Une introduction à la recherche de correspondances dans du code source

Michel Chilowicz
Email: michel.chilowicz@univ-paris-est.fr
Site web: http://igm.univ-mlv.fr/~chilowi/

UNIVERSITÉ PARIS-EST

Laboratoire d'Informatique Gaspard-Monge
Université Paris-Est

Tree Masters (6 décembre 2010)

Plan

Pourquoi rechercher des correspondances dans du code source ?

Deux contextes principaux :

- Recherche intra-projet $\longrightarrow$ meilleure maintenance du code
- Recherche inter-projets $\longrightarrow$ copie légitime ou non de code

Duplications au sein d'un projet

Origine

- Morceaux de code copiés-collés
- Opérations d'éditions pour adaptation à un nouveau contexte

Vie des duplications

- Parfois, solution temporaire pour tester une variante : factorisation ensuite
- D'autres fois conservés durablement : pourquoi ?
  - Par difficulté de généricisation
    (e.g. duplications pour architectures différentes)
  - Par nécessité du langage
    (e.g. réécriture pour différents types primitifs en Java)
  - Ou tout simplement par oisiveté du développeur

Processus de factorisation des clones

1. Recherche de correspondances
2. Factorisation en nouvelles fonctions, classes des correspondances
   $\longrightarrow$ généricisation automatique délicate

Gestion de référentiel évolutif de code

Objectif

Rechercher les similarités pour mieux représenter les *deltas* entre versions

Utilité pratique

- Visualisation des modifications plus adaptée qu'un *diff* ligne par ligne classique
- Meilleure compression par codage de plus haut niveau
- Compréhension du cycle de vie des clones entre branches

Spécificités des duplications inter-projets

Légitimité des copies

- Copies légitimes : entre logiciels libres
  (e.g. BSD $\rightarrow$ GPL, GPL $\rightarrow$ GPL...)
- Copies illégitimes
  - Par contrefaçon : entre logiciels propriétaires et libres
    (e.g. propriétaire $\rightarrow$ libre, GPL $\rightarrow$ propriétaire, GPL $\rightarrow$ BSD)
  - Dans un contexte académique : projets d'étudiants

Obfuscation

Opérations d'édition importantes entre exemplaires

Rappel favorisé au détriment de la précision

Opérations d'obfuscation (partiellement gérables statiquement)

- Modification de formatage
- Renommage d'identificateurs
- Remplacements par types compatibles (e.g. short $\rightarrow$ int)
- Réécriture neutre d'expressions (e.g. $a \rightarrow a * 1 + 0$)
- Ajout/suppression de code inutile
- Modification de structures de contrôle
- Déplacement de code
- Changements fonctionnels : développement, factorisation
- Traduction inter-langage (e.g. Java $\rightarrow$ C#)

Obfuscation dynamique

Exploitation des capacités d'introspection du langage pour :

- Changer du code obscurci en code original
- Charger du code externe

$\longrightarrow$ ralentit l'exécution

Parade : recherche de similarité sur les traces d'exécution

Changements algorithmiques

- Nécessite la compréhension du code original et sa réécriture
  $\longrightarrow$ incitation du plagieur à adopter ce comportement
- Équivalence algorithmique : généralement indécidable

Inégalité souhaitable

$$\left(\begin{array}{l}\text{effort}\\ \text{d'obfuscation}\end{array}\right) \geq \left(\begin{array}{l}\text{effort de compréhension}\\ \text{et réécriture du code}\end{array}\right)$$

Usage d'un outil de recherche de similarité
$\longrightarrow$ effort d'obfuscation $\nearrow$

Problématiques de l'analyse statique

Objectif : rechercher des correspondances et les représenter

- Quelles représentations intermédiaires du code source utiliser ?
- Comment définir objectivement une correspondance ?
- Quelles méthodes algorithmiques employer ?
- Comment synthétiser et représenter agréablement les correspondances ?

Représentations du code source

- Code d'origine considéré :
  - Code source brut
  - Code objet compilé
- Chaîne de lexèmes après analyse lexicale
- Chaîne de méta-lexèmes
- Arbre de syntaxe après analyse syntaxique
- Graphe de dépendances après résolution du flux d'exécution

Lexémisation

Code source brut

```
int somme(int[] t){
    int s = 0;
    for (int i=0; i < t.length; i++) s += t[i];
    return s;
}
```

Chaîne de lexèmes

```
int funcname lpar int [] id rpar lbrace
int id eq litteral semi
for lpar int id eq litteral semi id lt id dotlength semi id
postinc lpar id addassign id lbracket id rbracket semi
return id semi
rbrace
```

Meta-lexémisation

Code source brut

```
int somme(int[] t){
    int s = 0;
    for (int i=0; i < t.length; i++) s += t[i];
    return s;
}
```

Méta-lexèmes

Alphabet : k-uplets de lexèmes primitifs

Exemple pour $k = 4$:

