent, nous cherchons à identifier chacun des éléments de la chaîne $C = \alpha_1\sigma_1\alpha_2\sigma_2\cdots\alpha_k\sigma_k$ utilisée pour anti-unifier les chaînes des sous-arbres enfants directs de A et B. Afin de rendre la comparaison des sous-arbres enfants plus aisée, nous identifions chacun d'eux par une empreinte calculée au moyen d'une fonction de hachage : on supposera celle-ci suffisamment longue pour éviter toute collision. L'obtention de la chaîne des facteurs identifiés α_i entremelés de fossés σ_i est suffisante pour déduire un arbre anti-unifié, cet arbre n'étant que de profondeur 2. Si un arbre anti-unifié de profondeur plus importante est recherchée, nous comparons récursivement, selon la même méthode, deux à deux les arbres $\sigma_i(A)$ et $\sigma_i(B)$ correspondant aux fossés. Lorsqu'aucune des racines des fossés ne correspondent, la récursion est stoppée.

Exemple En prenant pour exemple les deux versions de la sommation des fonctions présentées, A = affectation(somme, binop(+, somme, acces(tab,i))) pour la moyenne arithmétique et B = affectation(addition, binop(+, addition, binop(*, accès(tab,i), accès(tab,i)))) pour la moyenne quadratique, nous comparons les racines (affectation) de types égaux. Nous obtenons ensuite la décomposition suivante sur les chaînes de sous-arbre enfants : $C = \sigma_1\ \sigma_2$. Nous confrontons ensuite $\sigma_2(A)$ et $\sigma_2(B)$: la racine binop est identique et les chaînes de sous-arbres sont comparés : +, somme, acces(tab,i) pour A et *, accès(tab,i), accès(tab,i) pour B. Nous en déduisons finalement l'arbre anti-unifié U suivant : U = affectation$(\sigma'_1, \text{binop}(+, \sigma'_2, \sigma'_3)$. Nous constatons que $\sigma'_1(A) = \sigma'_2(A)$ et $\sigma'_1(B) = \sigma'_2(B)$ d'où U = affectation$(\sigma'_1, \text{binop}(+, \sigma'_1, \sigma'_2))$.

13.1.3 Algorithme de regroupement

Étant donné un ensemble d'éléments syntaxiques distincts provenant du code examiné et représenté soit par séquence de lexèmes, soit par arbre de syntaxe, nous pouvons maintenir des groupes d'éléments partageant le même anti-unifié. Concrètement, nous déterminons pour chaque élément syntaxique si celui-ci peut appartenir à un des groupes existants. Pour le vérifier, nous calculons pour chacun des groupes le nouvel anti-unifié du groupe si l'élément devait y être ajouté avec le coût de substitution associé : nous sélectionons le groupe de coût le plus faible. Soit ce coût est inférieur à un seuil fixé de coût de substitution : l'élement est alors ajouté à ce groupe. Dans le cas contraire, un nouveau groupe ne comprenant que cet élément est créé.

Cette méthode de regroupement par anti-unification est notammant utilisée par l'outil de recherche de similarité CloneDigger [90]. Celui-ci réalise dans une première passe le regroupement des arbres de syntaxe représentant des instructions similaires par anti-unification. Chacune des instructions est alors remplacée par un nœud signifiant son groupe d'appartenance et une seconde passe réalise l'anti-unification des fonctions. Cette méthode est cependant de complexité temporelle prohibitive, car pour N éléments distincts à regrouper, jusqu'à $\frac{N(N-1)}{2}$ opérations d'anti-unification sont nécessaires dans les pires des cas où aucun élément ne peut être regroupé.

13.2 Groupement par matrice de similarité

Le groupement par l'usage d'une matrice de similarité peut être utilisé afin de regrouper des correspondances ou des unités structurelles.

Dans le premier cas, il s'agit de calculer une matrice de similarité ou de distance entre les composantes des correspondances préalablement au regroupement. Celle-ci peut par exemple être obtenue en utilisant une des métriques décrite en 12.1 et 12.2. Une telle matrice de similarité est notamment calculée pour le regroupement des fonctions internes et des fonctions feuilles du graphe d'appels inter-projet obtenues par factorisation (cf chapitre 7).

Dans le second cas, la matrice de similarité est obtenue par des métriques consolidées sur les unités structurelles (cf 12.3). Le regroupement permet ainsi d'obtenir des classes d'unités structurelles similaires. Appliqué, par exemple, à des jeux de projets d'étudiants, il permet de mettre en évidence des cliques de projets copiés ainsi que l'éventuelle présence de code standard, présent dans une classe d'unités de cardinalité élevée.

13.2.1 Un aperçu de quelques méthodes de groupement

Nous présentons succinctement quelques techniques de groupement d'éléments. Parmi elles, nous pouvons nous intéresser en premier lieu à celles utilisant des éléments vectoriels et déduisant des distances vectorielles pour le groupement. Celles-ci utilisent la notion de centre pour représenter chaque groupe : les éléments situés à l'intérieur d'un rayon fixé du centre sont inclus dans le même groupe. Ces méthodes sont généralement itératives : il s'agit de définir dans un premier temps un nombre arbitraire de groupes par le choix de certains éléments comme centres. Lors de chaque itération, les éléments sont rattachés au groupe dont le centre est le plus proche. À l'issue de l'itération, les centres de chaque groupe sont alors réévalués. Il y a alors convergence de la constitution des groupes au fil des itérations. Parmi ces algorithmes, nous pouvons citer celui des k-moyennes [16] permettant heuristiquement de minimiser la variance de chaque groupe. En pratique toutefois, il est difficile de pré-estimer le nombre de groupes nécessaires. Certains algorithmes tels que DBSCAN [8] permettent de créer un groupe pour des amas (de cardinalité supérieure à un seuil) d'éléments proches, ces groupes étant étendus par l'ajout d'éléments voisins. En pratique toutefois, si ces méthodes peuvent être adaptées aux métriques de partage de k-grams entre unités, comme pour PDE4Java [69], la recherche de groupes pour des métriques de comparaison de chaînes de lexèmes ou d'arbres de syntaxe est peu adaptée pour ces méthodes.

Nous pouvons ainsi nous orienter vers des méthodes pouvant employer des matrices de similarité ou de distance génériques, sans propriété spécifique intrinsèque à la métrique[1]. Certaines de ces méthodes réalisent l'agglomération d'éléments dans des groupes de manière gloutonne et permettent de déduire, par construction, des arbres de groupes, i.e. des dendrogrammes (cf 14.2.1). Une telle méthode est présentée dans la sous-section suivante. Des approches maximisant la connectivité de chaque groupe sont également envisageables : c'est le cas de l'heuristique MajorClust [20] adaptée par PDetect [74] pour regrouper des projets similaires.

[1] La métrique n'est pas nécessairement une distance dans le sens où l'inégalité triangulaire n'est pas garantie.

Finalement, pour une approche de rassemblement de correspondances ou unités structurelles selon la similarité du code, nous attendons d'un algorithme de regroupement l'optimisation des groupes produits (1), le respect d'une similarité seuil minimale t entre deux éléments d'un même groupe (2) et la non-existence de deux éléments isolés (dans un groupe uni-élément) de similarité supérieure à t (3). Si les deux dernières conditions (2) et (3) peuvent être aisément garanties et vérifiées par un algorithme, la première ne l'est pas. En effet, l'optimisation des groupes peut s'entendre par la minimisation de leur groupe, par la réduction de l'écart-type de la cardinalité de ceux-ci ou par toute autre fonction d'optimisation. Dans le cas général, la condition (1) d'optimisation des groupes rend le problème de détermination de ceux-ci NP-complet [19].

13.2.2 Une heuristique simple de groupement

Nous présentons ici une heuristique simple de groupement gloutonne utilisée pour le regroupement de fonctions similaires obtenues par la méthode de factorisation. Celle-ci garantit, par construction, qu'il n'existe pas deux éléments au sein d'un même groupe dont la similarité serait inférieure à seuil fixé t. Cependant aucune garantie n'est apportée quant à la nature de la répartition des éléments dans chacun groupes : l'existence de groupes de taille hétérogène est possible alors que ceux-ci auraient pu être réorganisés en groupes de taille plus homogène.

Au commencement, tous les éléments appartiennent à des groupes unitaires distincts. Chaque couple d'éléments (e_i, e_j) est examiné par ordre de similarité décroissante jusqu'au dernier couple satisfaisant le seuil de similarité t. Si e_i et e_j appartiennent à des groupes différents, nous fusionnons ces deux groupes ssi tous les couples de leurs éléments présentent une similarité supérieure à t ; dans le cas contraire e_i et e_j ne sont pas fusionnés. Si (e_i, e_j) appartiennent au même groupe, il n'y a rien à faire. Pour chaque groupe, l'historique des fusions l'ayant construit peut être conservé par l'intermédiaire de la construction d'un arbre de groupes (dendrogramme). Nous notons que cette première itération de l'heuristique peut nécessiter jusqu'à $O(n^2)$ opérations.

Tous les couples d'éléments examinés, une étape facultative consiste à itérer la recherche de groupes à fusionner en examinant chacune des paires de groupes et vérifier si la fusion, en respectant le seuil de similarité pour chacun des couples de groupes, est possible.

Nous notons que l'ordre d'examen des couples d'éléments ou de groupes a une influence sur la constitution des groupes finaux.

14

Méthodes de visualisation de correspondances et de similarité

Sommaire

Les duplications de code existant au sein d'un projet ou entre plusieurs projets peuvent être nombreuses et protéiformes. Si des méthodes de consolidation de correspondances telles que celle présentée au chapitre 11 permettent de proposer des correspondances plus étendues et moins nombreuses, il reste à présenter ces résultats à l'utilisateur d'une manière conviviale avec la possibilité d'étudier des similarités à plusieurs échelles du code. Dans un premier temps, nous examinons quelques méthodes de visualisation de correspondances pour ensuite nous intéresser à la visualisation de valeurs de similarité, permettant d'avoir une vision plus globale de l'état des similarités au sein de volumes de code importants, afin d'affiner ensuite l'échelle de l'examen pour étudier des correspondances individuelles.

14.1 Visualisation des correspondances

14.1.1 Confrontation des sources

La visualisation de correspondances par confrontation des sources est proposée par la plupart des outils de recherche de similitudes. Elle consiste à afficher les différents extraits de code source correspondant à la similitude trouvée afin de permettre à l'utilisateur de juger de la pertinence du clone trouvé. Ces extraits peuvent être concaténés dans un fichier texte ou bien visualisés par l'intermédiaire d'une interface graphique. Une telle interface graphique peut se présenter sous la forme de deux zones d'affichage défilantes avec la possibilité d'aligner un exemplaire de correspondance avec un autre, ce qui permet de prendre également connaissance du contexte environnant de chaque exemplaire de clone. Pour une meilleure lisibilité, les exemplaires d'une même correspondance peuvent être marqués par une même couleur. Les sous-zones correspondantes ainsi que les fossés pour des clones non-exacts peuvent également être mis en valeur. Il est possible d'utiliser une application dédiée de visualisation ou bien une sortie HTML, comme cela est l'usage pour les outils de recherche de similitudes uniquement disponibles sous la forme de services Web [76, 94, 95].

Ce procédé de visualisation de similarité, s'il permet d'évaluer précisément la qualité de chaque correspondance, ne peut être utilisé efficacement pour une vue d'ensemble de la nature des similarités trouvées au sein d'un projet ou de plusieurs projets de taille conséquente : des méthodes de visualisation plus globales (*dotplot* ou visualisation de matrice de similarité) doivent être employées. D'autre part, si chaque correspondance comprend de multiples exemplaires, la visualisation simultanée de tous ces exemplaires peut s'avérer difficile.

14.1.2 Graphe de zones de correspondances

L'utilisation de graphes de zones de correspondances (également désigné par *dotplot*) est un moyen assez populaire de visualisation de correspondances. Il s'agit de représenter dans un plan à deux dimensions les emplacements des correspondances par des points. Concrètement si l'on souhaite représenter les clones entre deux ensembles d'unité de code P et Q, nous convenons d'un ordre pour les différentes unités : $P = \{p_1, p_2, \cdots, p_n\}$ et $Q = \{q_1, q_2, \cdots, q_m\}$. Ces différentes unités sont concaténées et divisées en autant de portions que l'axe des abscisses (pour P) ou l'axe des ordonnées (pour Q) contient de pixels. Ainsi chaque pixel représente une zone de code de taille identique et des pixels contigus représentent des zones de code contigües. Les pixels de coordonnées correspondant à une paire de zones de code mises en correspondances sont marqués pour représenter la similarité. En pratique les unités de code analysées pour les abcisses et les ordonnées sont les mêmes ($P = Q$) ce qui permet l'obtention d'un graphe symétrique avec une diagonale marquée (les zones de code sont similaires avec elles-mêmes). Il est envisageable de moduler la nuance du pixel en fonction du degré de similarité de la paire de zones symbolisée par le pixel. Un graphe explorable avec modification interactive d'échelle pourrait être également intéressant.

14.1.3 Graphe d'appels factorisé

Le graphe d'appels factorisé d'un ensemble de projets peut être obtenu par l'application de la méthode de factorisation décrite au chapitre 7. Au cours de cet algorithme, chaque fonction des projets analysés est découpée en une séquence d'appels de fonction : les fonctions appelées

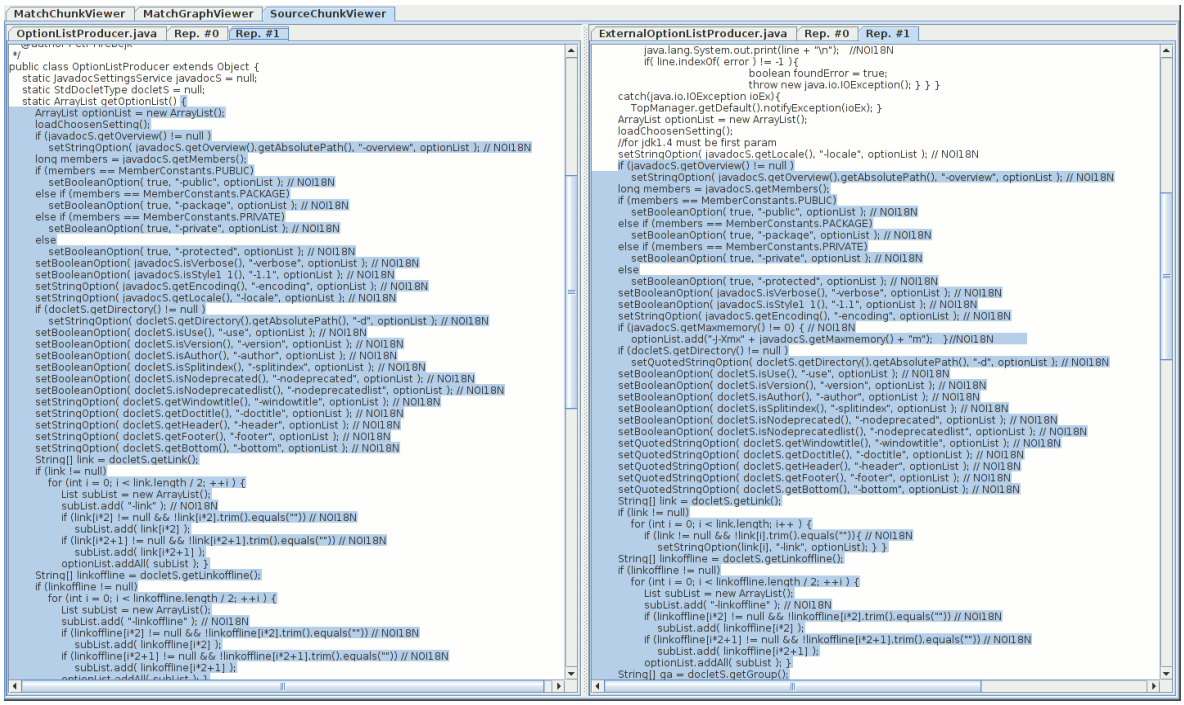


FIG. 14.1 – Visualisation de correspondances avec confrontation des sources avec Plade (cf annexe B.4)

peuvent être des fonctions réelles du code source ou alors des fonctions synthétisées créées à partir de la factorisation d'une séquence de lexèmes primitifs partagée par plusieurs fonctions. Le graphe d'appels factorisé est donc un graphe d'appels classique augmenté par les fonctions factorisées. Il permet ainsi de visualiser des portions de code partagées. Le graphe d'appels factorisé, dans sa forme exhaustive ne présente pas un intérêt immédiat pour une visualisation par un humain : le nombre de fonctions externalisées par factorisation et le nombre de liens d'appel peuvent être potentiellement assez élevés ce qui rend l'exploration du graphe difficile. Cependant des critères de filtrage peuvent être envisagés afin d'explorer certains types de similarité.

Nous pouvons noter que outre l'utilisation qui semble naturelle de ce type de représentation pour le résultat obtenu par la méthode de factorisation, celle-ci peut être exploitée par tout autre méthode de recherche de similitudes. Il s'agit alors de considérer pour feuilles du graphe les portions de code issues de correspondances trouvées puis de déterminer par clôture transitive l'ensemble des fonctions appelant ces portions de code. Nous obtenons ainsi un graphe d'appels représentant uniquement les fonctions appelant (directement ou indirectement) les portions de code mises en correspondance (ces fonctions étant les nœuds internes) et les portions de code elles-mêmes (feuilles). La principale différence avec le graphe d'appels factorisé réside dans le fait que les portions en correspondance ne sont pas elle-mêmes factorisées entre-elles.

14.2 Visualisation de matrice de similarité

Étant donnée une matrice de similarité entre différentes unités de code, il est intéressant de pouvoir visualiser via une méthode humainement intelligible les relations de similarité entre unité. Si finalement la visualisation de correspondances concrètes s'avère indispensable afin de valider par un juge humain des correspondances détectées automatiquement, la visualisation de la matrice de similarité sur des unités de code de plus ou moins haut niveau permet de diriger l'exploration du code et des correspondances vers des zones de plus fort intérêt. Nous sommes malheureusement limités dans nos capacités de visualisation à des représentations à trois dimensions avec propriétés colorimétriques. Des graphes à trois dimensions de type scatter-plot peuvent être utilisés pour représenter des points correspondants à des projections de vecteurs issus de métriques vectorielles sur le code. Cependant si *in fine* toutes les métriques de similarité sont issues de métriques vectorielles, celles-ci sont à valeurs sur un espace vectoriel de dimension dénombrable mais trop élevée pour pouvoir être manipulées. Il s'agit donc, à partir de métriques de similarité, d'utiliser des heuristiques permettant de représenter les relations de similarité entre différentes unités en une, deux ou trois dimensions.

14.2.1 Visualisation de groupes

Nous décrivons au chapitre 13 quelques méthodes de groupement de correspondances ou d'unité structurelles similaires. Il est ensuite utile de pouvoir visualiser les similarités entre ces groupes.

Arbre de hiérarchie de groupes

Certaines méthodes de regroupement d'élements permettent la constitution d'un arbre généralement binaire de ces éléments. Ainsi, lorsque deux groupes (composés d'un ou plusieurs éléments) présentent une similarité significative, ceux-ci peuvent être associés par une relation de fraternité avec la création d'un super-groupe dont ils sont enfants. L'arbre obtenu peut alors être représenté sous la forme d'un dendrogramme. La longueur des arêtes entre un enfant et son parent peut être modulée afin, par exemple, d'être proportionnelle à la différence entre la distance entre cet enfant et le parent et une métrique de distance entre tous les enfants de la fratrie et le parent (qui peut être par exemple la moyenne de distance entre chaque enfant et le parent).

L'utilisation de dendrogramme est surtout populaire en algorithmique génomique afin de représenter des liens de parenté entre espèces selon la similarité de leur génome. Pour la comparaison de code source, il peut être également intéressant pour comparer différentes versions d'un même projet ou alors des projets de parenté proche (telles que des versions de noyaux Unix BSD en figure 14.2). Son emploi pour l'aperçu de la similarité pour un jeu de projets *a priori* d'évolution indépendante mais pouvant comporter des zones de code dupliquées (tels que des projets d'étudiants) est sans doute moins adapté.

Visualisation de similarité entre groupes

Lorsque les groupes ne sont pas ordonnés hiérarchiquement, la similarité entre ceux-ci peut être visualisée en utilisant des méthodes standard de représentation visuelle de matrice de similarité. Il reste ensuite à déterminer la matrice de similarité entre les différents groupes.

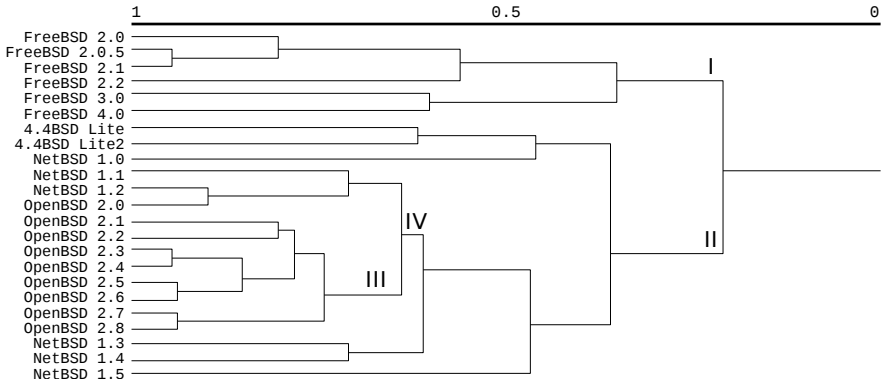


FIG. 14.2 – Dendrogramme[a] obtenu par la comparaison de différents noyaux Unix BSD

[a]Ce dendrogramme est extrait de [84]. Il a été obtenu d'après une matrice de similarité basée sur une métrique utilisant des correspondances calculées par CCFinder et exprimées sous la forme d'intervalles de lignes de code concordantes. La métrique employée utilise une normalisation calculant la moyenne entre $s_{\min}$ et $s_{\max}$ (tels que définis en section 12.3.2).

La similarité $s(C_i, C_j)$ entre deux groupes $C_i = \{a_1, a_2, \cdots, a_m\}$ et $C_j = \{a_1, a_2, \cdots, a_n\}$ peut être obtenue soit par calcul de la métrique de similarité entre deux membres quelconques de ces groupes, soit par combinaison des métriques de similarité entre toutes les paires de membres de ces groupes. Ainsi par exemple, la similarité minimale ou maximale, la moyenne arithmétique ou géométrique peuvent être utilisés.

14.2.2 Carte de similarité

Une carte de similarité (dont un modèle est spécifié en figure 14.3) est comparable à un graphe de zones de correspondances ; la principale différence résidant dans la représentation de chaque unité syntaxique comme un élément atomique avec un certain nombre de pixels. Les unités syntaxiques de comparaison peuvent être des fonctions, unité de compilation voire des paquetages ; à chacune des paires d'unité de compilation est associée une valeur de similarité (représentable par exemple par une propriété colorimétrique). Un exemple de carte de similarité entre des rendus de projets d'étudiants peut être observé en figure 7.11 en illustration de la méthode de factorisation de fonctions de lexèmes.

14.2.3 Graphe de similarité

Définition 14.1. Le graphe de similarité de la matrice de similarité M est le graphe dont les nœuds sont les unités syntaxiques représentées dans la matrice et les arêtes les paires de nœuds (unités syntaxiques) dont la similarité est supérieure à un seuil t.

La représentation par graphe de similarité est particulièrement adaptée à la comparaison d'un nombre réduit d'unités syntaxiques. Dans le cas contraire, la présence d'un nombre de nœuds et d'arêtes potentiellement important peut limiter la lisibilité. Une solution peut toutefois consister à choisir judicieusement le seuil de similarité t afin de n'afficher que les nœuds et arêtes de similarité significative. Nous présentons deux méthodes de visualisation de graphe de similarité : la première est basée sur une correspondance entre score de similarité

	$\leftarrow w(f_1) \rightarrow$		$\leftarrow w(f_j) \rightarrow$		$\leftarrow w(f_n) \rightarrow$
$\uparrow$ $w(e_1)$ $\downarrow$	$s(e_1, f_1)$		$s(e_1, f_i)$		$s(e_1, f_n)$
$\uparrow$ $w(e_i)$ $\downarrow$	$s(e_i, f_1)$		$s(e_i, f_j)$		$s(e_i, f_n)$
$\uparrow$ $w(e_m)$ $\downarrow$	$s(e_m, f_1)$		$s(e_m, f_j)$		$s(e_m, f_n)$

FIG. 14.3 – Carte de similarité entre unité syntaxiques $E = \{e_1, e_2, \cdots, e_m\}$ et $F = \{f_1, f_2, \cdots, f_n\}$

et longueur des arêtes, tandis que la seconde arrange graphiquement les nœuds sous la forme d'un cercle (ou d'une sphère).

Représentation de la similarité par la longueur des arêtes

Le graphe de similarité peut être tracé dans un espace à deux ou trois dimensions en cherchant à employer des longueurs d'arête entre sommets proportionnelle à la valeur de similarité entre les unités représentés par ces sommets. Nous pouvons ainsi essayer de minimiser la moyenne des longueurs d'arêtes pondérées par les valeurs de similarité associées : ainsi pour des sommets $e_1, e_2, \cdots, e_n$, nous cherchons des coordonnées $c(e_i)$ pour chaque sommet e_i dans l'espace de tracé pour les sommets telles que la quantité $\sum_{1 \leq i < j \leq n} 1 - F(s(e_i, e_j))\lambda\left(c(e_i), c(e_j)\right)$ soit minimale (λ étant la distance euclidienne entre des coordonnées et F une fonction croissante de la similarité à valeurs réelles dans [0..1]). Cependant d'autres critères que la minimisation des longueurs pondérées doivent être considérés pour faciliter la lisibilité du tracé telle que la minimisation des intersections d'arêtes ou le respect d'une hiérarchie entre les sommets lorsque cela est possible[1].

Dans le cas général, la détermination de coordonnées des sommets dans l'espace de tracé minimisant un ou plusieurs critères est un problème NP complet [11]. Différentes heuristiques peuvent être utilisées dont celles proposées par l'outil de tracé GraphViz [10] pour un tracé dans le plan. Celui-ci procède en quatre étapes principales. Dans un premier temps, les rangs de chaque sommets sont déterminés par une heuristique de type *network simplex*, ensuite les sommets sont ordonnés sur chaque rang puis sont positionnés, enfin des points d'inflexion de splines sont déterminés pour le tracé des arêtes. La figure 14.4 présente un exemple de tracé par GraphViz sur les données de similarité de [84] précédemment illustrées par un dendrogramme en figure 14.2.

[1]La détermination d'une hiérarchie entre sommets est analogue alors au problème du groupement des nœuds en arbre pour la génération d'un dendrogramme

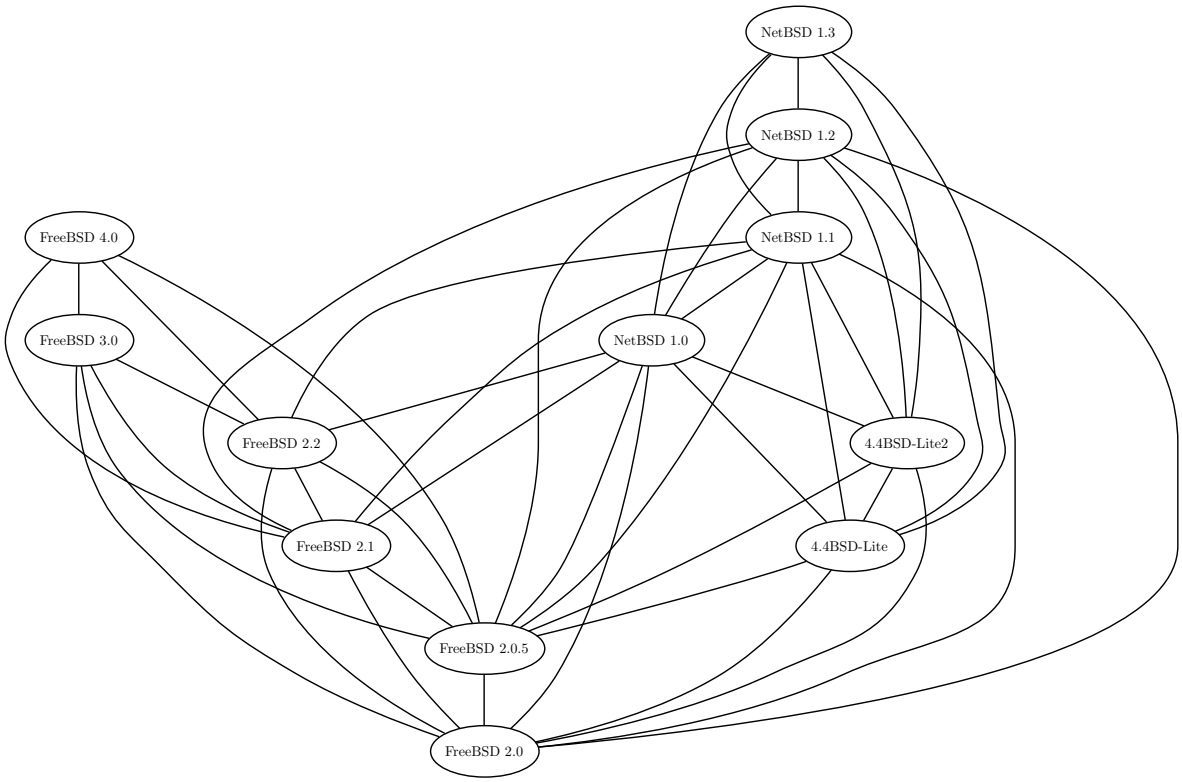


FIG. 14.4 – Tracé par *GraphViz* du graphe de similarité en deux dimensions de la matrice de similarité de noyaux BSD présentée dans [84]

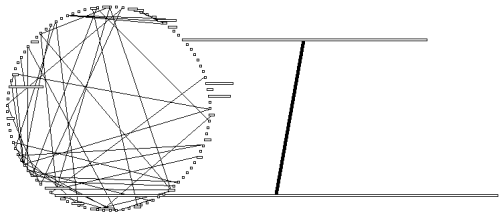


FIG. 14.5 – Représentation polymétrique (relations de similarité externe, quantification de la similarité interne par la largeur des nœuds) sur une roue de similarité du projet *Mailsorting* (extrait de [125])

Visualisation polymétrique de propriétés des entités comparées

Si nous nous sommes intéressés à la représentation des relations de similarité entre entités comparées, il peut être également utile de faire figurer certaines métriques concernant les entités elle-mêmes. Parmi les métriques utiles concernant la duplication de code, nous pouvons citer :

- Le volume de code copié à l'extérieur ($\mathcal{V}_e(e)$) ou à l'intérieur de l'entité ($\mathcal{V}_i(e)$). Nous pouvons affiner la métrique de volume copié à l'extérieur en la paramétrant par la distance du clone extérieur par rapport à l'unité considérée. Ainsi $\mathcal{V}_k(e)$ désigne le volume de code copié et présent dans une unité dont le LCA commun avec e est à une distance de k dans l'arbre des unités.
- Le volume de couverture total par des exemplaires externes de code dupliqué présents également dans l'entité ($\mathcal{V}_c$). Si un unique clone est présent dans l'entité, le ratio $\frac{\mathcal{V}_c(e)}{\mathcal{V}_i(e)}$ (ratio de multiplicité) représente le nombre d'exemplaires du clone dans l'ensemble des projets comparés (moins l'exemplaire de l'entité courante).
- Le volume de code total de l'entité $\mathcal{V}(e)$. Si les clones ne sont pas chevauchants, nous pouvons définir le volume non dupliqué de l'unité par la valeur $\mathcal{V}_n(e) = \mathcal{V}(e) - \mathcal{V}_e(e) - \mathcal{V}_i(e)$.

Ces métriques caractéristiques des entités étudiées ainsi que les valeurs normalisées dérivées peuvent alors être représentées par différents paramètres du tracé du sommet dont par exemple les coordonnées de placement dans l'espace de tracé, sa couleur, les caractéristiques de la forme géométrique de représentation (hauteur, largeur, épaisseur de tracé pour un rectangle)... Un exemple envisageable serait de représenter chaque entité par un rectangle d'aire proportionnelle au volume total de code sous-jacent avec un fond coloré d'intensité proportionnelle au volume de code copié à l'extérieur et un périmètre d'épaisseur reflétant le ratio de duplication. D'autres méthodes de tracé polymétrique sont discutés dans [125] avec une discussion de leur intérêt pour un aperçu rapide de l'état de duplication de code au sein d'un ou plusieurs projets. Un exemple extrait de [125] est présenté en figure 14.5 ; chacune des unités y est placé sur un cercle (de sorte à maximiser la longueur des arêtes représentant les relations de duplication externes en plaçant diamétralement opposées les unités similaires), la largeur des nœuds symbolisant les unités étant proportionnelle aux duplications internes.

Cinquième partie

Perspectives et conclusion

15 Conclusion

Sommaire

15.0.4 Récapitulatif

La recherche de similarité au sein de code source peut avoir de nombreux objectifs : élimination de duplication pour faciliter la maintenance du code source, suivi d'historique de versions, mise en évidence de code plagié... À chacun de ces objectifs correspond une définition de similarité qui lui est propre. Au-delà des applications, la similarité reste une notion subjective liée au jugement humain. Afin d'établir des méthodes algorithmiques déterministes de recherche de similarité, il est nécessaire de convenir d'une représentation du code source ainsi que d'une fonction déterministe et calculable quantifiant la pertinence de la similitude entre deux extraits de la représentation correspondant à des morceaux de code source.

Égalité exacte des extraits de représentation

Nous nous sommes intéressés principalement ici à deux formes de représentation du code source : la séquence de lexèmes et l'arbre de syntaxe. Dans un premier temps, nous considérons une fonction de quantification de pertinence booléenne. Deux exemplaires d'extraits de

représentation forment une correspondance pertinente ssi ces deux extraits sont exactement égaux. Cela implique une égalité de facteurs terme à terme pour des séquences de lexèmes ou de sous-arbres pour des arbres de syntaxe. Nous avons examiné des techniques classiques pour rechercher de telles égalités exactes de facteurs (séquences de lexèmes) ou sous-arbres (séquences de sous-arbres). Elles reposent sur des méthodes d'indexation utilisant diverses structures. Tables de hachages et arbres de recherche peuvent être utilisés pour indexer des empreintes basées sur des valeurs de hachage de facteurs de taille k fixée, appelés k-grams, extraits des séquences de lexèmes. Le hachage par empreintes peut être étendu aux sous-arbres comme nous avons eu l'occasion de l'étudier au chapitre 9. Nous pouvons ainsi en déduire des classes d'égalité d'extraits de représentations, k-grams ou sous-arbres individuels.

Une première difficulté s'annonce dans le choix de la granularité de recherche d'extraits égaux de représentations. Cibler de petits k-grams ou des sous-arbres de taille faible permet de reporter des clones plus petits mais potentiellement en plus grand effectif ce qui rend le passage à l'échelle plus hasardeux.

De classes d'égalité d'extraits individuels de représentations ((méta)-lexème ou sous-arbre), il est nécessaire d'obtenir des classes d'égalité de chaînes d'extraits composés. Cela se traduit par la recherche de séquences de (méta-)lexèmes égales de longueur arbitraire ainsi que de séquences de sous-arbres frères égaux. Ce passage est réalisé en utilisant la relation d'ordre séquentiel des extraits individuels de représentation et en indexant les suffixes de ces extraits. Concrètement des structures d'indexation de suffixes telles qu'une table ou un arbre de suffixes sont mises en œuvre. Elles nous permettent d'obtenir des classes d'égalité de séquences d'extraits avec modélisation des relations de chevauchement de celles-ci par un graphe de facteurs maximaux répétés.

La recherche de similarité sur du code source par mise en évidence de classes d'égalité d'extraits de représentation bénéficie d'une complexité temporelle avantageuse linéaire en la taille des représentations manipulées. Toutefois l'usage d'une fonction booléenne d'égalité exacte comme base de report de correspondances devient insuffisante pour le rappel de duplications ayant fait l'objet de procédés d'édition plus — obfuscation — ou moins — copie intra-projet avec adaptation à un nouveau contexte – importants. À cet effet, nous avons évoqué certains procédés d'abstraction de représentations.

Abstraction

L'introduction d'abstractions sur les représentations manipulées permet l'utilisation de méthodes algorithmiques peu coûteuses de recherche de correspondances exactes tout en introduisant une insensibilité à certaines classes d'opérations d'édition. Nous avons ainsi discuté de la cohabitation de plusieurs profils d'abstraction sur une représentation par arbre de syntaxe. Chaque profil donne lieu à la création d'une empreinte pour chaque sous-arbre, empreinte indexée dans une base. Il est possible de relever ainsi des classes d'égalité exacte selon un profil d'abstraction considéré. Les profils plus spécialisés peuvent servir afin d'affiner des correspondances avec l'introduction d'une mesure d'exactitude permettant une quantification plus fine de la pertinence qu'une fonction booléenne ainsi que la détermination des opérations d'édition séparant deux correspondances égales selon un profil plus général.

Opérations de macro-édition

Les opérations d'abstraction ne peuvent insensibiliser la représentation, qu'elle soit sous forme de séquences de lexèmes ou d'arbres de syntaxe, que pour des opérations d'édition délimitées (abstraction de types, suppression de certains éléments sémantiquement non-essentiels...). Lorsque des opérations d'ajout ou suppression arbitraires de morceaux de code sont envisagées, utiliser des profils d'abstraction liés à des méthodes de hachage exact se revèle combinatoirement impraticable.

Les correspondances basées sur une égalité exacte selon une représentation peuvent alors être utilisées comme germes et être combinées pour former des correspondances englobantes plus étendues et approchées. Les solutions existantes de raccordement de facteurs exacts de lexèmes en correspondance ont été évoquées au chapitre 5 : elles se basent sur des méthodes d'alignement local par programmation dynamique. Nous nous sommes plutôt intéressés ici à l'extension de sous-arbres de syntaxe exactement égaux à des 2-correspondances approchantes. Une heuristique de regroupement des germes exacts selon leur proximité dans leur arbre de syntaxe hôte a été décrite et testée sur quelques projets. Les paires de sous-arbres en correspondance approchée pourraient ensuite être comparées entre-elles pour être regroupées au sein de correspondances d'effectif plus important.

Opérations de micro-édition

Les opérations de micro-édition telles que typiquement la réécriture d'expression apparaissent parmi les obstacles les plus gênants à la recherche de similarité. Celles-ci réduisent le volume des extraits de représentation reportés par une méthode de recherche exacte. Or pour éviter l'obtention d'un grand nombre de correspondances triviales, il est nécessaire d'ignorer des extraits de volume trop faible. L'usage d'une valeur de compromis est nécessaire.

Une piste à explorer consisterait à rechercher non-exhaustivement des correspondances exactes de faible volume afin d'évaluer une estimation de la densité de celles-ci sur le code analysé. Une densité élevée pourrait suggérer ensuite l'usage de méthodes d'alignement locales plus coûteuses autorisant des 2-correspondances entre extraits de représentation approchés. Les extraits en correspondance sont alors caractérisés par une distance en opérations d'éditions élémentaires (distance de Levenshtein).

Ajout des liens d'appel

Nous nous sommes intéressés à la prise en compte des liens d'appel de fonctions aussi bien pour une représentation par séquences de lexèmes que par arbre de syntaxe. Pour des séquences de lexèmes, une méthode de factorisation de fonctions a ainsi été présentée permettant de modéliser plusieurs projets comparés par un graphe d'appel de fonctions représentant des morceaux de code élémentaire partagés. Cette méthode est particulièrement adaptée pour des opérations de factorisation ou de développement de fonctions. Pour les arbres de syntaxe, la considération des liens d'appel s'intègre à l'heuristique d'extension de germes à des correspondances approchées présentées au chapitre 11.

15.0.5 Perspectives

La recherche de similarité au sein de code source offre certaines perspectives sur certaines applications encore peu explorées. Si la recherche de clones intra-projet fait l'objet d'une recherche intense ainsi que de nombreuses études, la détection de correspondances sur des versions obfusquées reste peu abordée. Enfin la connaissance de similarité pourrait être le point de départ à l'élaboration de méthodes de compression spécifiques au code source, par exemple pour son stockage au sein de référentiels avec suivi d'historique de versions. Cette voie d'exploration demeure encore peu explorée. Nous choisissons, pour clôturer ce document, de nous interroger sur quelques idées à explorer pour une meilleure détection de code copié faisant l'objet d'opérations d'édition.

Modèle statistique d'idiomaticité

Nous avons eu l'occasion de nous interroger sur certaines formules idiomatiques de code rencontrées au cours d'analyse de correspondances (cf sections 10.5 et 11.5). Le développement d'une métrique d'idiomaticité pourrait permettre d'affiner l'évaluation de pertinence de correspondances. À cet effet, déterminer et indexer les motifs de code les plus courants est nécessaire. Délimités à un unique projet, ces motifs resteraient utiles à des fins de factorisation du code. Lorsque ces motifs sont collectés sur un grand nombre de projets, ils permettent de pondérer la pertinence de correspondances relevées dans le cadre de la recherche de code plagié. Enfin l'usage de profils d'abstraction peut être adapté à l'idiomaticité du code utilisé. Ainsi si une duplication courante selon un profil général est mise en évidence, un profil de plus faible abstraction pourra être appliqué afin d'affiner la comparaison.

On notera également que la recherche de motifs courants inter-projets peut participer à l'évolution du langage étudié et suggérer l'introduction de nouvelles constructions syntaxiques ou de nouvelles fonctionnalités au sein de ses biliothèques standard afin de limiter la verbosité du code et améliorer la productivité de développement.

Contextualisation des paramètres et interaction avec l'utilisateur

Une des faiblesses des processus actuels de recherche de similarité réside dans le caractère statique de leur paramétrage. Ainsi le choix d'un seuil de volume de report pour des méthodes utilisant l'égalité exacte sur des représentations ou de ratios pour la consolidation de germes s'avère délicat. Un choix conservatif peut être réalisé au détriment de la précision. On pourra ensuite utiliser une métrique de pertinence sur ces correspondances reportées afin de n'en présenter qu'un sous-ensemble à l'utilisateur, avec un certain ordre de présentation. Plutôt qu'une analyse *a posteriori*, il pourrait être intéressant de s'interroger sur la pertinence même de sélection de certains germes ainsi que du caractère judicieux de leur consolidation. Ce comportement est plus avantageux au niveau de la complexité temporelle.

Les paramètres utilisés pourraient ainsi être adaptés au contexte des correspondances localisées. Le volume des correspondances peut être adapté à leur idiomaticité évaluée par des méthodes statistiques. Des petits germes rares peuvent ainsi être reportés au détriment de germes plus volumineux mais courants. Au-delà d'une considération purement statistique, une évaluation de l'utilisateur peut être intégrée afin d'affiner, au cours des emplois de l'outil, les paramètres contextuels. Ainsi par exemple, si l'utilisateur évalue pertinentes plusieurs

occurrences de correspondances consolidées portant sur des fusions à ratio faible entre blocs appartenant à un certain type de structure, le processus de consolidation pourra, par la suite, réduire le ratio de fusion lié à ce type de structure. L'objectif serait de converger, au fil des itérations, vers une précision et un rappel des correspondances optimal, selon le jugement de l'utilisateur. Il reste toutefois des aspects sémantiques influant le jugement de pertinence de l'utilisateur mais difficilement accessibles pour une méthode heuristique automatique.

Amélioration du processus de factorisation

La factorisation de chaînes de lexèmes en facteurs partagés avec intégration du graphe d'appel présenté au chapitre 7 souffre de certaines limitations déjà évoquées. La plus limitante concerne l'impossibilité d'utiliser un jeu incrémental de projets, le processus de factorisation devant être renouvelé complètement. Nous pourrions ainsi envisager la possibilité de fusionner des graphes issus de la factorisation de projets distincts.

Rapprochement du code machine

Nous avons exclu du champ de notre étude l'analyse de traces d'exécution du code. Celles-ci pourraient néanmoins s'avérer utiles dans le cadre de code source fortement obfusqué même si une obfuscation avancée peut permettre de modifier les traces d'exécution. Un procédé intermédiaire pourrait être la considération de représentations, pour l'analyse statique, plus proche du code machine. Toutefois si certaines modifications réalisées par un compilateur peut permettre d'introduire un certain niveau de normalisation du code, d'autres au contraire pourraient au contraire obfusquer certaines structures.

Annexes

A

Liste des outils de recherche de similitudes

De nombreux logiciels de recherche de similitudes sur du code-source utilisant plusieurs types de représentations du code et des approches différentes coexistent actuellement. Nous présentons ici une liste (inévitablement non exhaustive) des outils publiquement disponibles dont nous avons connaissance. Ceux-ci sont classés selon le type de représentation intermédiaire qu'ils manipulent ainsi que la principale approche algorithmique sous-jacente pour la recherche de similarité. Pour chacun des outils nous spécifions si celui-ci est uniquement accessible par le biais d'un webservice ou alors s'il est possible de télécharger une version binaire ou son code source sous une licence libre. Pour l'énumération de cette énumération d'outils, nous nous sommes inspiré du comparatif qualititatif réalisé par Bellon [86]. Outre les outils se définissant comme des logiciels de recherche de clones dans une optique de refactorisation, nous avons également considéré les logiciels considérés comme des outils de recherche de code copié par plagiat.

A.1 Code source brut

A.1.1 Alignement

DupLoc [65] DupLoc permet la visualisation de lignes de code similaires sous la forme de matrice DotPlot où la cellule (i, j) est noircie si les lignes i et j des unités de compilation concaténées sont identiques. Avant l'étape de comparaison ligne par ligne, le code source est légèrement normalisé (lignes vides supprimées). Il est possible d'utiliser différentes échelles pour agrandir certaines zones ou réduire la taille affichée de la matrice avec affichage des pixels en niveaux de gris selon le niveau de similarité.

A.1.2 Autre approche algorithmique

CodeMatch [85] CodeMatch utilise directement le code source brut avec une normalisation du formatage : son adaptation à un nouveau langage est rapide. Il réalise des comparaisons par paires d'unité de compilation en calculant des métriques de similarité basées sur plusieurs cri-

tère. Une métrique recherche la plus longue séquence commune d'instructions entre des unités, une instruction étant définie par le premier mot-clé d'une ligne. D'autres métriques cherchent à mettre en évidence des identificateurs partagés (chaînes entières ou sous-séquences) ainsi que des portions de commentaires dupliqués. CodeMatch présente ainsi une utilité pour la copie avec opérations d'obfuscation manuelles non systématisées préservant certains identificateurs et commentaires. Ce logiciel est en licence propriétaire avec possibilité d'obtenir une version de démonstration.

A.2 Séquences de lexèmes

A.2.1 Metalexèmisation

CPD [92] CPD (Copy Paste Duplication) est un outil de recherche de clones au sein d'une répertoire de fichiers source. Plusieurs versions ont été implantées utilisant des méthodes différentes de recherche de clones exacts sur séquences de lexèmes. La première version utilisait l'algorithme Greedy String Tiling (sans l'utilisation de fonctions de hachage incrémentales de type Karp-Rabin) afin d'agglomérer des lexèmes consécutifs identiques. La deuxième version utilisait une table de suffixes pour la recherche de sous-chaînes répétées. Quant à la troisième version, considérée comme plus rapide, elle se base sur l'utilisation d'empreintes générée par une fonction de hachage incrémentale. CPD ne permet que de trouver des clones exacts (sous-chaînes de séquences identiques) qui sont représentés sous forme textuelle (localisation du clone et extrait du code source correspondant).

CPD est un composant du projet Open Source (sous licence de type BSD) PMD qui est un analyseur de code source Java, qui permet, outre la recherche de clone dupliqué, la détermination de code mort (code non accessible, variables ou paramètres de méthodes inutilisés) et la recherche de code pouvant être optimisé. Des greffons liés à PMD existent pour de multiples environnements de développement (dont Eclipse, Netbeans et Emacs). Si PMD est destiné à analyser du code Java, CPD peut lexèmiser et rechercher des clones sur des unités de compilation d'autres langages (C, C++, Fortran et PHP).

JPlag [93, 82] JPlag est un outil de recherche de similitudes utilisant une représentation des codes source par séquences de lexèmes. Il compare chaque paire d'un jeu de projets soumis en utilisant l'algorithme Running Karp-Rabin Greedy String Tiling [RKR-GST]. Les résultats sont proposés sous la forme de pages HTML avec une sélection des groupes de correspondances les plus longues (les paires de correspondances identiques sont regroupées). Chaque paire de projets peut être comparée à l'aide de deux cadres avec l'utilisation de couleur pour marquer les correspondances communes. Le score de similarité calculé pour chaque paires de projets est la moyenne du score d'inclusion et de couverture.

JPlag est accessible sous la forme d'une applet Java de soumission de projets et nécessite une inscription préalable qui doit être approuvée par l'administrateur. Les calculs sont réalisés sur un serveur, le code-source n'étant pas publiquement disponible.

Plaggie [58] Plaggie est un logiciel Open Source de recherche de similitudes utilisant l'algorithme de métalexèmisation RKR-GST originel. Les auteurs disent s'être inspiré de JPlag pour le réaliser avec la différence notable de sa disponibilité sous licence GPL. Plaggie compare chaque paire de fichiers Java qui lui sont soumis en les représentant par des séquences

de lexèmes obtenus par parcours en profondeur d'un arbre de syntaxe concret obtenu par un analyseur syntaxique généré par Cup.

Moss [56, 94] Moss est un service web à sources fermées réalisant une recherche de similarité sur des séquences de lexèmes. Il utilise une base d'empreintes pour le jeu de projets étudiés en générant pour chaque k-gram une empreinte ; les empreintes de k-grams sont ensuite sélectionnées sur une fenêtre glissante par la méthode Winnowing. Le résultat de la recherche est visualisable sous la forme de pages HTML téléchargeables.

A.2.2 Indexation de suffixes

ccfinderx [71, 89] CCFinderX est le successeur sous licence libre MIT de CCFinder. Il propose un outil de recherche de clones exacts sur une forme lexèmisée et normalisée du code source en utilisant un arbre de suffixes. Une interface graphique permet la visualisation de correspondances sur le code source avec affichage d'une matrice dotplot globale.

A.2.3 Autre approche algorithmique

Unique [96] Unique est un logiciel de recherche de chaînes de lexèmes similaires développé en Python et proposé sous licence GPLv3. Une phase de pré-traitement du code source permet d'obtenir une chaîne de lexèmes d'un ensemble de fichiers chargée en mémoire centrale qui sera utilisée pour la recherche. La lexémisation réalisée est basique (découpage en chaînes de caractères non-espace), applicable à tous les langages et sans opération d'abstraction. Unique est paramétrable par la longueur minimale ($l_{\min}$) et maximale ($l_{\max}$) en nombre de lexèmes des facteurs similaires à trouver. Un algorithme temporellement quadratique en nombre de lexèmes dans le pire des cas est utilisé afin de localiser, pour chaque occurrence de facteur de longueur puissance de 2 comprise entre $l_{\min}$ et $l_{\max}$ l'ensemble des facteurs exactement identiques. Celui-ci permet également d'associer des facteurs comprenant des opérations de substitution de lexèmes (ce qui pallie à l'absence d'abstraction de lexèmes) ; cependant l'insertion ou la suppression de lexèmes n'est pas supportée. Afin de limiter la portée de la recherche, un filtrage est réalisé préalablement afin de sélectionner les fichiers partageant le plus de lexèmes communs.

SID [95] SID est un service web de recherche de similarité particulièrement destiné à la recherche de cas de plagiat au sein de jeux de projets d'étudiants. Les projets y sont comparés deux par deux afin d'évaluer une métrique de similarité s basée sur la complexité de Kolgomorov. Sont définies ainsi la complexité de Kolgomorov $K(p_i)$ pour un projet p_i correspondant à la quantité d'information théorique contenue dans ce projet ainsi que la complexité de Kolgomorov conditionnelle $K(p_i|p_j)$ qui est une mesure de l'information théorique nouvelle introduite par p_i connaissant p_j. La métrique de similarité est alors définie ainsi : $s(p_1, p_2) = \frac{K(p_1)-K(p_1|p_2)}{K(p_1p_2)}$ qui peut être assimilée à une métrique de type union. Afin d'estimer la complexité de Kolgomorov d'une séquence de lexèmes issue d'un projet, la taille de la séquence compressée par une méthode de compression est utilisée. Pour SID, une technique de compression de type Lempel-Ziv avec recherche de facteurs répétés sur une fenêtre de taille illimitée est utilisée. L'algorithme utilisé, qui n'est pas explicitement décrit, permet la mise en correspondance de séquences approchées avec quelques opérations d'édition.

A.3 Arbres de syntaxe

A.3.1 Autre approche algorithmique

CloneDigger [64] Après lexèmisation du code source, CloneDigger utilise une approche par antiunification afin de grouper des structures de code similaires. Dans un premier temps, toutes les instructions sont insérés dans des groupes, chaque groupe étant représenté par une schéma d'instruction antiunifié. Pour cela, une distance d'édition par rapport à chaque schéma de chaque groupe est calculée : l'instruction est insérée soit dans le groupe dont le schéma est le plus proche (le schéma évolue alors), soit dans un nouveau groupe. Dans un second temps, les instructions sont regroupées de nouveau en utilisant les schémas représentatifs définitifs. Les blocs englobant les instructions sont ensuite eux aussi regroupés par antiunification. Actuellement, les clones résultants sont reportés par une représentation de type diff (alignement global) caractère par caractère. Aucune métrique de similarité entre unités de compilation n'est calculée.

B
Le logiciel Plade

Nous présentons ici brièvement le logiciel Plade de recherche de similarité sur du code source que nous avons développé dans le cadre de notre travail. La majorité de ses composantes sont développées en Java 1.5. Il implante différentes techniques de recherche de correspondances dont la technique de factorisation sur fonctions de lexèmes introduite dans le chapitre 7 ainsi qu'une méthode de hachage et d'indexation de sous-arbres (chapitres 9 et 10) avec consolidation par extension des correspondances obtenues.

Plade adopte une architecture modulaire visant à séparer structures et algorithmes généraux des implantations plus spécifiques destinées à la recherche de similarité dans du code source. La mise en œuvre de procédés de pré-traitement du code afin d'obtenir des représentations spécifiques ou le post-traitement de correspondances trouvées doit être simple. Le support de nouveaux langages se veut également aisé.

B.1 Structures et algorithmes généralistes (*algostruct*)

La majorité des structures et algorithmes généralistes utiles pour la recherche de similarité sont déportés dans ce paquetage. Nous présentons quelques constituants essentiels de ce module :

- *algostruct.serialization* contient des primitives pour la sérialisation d'objets simples. Il est possible de définir ses propres classes de sérialisation par un code unique permettant une représentation binaire, textuelle ou XML.
- *algostruct.bptree* implante une structure de *B+-tree* stockable sur mémoire de masse. Elle utilise les possibilités de *algostruct.serialization* pour la sérialisation binaire des objets emmagasinés.
- *algostruct.graph* propose des structures de graphe ainsi que quelques algorithmes classiques afin de les parcourir, d'en extraire les composantes fortement connexes, de calculer des LCA de nœuds ou de réaliser certaines transformations (juxtaposition de graphes, regroupement de nœuds). Les graphes sont sérialisables sous forme immuable ou alors

stockés et dynamiquement modifiables par le biais de B+-tree. Ils sont visualisables sous forme textuelle, XML ou au format DOT. Une petite interface graphique Swing est également proposée. Un import XML de graphes est possible.
- *algostruct.{map, set, array, matrix}* implantent respectivement des structures de dictionnaire, d'ensemble, de tableau et de matrice avec possibilités d'import et d'export par sérialisation.
- *algostruct.align* propose des implantations d'algorithmes d'alignement global par programmation dynamique et d'alignement local inspiré de la méthode de Smith-Waterman avec coupure.
- *algostruct.suffixarray* définit des structures de table de suffixes, table de LCP, arbre des intervalles et graphe de farmax. Des algorithmes génériques pour le calcul de la table de suffixe et du graphe de farmax afférents sont présents.

Au cours de leur évolution, chacune de ces composantes pourrait avoir pour destinée de connaître un développement au sein d'une paquetage indépendant ou de voir leur utilisation remplacée par des solutions externes plus flexibles et performantes.

B.2 Noyau : représentations du code et référentiels de stockage (*plade.core*)

Le cœur de Plade est majoritairement constitué de structures et de référentiels pour le stockage de représentations du code. Un référentiel permettant l'organisation hiérarchique du code source géré est disponible ainsi qu'un référentiel stockant une forme compressée du code traité. Le code source doit ensuite pouvoir être représenté sous la forme de séquences de lexèmes ou arbres de syntaxe avec éventuels liens d'appel entre fonctions : des référentiels *ad-hoc* de stockage persistants utilisant des structures sérialisées de *algostruct* sont proposés. Un référentiel pour les types de nœuds manipulés est présent.

Un référentiel d'empreintes économe en mémoire utilisant *algostruct.bptree* est implanté pour les applications de hachage de k-grams de séquences de lexèmes et de sous-arbres.

B.3 Implantations d'algorithmiques spécifiques

B.3.1 Factorisation de fonctions de lexèmes (*plade.fact*)

Ce paquetage est une implantation de la méthode de factorisation de fonctions de projet expliquée au chapitre 7. Il utilise les structures de table de suffixes, de graphe de farmax ainsi que des structures intermédiaires implantées dans *algostruct*. Le graphe d'appel obtenu par fusion des graphes originels et externalisation des portions de code partagées est sérialisé suite à son calcul. Il peut être filtré sur des critères de multiplicité ou de volume des nœuds. Les graphes obtenus sont exploitables par *algostruct.graph* pour être représentés sous format XML ou DOT.

B.3.2 Indexation d'arbres de syntaxe et extension (*plade.fingertree*)

Ce module implante les méthodes de hachage de sous-arbres présentées au chapitre 9 par fonctions cryptographiques de hachage et fonctions polynomiales. Les sous-arbres peuvent ensuite être indexés selon plusieurs profils d'abstraction.

La base d'indexation est analysable pour retrouver les classes d'équivalence de sous-arbres. L'interrogation à partir d'un arbre requête est possible afin de retrouver les séquences d'arbres frères partagées entre l'arbre requête et les arbres indexés de la base, avec l'usage d'une structure d'indexation de suffixes. Les groupes de correspondances peuvent être transformées en 2-correspondances et faire l'objet d'un post-traitement afin de consolider les composantes proches dans l'arbre de syntaxe.

B.4 Interfaces utilisateurs

Des interfaces en ligne de commande sont disponibles afin de créer des environnements de travail comportant des référentiels de code ainsi que de représentations. À l'issue de l'application de méthodes algorithmiques de recherche de similarité, des référentiels de résultats peuvent être récupérés dans des formats textuels, binaires ou XML. Nous pouvons ainsi obtenir des graphes de correspondances ainsi que des matrices de similarité. Ces résultats peuvent ensuite être traités par un logiciel externe.

L'implantation des interfaces est facilitée par l'emploi du paquetage *factools* permettant d'expliciter et initialiser des objets arguments par une syntaxe déclarative. Une bibliothèque aide également au découpage des tâches algorithmiques avec gestion des dépendances entre-elles et découpage en quantums de calcul pour un ordonnancement sur plusieurs processeurs.

Nous proposons, afin de visualiser des jeux de correspondances et matrice de similarité, une interface simple de visualisation comprenant un tracé de carte de similarité de type *dotplot* pour une vision globale ainsi que le répérage contextuel dans le code source des correspondances. Des possibilités de filtrage des correspondances selon certains critères (unités structurelles d'appartenance, multiplicité, volume, score d'exactitude...) sont possibles. Cette interface n'offre pour l'heure qu'un moyen d'exploration statique des correspondances : il pourrait être intéressant d'y introduire un moyen de lancer interactivement des processus de recherche avec des paramètres spécifiques indiqués par l'utilisateur.

B.5 Paquetages langage-spécifiques

Les méthodes de recherche de similarité implantées par Plade se veulent génériques et ne dépendent d'aucune contrainte liée à un langage spécifique. Il est ainsi possible de développer assez rapidement des paquetages d'analyse lexicale et/ou syntaxique adaptés à des langages particuliers. Ces paquetages ont pour seule contrainte de produire des séquences ou arbres étiquetés par des informations d'origine dans le code source originel. Chaque lexème ou nœud est représenté par un type paramétré. Les types peuvent être organisés sous forme de graphe afin de gérer facilement ensuite différents profils d'abstraction. Par exemple en Java, les nœuds correspondants aux types primitifs *int*, *long* peuvent être descendants du nœud *number*, lui

même descendant de *type* : selon le profil d'abstraction employé tous les nœuds descendants de *type* pourront être distingués, ou au contraire abstraits par l'ancêtre *type*. Il peut être envisageable d'utiliser une taxonomie des nœuds ou lexèmes commune pour des langages proches afin de faciliter la recherche de similarité inter-langage.

Actuellement des lexémiseurs pour les langages Java et C générés à partir de lexiques JFlex sont implantés. Une liaison vers les analyseurs syntaxiques d'Eclipse pour le langage Java (JDT) et les langages C et C++ sont également proposés.

B.6 Évolutions et perspectives de développement

Plade est à l'heure de la rédaction de ce document encore en phase intense de développement. Certaines méthodes algorithmiques restent à implanter de même que des optimisations peuvent être apportées à du code existant. L'objectif principal est d'obtenir un outil pratique pour la recherche et l'exploration de correspondances avec une bonne résistance aux opérations d'édition. Une utilisation dans un contexte de recherche de similarité entre projets soumis par des étudiants paraît prometteuse ; particulièrement si celle-ci est couplée à l'usage d'une plate-forme de rendu et de tests automatisés de projets telle que Subjects [136], développée par des étudiants de l'UFR Ingénieurs 2000 de l'Université Paris-Est Marne-la-Vallée. Un emploi dans une optique de réingénierie de projet paraît également crédible. Mais Plade reste avant tout un outil de test et d'évaluation de méthodes de recherche de similarité : cela nous conduira à mettre au point des moyens de comparaison d'outils et méthodes hétérogènes. Il est prévu que dans un futur proche, une version mature de Plade soit rendue disponible sous une licence OpenSource de type Affero GPL.

C

Bibliographie thématique

Publications personnelles

[1] M. Chilowicz, É. Duris, and G. Roussel. Syntax tree fingerprinting : a foundation for source code similarity detection. Technical report, 2009. URL `http://igm.univ-mlv.fr/~chilowi/research/syntax_tree_fingerprinting/`.

[2] M. Chilowicz, É. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *International Conference on Program Comprehension*, pages 243–247, Vancouver, Canada, 2009. URL `http://igm.univ-mlv.fr/~chilowi/research/syntax_tree_fingerprinting/`.

[3] M. Chilowicz, É. Duris, and G. Roussel. Finding similarities in source code through factorization. *Electronic Notes in Theoretical Computer Science*, 238(5) :47–62, 2009. doi : 10.1016/j.entcs.2009.09.040. URL `http://igm.univ-mlv.fr/~chilowi/research/factorization/`.

[4] M. Chilowicz, É. Duris, and G. Roussel. Towards a multi-scale approach for source code approximate match report. In *International Workshop on Software Clones*, pages 89–90, Cape Town, South Africa, 2010. ACM. ISBN 978-1-60558-980-0.

Algorithmique générale

[5] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2) :221–242, 1993. doi : 10.1137/0222017. URL `http://link.aip.org/link/?SMJ/22/221/1`.

[6] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2) :121–137, 1979. ISSN 0360-0300. doi : 10.1145/356770.356776. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.6637&rep=rep1&type=pdf`.

[7] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13 (2) :94–102, 1970. ISSN 0001-0782. doi : 10.1145/362007.362035.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996. URL `http://www.cs.ualberta.ca/~joerg/papers/KDD-96_final.pdf`.

[9] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, 2007. doi : 10.1007/978-3-540-74450-4_41. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.5421&rep=rep1&type=pdf`.

[10] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3) :214–230, 1993. ISSN 0098-5589. doi : http://dx.doi.org/10.1109/32.221135. URL `http://www.graphviz.org/Documentation/TSE93.pdf`.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[12] R. Hamming. Error detecting and error correcting codes. *Bell System Tech Journal*, 26 :147 –160, 1950. URL `http://www.lee.eng.uerj.br/~gil/redesII/hamming.pdf`.

[13] M. Kowaluk and A. Lingas. LCA queries in directed acyclic graphs. *Automata, Languages and Programming*, pages 241–248, 2005. doi : 10.1007/11523468_20. URL `http://www.cs.nctu.edu.tw/~tjshen/doc/fulltext.pdf`.

[14] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10 :707–10, 1966. URL `http://sascha.geekheim.de/wp-content/uploads/2006/04/levenshtein.pdf`.

[15] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 2008. ISBN 0387339981, 9780387339986. URL `http://books.google.fr/books?id=25fue3UYDN0C&lpg=PP1&pg=PP1#v=onepage&q=&f=false`.

[16] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28 :129–137, 1982. URL `http://www.cs.toronto.edu/~roweis/csc2515-2006/readings/lloyd57.pdf`.

[17] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows Wheeler Transform and applications to sequence comparison and data compression. *Combinatorial Pattern Matching*, pages 178 – 189, 2005. doi : 10.1007/11496656_16.

[18] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74 :358–366, 1953.

[19] J. Sima and S. E. Schaeffer. On the NP-completeness of some graph cluster measures. *SOFSEM 2006 : Theory and Practice of Computer Science*, pages 530–537, 2006. URL `http://arxiv.org/abs/cs.CC/0506100`.

[20] B. Stein and O. Niggemann. On the nature of structure and its identification. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 122–134. Springer-Verlag, 1999. ISBN 3-540-66731-8. URL `http://www-sst.informatik.tu-cottbus.de/~db/doc/People/LNCS/papers/16650122.pdf`.

[21] R. Tarjan. Depth-first search and linear graph algorithms. In *SWAT '71 : Proceedings of the 12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121. IEEE Computer Society, 1971. doi : http://dx.doi.org/10.1109/SWAT.1971.10.

[22] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13 :31–46, 1987. URL `http://acl.ldc.upenn.edu/J/j87/J87-1004.pdf`.

[23] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2) :189 – 208, 1967. ISSN 0019-9958. doi : DOI:10.1016/S0019-9958(67)80007-X.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions On Information Theory*, 23(3) :337–343, 1977. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.8921&rep=rep1&type=pdf`.

Structures d'indexation de suffixes

[25] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2004. URL `http://www.fli-leibniz.de/www_bioc/journal_club/AboKurOhl2004.pdf`.

[26] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, Atlanta, Georgia, United States, 1996. Society for Industrial and Applied Mathematics. ISBN 0-89871-366-8.

[27] R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In *Proceedings of the 14th Annual Symposium Combinatorial on Pattern Matching*, pages 70–82, Morelia, Michoacán, Mexico, 2003. URL `http://www.cs.ucr.edu/~stelo/cpm/cpm03/Clifford.pdf`.

[28] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, page 137. IEEE Computer Society, 1997. ISBN 0-8186-8197-7. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.555&rep=rep1&type=pdf`.

[29] M. Gallé, P. Peterlongo, and F. Coste. In-place Update of Suffix Array while Recoding Words. In *Proceedings of the Prague Stringology Conference*, pages 54–67, 2008. URL `http://hal.inria.fr/inria-00327582/en/`.

[30] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner : A unifying view of linear-time suffix tree construction. *Algorithmica*, 19 :331–353, 1997. URL `http://www.zbh.uni-hamburg.de/staff/kurtz/papers/GieKur1997.pdf`.

[31] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955, 2003. URL `http://www.mpi-sb.mpg.de/~juha/publications/icalp03-final.ps.gz`.

[32] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest common-prefix computation in suffix arrays and its applications. In *12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer-Verlag, 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.8221&rep=rep1&type=pdf`.

[33] S. Kurtz. Reducing the space requirement of suffix trees. *Software : Practice and Experience*, 29(13) :1149–1171, 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4903&rep=rep1&type=pdf`.

[34] J. Kärkkäinen. Suffix cactus : A cross between suffix tree and suffix array. In *Combinatorial Pattern Matching*, pages 191–204. Springer, 1995.

[35] U. Manber and G. Myers. *Suffix arrays : a new method for on-line string searches*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990. URL `http://www.cs.arizona.edu/people/udi/suffix.ps`.

[36] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2) :262–272, 1976. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.130.8022&rep=rep1&type=pdf`.

[37] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2) :4, 2007. ISSN 0360-0300. doi : 10.1145/1242471.1242472. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.5990&rep=rep1&type=pdf`.

[38] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2) :241 – 257, 2010. ISSN 1570-8667. doi : DOI:10.1016/j.jda.2009.02.007. URL `http://www-igm.univ-mlv.fr/~lecroq/articles/jda2009.pdf`.

[39] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software : Practice and Experience*, 37(3) :309–329, 2007. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.9152&rep=rep1&type=pdf`.

[40] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, 1995. URL `http://cs.helsinki.fi/u/ukkonen/SuffixT1.ps`.

[41] P. Weiner. Linear pattern matching algorithm. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Alignement de séquences

[42] G. Cobena. Gestion des changements pour les données semi-structurées du web, 2003. URL `ftp://ftp.inria.fr/INRIA/publication/Theses/TU-0789.pdf`. Thèse de doctorat de l'École Polytechnique.

[43] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *LNCS : Automata, Languages and Programming*, pages 146–157, 2007. doi : 10.1007/978-3-540-73420-8_15. URL `http://www.mit.edu/~brossman/ted.pdf`.

[44] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6) :341–343, 1975. URL `http://www.dat.ruc.dk/~keld/teaching/algoritmedesign_f03/Artikler/05/Hirschberg75.pdf`.

[45] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, pages 91–102, London, UK, 1998. Springer-Verlag. ISBN 3-540-64848-8.

[46] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BioMed Central Bioinformatics*, 9(Suppl 2) :S10, 2008.

[47] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1) :195–197, 1981. ISSN 0022-2836. URL `http://gel.ym.edu.tw/~chc/AB_papers/03.pdf`.

[48] H. Touzet and S. Dulucq. Analysis of tree edit distance algorithms. In *In Proceedings of the 14th annual symposium on Combinatorial Pattern Matching*, pages 83–95. Springer-Verlag, 2003. URL `http://www.lifl.fr/~touzet/Publications/cpmtouzet.pdf`.

[49] M. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons* 1. *Journal of Molecular Biology*, 197(4) : 723–728, 1987. URL `http://www.cmb.usc.edu/papers/msw_papers/msw-079.pdf`.

[50] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6) :1245–1262, 1989. ISSN 0097-5397. URL `http://www.cs.ust.hk/mjg_lib/Library/ZhSh89.pdf`.

Hachage

[51] D. Eastlake and P. Jones. RFC 3174 : US secure hash algorithm 1 (SHA1), 2001. URL `http://www.ietf.org/rfc/rfc3174.txt`.

[52] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7.

[53] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2) :249–260, 1987. ISSN 0018-8646. URL `http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf`.

[54] D. Knuth. *The Art of Computer Programming, Volume 3 : Sorting and Searching, Third Edition.* Addison-Wesley, 1997. ISBN 0-201-89685-0.

[55] R. Rivest. RFC 1321 : The MD5 message-digest algorithm, 1992. URL http://www.ietf.org/rfc/rfc1321.txt.

[56] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing : Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003. URL http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf.

[57] A. F. Webster and S. E. Tavares. On the design of s-boxes. In *Proceedings of the Advances in Cryptology conference*, page 523. Springer, 1986. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.8139&rep=rep1&type=ps.

Méthodes et outils de recherche de similarité sur du code source

Descriptifs de méthodes de recherche de similarité

[58] A. Ahtiainen, S. Surakka, and M. Rahikainen. Plaggie : GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 141–142. ACM, 2006. doi : 10.1145/1315803.1315831. URL http://user.it.uu.se/~mattiasw/papers/kc_preproceedings_2006.pdf#page=134.

[59] G. Antoniol, G. Casazza, M. D. Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2) :181 – 196, 2001. ISSN 0164-1212. doi : DOI:10.1016/S0164-1212(01)00061-9.

[60] M. Balazinska, E. Merlo, M. Dagenais, B. Lage, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. volume 0, page 98. IEEE Computer Society, 2000. doi : 10.1109/WCRE.2000.891457.

[61] H. A. Basit and S. Jarzabek. A case for structural clones. In *Proceedings of the Third International Workshop on Software Clones*, pages 7 – 11, 2009. URL http://www.informatik.uni-bremen.de/st/IWSC/basit.pdf.

[62] H. A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering*, 35 :497–514, 2009. ISSN 0098-5589. doi : 10.1109/TSE.2009.16. URL http://www.comp.nus.edu.sg/~stan/PAPERS/CloneMiner%20TSE.pdf.

[63] 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*, page 368. IEEE Computer Society, 1998. ISBN 0-8186-8779-7. URL http://www.semdesigns.com/Company/Publications/ICSM98.pdf.

[64] P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the International Workshop on Software Clones*, pages 22 – 27, 2009.

[65] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software*

Maintenance, page 109, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0016-1.

[66] T. Imai, Y. Kataoka, and T. Fukaya. Evaluating software maintenance cost using functional redundancy metrics. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, page 299. IEEE Computer Society, 2002. doi : 10.1109/CMPSAC.2002.1045018.

[67] R. Irving. Plagiarism and collusion detection using the Smith-Waterman algorithm. Technical report, University of Glasgow, 2004. URL `http://www.dcs.gla.ac.uk/publications/PAPERS/7444/TR-2004-164.pdf`.

[68] P. Jablonski and D. Hou. Cren : a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA workshop on Eclipse technology eXchange*, pages 16–20. ACM, 2007. ISBN 978-1-60558-015-9. doi : 10.1145/1328279.1328283. URL `http://www.cs.mcgill.ca/~martin/etx2007/papers/4.pdf`.

[69] A. Jadalla and A. Elnagar. Pde4java : Plagiarism detection engine for java source code : a clustering approach. *International Journal of Business Intelligence and Data Mining*, pages 121–135, 2008. doi : 10.1504/IJBIDM.2008.020514. URL `http://repository.gunadarma.ac.id:8000/iiWAS2007(167-175)_6.pdf`.

[70] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering*, pages 96–105. IEEE-CS, 2007. ISBN 0-7695-2828-7. URL `http://groups.csail.mit.edu/pag/reading-group/jiang07deckard.pdf`.

[71] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder : A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7) :654–670, 2002. ISSN 0098-5589. doi : 10.1109/TSE.2002.1019480. URL `http://sel.ist.osaka-u.ac.jp/~kamiya/publication/tse200207.pdf`.

[72] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE Computer Society, 2006. ISBN 0-7695-2719-1. URL `http://www.informatik.uni-bremen.de/st/papers/astclones-wcre06.pdf`.

[73] T. Lavoie, M. Eilers-Smith, and E. Merlo. Challenging cloning related problems with gpu-based algorithms. In *Proceedings of the 4th International Workshop on Software Clones*, pages 25–32. ACM, 2010. ISBN 978-1-60558-980-0. doi : 10.1145/1808901.1808905.

[74] L. Moussiades and A. Vakali. PDetect : A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6) :651–661, 2005. ISSN 0010-4620. doi : 10.1093/comjnl/bxh119. URL `http://iridium.csd.auth.gr/~kalaitzv/wiki/images/b/b5/Compj05.pdf`.

[75] K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, 8(4) :30–41, 1976. ISSN 0097-8418. doi : 10.1145/382222.382462.

[76] L. P. Prechelt, U. Karlsruhe, and G. Malpohl. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8 :1016–1038, 2000. URL http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf.

[77] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the Third International Workshop on Software Clones*, pages 28 – 34, 2009.

[78] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, pages 679–684. ACM, 2006. ISBN 1-59593-315-8. URL http://www.cis.uab.edu/gray/Pubs/acmse-2006-robert.pdf.

[79] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Proceedings of the Asia-Pacific Software Engineering Conference*, page 327. IEEE Computer Society, 2002. ISBN 0-7695-1850-8.

[80] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 126–130. IEEE, 2003.

[81] R. Wettel and R. Marinescu. Archeology of code duplication : Recovering duplication chains from small duplication fragments. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, page 63. IEEE Computer Society, 2005. ISBN 0-7695-2453-2. doi : 10.1109/SYNASC.2005.20. URL http://www.loose.upt.ro/download/papers/wettel_marinescu-synasc2005.pdf.

[82] M. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Technical report, Dept. of Computer Science, University of Sydney, 1993. URL http://www.pam1.bcs.uwa.edu.au/~michaelw/ftp/doc/RKR_GST.ps.

[83] M. Wise. Neweyes : A system for comparing biological sequences using the Running Karp-Rabin Greedy String-Tiling algorithm. In *Third International Conference on Intelligent Systems for Molecular Biology*, pages 393–401. AAAI Press, 1995. URL http://www.it.usyd.edu.au/research/tr/tr463.pdf.

[84] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In F. Bomarius and S. Komi-Sirviö, editors, *Product Focused Software Process Improvement*, volume 3547 of *Lecture Notes in Computer Science*, pages 530–544. Springer Berlin / Heidelberg, 2005. URL http://sel.ist.osaka-u.ac.jp/~lab-db/betuzuri/archive/369/369.pdf.

[85] B. Zeidman. Software forensics tools enter the courtroom. *IEEE Spectrum*, (10), 2010. URL http://spectrum.ieee.org/computing/software/software-forensics-tools-enter-the-courtroom.

Comparatifs de méthodes de recherche de similarité

[86] 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. URL http://plg.uwaterloo.ca/~migod/846/papers/bellon-tse07.pdf.

[87] C. K. Roy and J. R. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, volume 2008, pages 153–162. IEEE, 2008. URL http://www.cs.queensu.ca/home/cordy/Papers/RC_ICPC08_ScenarioEval.pdf.

[88] S. Schulze, S. Apel, and C. Kästner. Code clones in feature-oriented software product lines. In *Generative Programming and Component Engineering (GPCE)*. ACM, 2010. URL http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/SAK10.pdf.

Outils de recherche de similarité

[89] CCFinder-X. URL http://www.ccfinder.net/.

[90] CloneDigger. URL http://clonedigger.sourceforge.net/.

[91] CloneDr. URL http://www.semdesigns.com/Products/Clone/.

[92] Copy Paste Duplication (PMD). URL http://pmd.sourceforge.net/cpd.html.

[93] JPlag. URL https://www.ipd.uni-karlsruhe.de/jplag/.

[94] Moss. URL http://theory.stanford.edu/~aiken/moss.

[95] Sid. URL http://genome.math.uwaterloo.ca/SID/.

[96] Unique. URL http://sourceforge.net/projects/unique/.

Études de duplications intra- ou inter-projets

[97] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209. IEEE Computer Society, 2004. ISBN 0-7695-2213-0. URL http://homepages.cwi.nl/~bruntink/papers/icsm04.pdf.

[98] J. Harder and N. Göde. Modeling clone evolution. In *Proceedings of the Third International Workshop on Software Clones*, pages 17 – 21, Kaiserslautern, Germany, 2009. URL http://www.informatik.uni-bremen.de/st/IWSC/harder.pdf.

[99] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28. IEEE Computer Society, 2006. ISBN 0-7695-2719-1. doi : http://dx.doi.org/10.1109/WCRE.2006.1. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.771&rep=rep1&type=pdf.

[100] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weißgerber. Subjectivity in clone judgment : Can we ever agree ? In *Duplication, Redundancy, and Similarity in Software*, 2007.

[101] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 7. IEEE Computer Society, 2007. ISBN 0-7695-2961-5. doi : http://dx.doi.org/10.1109/FLOSS.2007.10. URL `http://mockus.us/papers/ossreuse.pdf`.

[102] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 81–90. IEEE Computer Society, 2008. ISBN 978-0-7695-3429-9. doi : http://dx.doi.org/10.1109/WCRE.2008.54. URL `http://research.cs.queensu.ca/home/cordy/Papers/RC_WCRE08_OSclones.pdf`.

[103] S. Uchida, T. Kamiya, A. Monden, K.-I. Matsumoto, N. Ohsugi, and H. Kudo. Software analysis by code clones in Open Source programs. *Journal of Computer Information Systems*, XLV(3) :1–11, 2005.

[104] C. A. Villano, G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the linux kernel. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001. URL `http://rcost.unisannio.it/rcost_www/mdipenta/papers/scam2001.pdf`.

Analyse statique de code source

Publications concernant l'analyse statique

[105] Générateur d'analyseur syntaxique javacc. URL `https://javacc.dev.java.net/`.

[106] J. Cervelle, R. Forax, and G. Roussel. Tatoo : an innovative parser generator. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 13–20. ACM, 2006. ISBN 3-939352-05-5. doi : 10.1145/1168054.1168057. URL `http://weblogs.java.net/blog/forax/archive/paper.pdf`.

[107] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, 1997. URL `http://www.researchspace.auckland.ac.nz/bitstream/2292/3491/2/TR148.pdf`.

[108] M. Halstead. *Elements of Software Science*. Elsevier Science, 1977. ISBN 978-0444002051. URL `http://books.google.fr/books?lr=&as_brr=0&q=isbn%3A9780444002051`.

[109] D. Low. Protecting java code via code obfuscation. *Crossroads*, 4(3) :21–23, 1998. ISSN 1528-4972. doi : 10.1145/332084.332092.

[110] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, page 407. IEEE Computer Society Press, 1976. URL `http://www.literateprogramming.com/mccabe.pdf`.

[111] N. A. Naeem, M. Batchelder, and L. Hendren. Metrics for measuring the effectiveness of decompilers and obfuscators. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 253–258. IEEE Computer Society, 2007. ISBN

0-7695-2860-0. doi : 10.1109/ICPC.2007.27. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.3820&rep=rep1&type=pdf.

[112] T. J. Parr and R. W. Quong. Antlr : a predicated-ll(k) parser generator. *Software Practice Experience*, 25(7) :789–810, 1995. ISSN 0038-0644. doi : 10.1002/spe.4380250705. URL https://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol25/issue7/spe964tp.pdf.

[113] G. Wroblewski. General method of program code obfuscation. Technical report, 2002. URL http://www.mysz.org/papers/147sp.pdf.

Outils d'analyse statique

[114] Générateur d'analyseur syntaxique LALR(1) Bison. URL http://www.gnu.org/software/bison/.

[115] Coverity Prevent. URL http://www.coverity.com/products/static-analysis.html.

[116] Générateur d'analyseur syntaxique Cup. URL http://www2.cs.tum.edu/projects/cup/.

[117] Générateur d'analyseur lexical C Flex. URL http://flex.sourceforge.net/.

[118] Générateur d'analyseur lexical Java JLex. URL http://www.cs.princeton.edu/~appel/modern/java/JLex/.

[119] Générateur d'analyseur syntaxique SableCC. URL http://sablecc.org/.

[120] Compilateur SmartEiffel. URL http://smarteiffel.loria.fr/.

[121] Langage de réécriture d'arbres Stratego. URL http://strategoxt.org/.

[122] Langage de réécriture d'arbres TOM. URL http://tom.loria.fr/.

[123] Langage de réécriture d'arbres TXL. URL http://www.txl.ca/.

Visualisation de relations de duplication

[124] C. Kapser and M. W. Godfrey. Improved tool support for the investigation of duplication in software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 305–314. IEEE Computer Society, 2005. ISBN 0-7695-2368-4. doi : http://dx.doi.org/10.1109/ICSM.2005.52. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.2515&rep=rep1&type=pdf.

[125] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 100–109. IEEE Computer Society, 2004. ISBN 0-7695-2243-2. URL http://scg.iam.unibe.ch/archive/papers/Rieg04bWCRE2004ClonesVisualization.pdf.

Divers

Publications diverses (filigranage, stylométrie...)

[126] I. J. Cox, S. Member, J. Kilian, F. T. Leighton, and T. Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing*, 6 :1673–1687, 1997. URL http://www.comlab.uniroma3.it/multimedia/coxspectr.

[127] E. W. Dijkstra. Letters to the editor : go to statement considered harmful. *Commun. ACM*, 11(3) :147–148, 1968. ISSN 0001-0782. doi : 10.1145/362929.362947.

[128] T. F.J. Neural network applications in stylometry : The federalist papers. *Computers and the Humanities*, 30 :1–10(10), 1996. URL http://www.ingentaconnect.com/content/klu/chum/1996/00000030/00000001/00106225.

[129] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, pages 893–896. ACM, 2006. ISBN 1-59593-375-1. doi : 10.1145/1134285.1134445. URL http://www.icsd.aegean.gr/lecturers/Stamatatos/papers/ICSE06.pdf.

[130] G. Kacmarcik and M. Gamon. Obfuscating document stylometry to preserve author anonymity. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 444–451, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

[131] I. Krsul and E. H. Spafford. Authorship analysis : identifying the author of a program. *Computers & Security*, 16(3) :233 – 257, 1997. ISSN 0167-4048. doi : 10.1016/S0167-4048(97)00005-9. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.2790&rep=rep1&type=pdf.

[132] K. Monostori, A. Zaslavsky, and H. Schmidt. MatchDetectReveal : finding overlapping and similar digital documents. In *Proceedings of the International Conference on Challenges of Information Technology Management in the 21st century*, pages 955–957, Hershey, PA, USA, 2000. IGI Publishing. ISBN 1-878-28984-5. URL http://www.csse.monash.edu/projects/MDR/papers/irma2000_mdr_monostori.pdf.

[133] M. D. Swanson, B. Zhu, A. H. Tewfik, and L. Boney. Robust audio watermarking using perceptual masking. *Signal Process.*, 66(3) :337–355, 1998. ISSN 0165-1684. doi : 10.1016/S0165-1684(98)00014-0. URL http://speech.csie.ntu.edu.tw/previous_version/paper/SP98-5-4.pdf.

Logiciels

[134] Concurrent Versions System. URL http://www.nongnu.org/cvs/.

[135] Mercurial. URL http://mercurial.selenic.com/.

[136] Plate-forme de rendu de projets d'étudiants Subjects, 2010. URL http://sourceforge.net/projects/subjects/.

[137] Subversion. URL http://subversion.tigris.org/.

Sociétés et organisations

[138] Black Duck Software. URL http://www.blackducksoftware.com/.

[139] Palamida. URL http://www.palamida.com/.