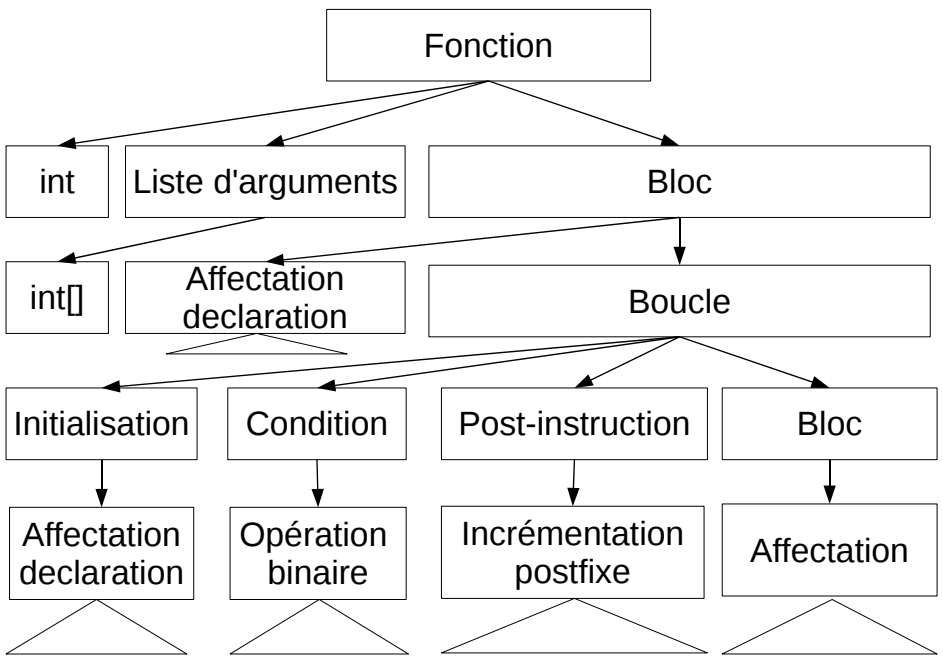
	int	funcname	lpar	int	[]	id	rpar
1	<int	funcname	lpar	int>			
2		<funcname	lpar	int	[]>		
3			<lpar	int	[]	id>	
4				<int	[]	id	rpar>
...					...		

Analyse syntaxique

Code source brut

```
int somme(int[] t){
    int s = 0;
    for (int i=0; i < t.length; i++) s += t[i];
    return s;
}
```

Arbre de syntaxe

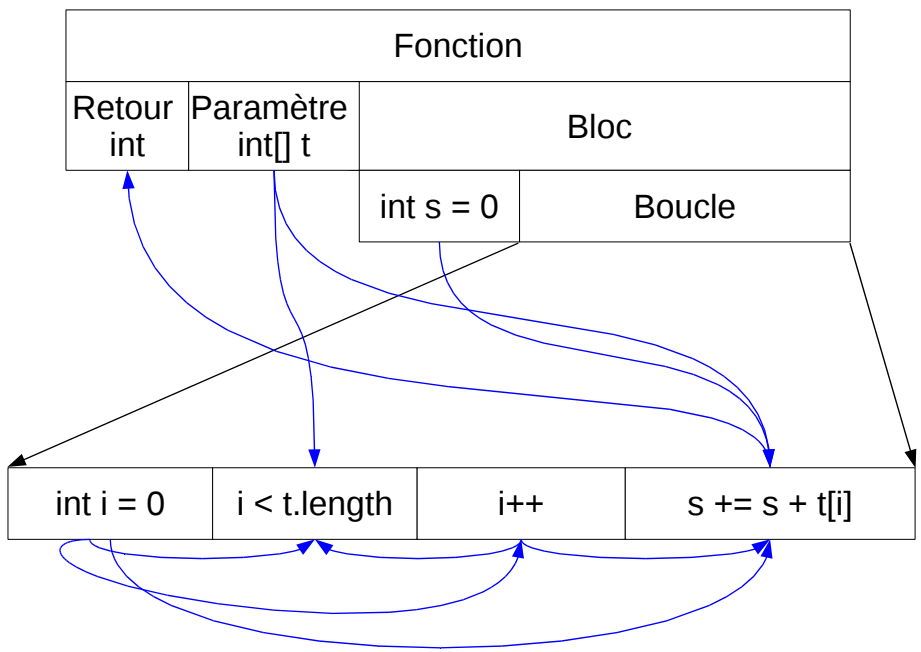


Résolution de dépendances

Code source brut

```
int somme(int[] t){
    int s = 0;
    for (int i=0; i < t.length; i++) s += t[i];
    return s;
}
```

Graphe de dépendances



Renommage d'identificateurs, changement de types

- Effet : substitution de lexèmes ou feuilles
- Proposition : abstraction des identificateurs et types

Abstraction quasiment systématiquement utilisée

Insertion et suppression de code inutile

Chaîne de lexèmes

- Effet : ajout/suppression de sous-chaînes de lexèmes
- Proposition : raccordement de correspondances exactes proches

Arbre de syntaxe

- Effet : insertion d'un sous-arbre ou de sous-arbres consécutifs frères
- **Proposition** : consolidation des sous-arbres spatialement proches

Graphe de dépendances

Mise en évidence du code non atteignable par analyse statique

Réécriture d'expressions

Chaînes de lexèmes

- Effet : modifications locales, insertion/suppression de quelques lexèmes, transpositions possibles
- Proposition : correspondances approchées par alignement des chaînes (programmation dynamique)

Arbre de syntaxe

- Effet : structure de haut niveau inchangée, modification de petits sous-arbres
- Propositions :
  - Comparaison d'arbres par distance d'édition : coûteux
  - Abstraction de petits sous-arbres : perte de précision

Transposition de code

Approche générale

1. Rechercher des correspondances exactes ou approchées
2. Assembler les correspondances permutées

Arbre de syntaxe

Normalisation de l'ordre de certains sous-arbres frères (fonctions, opérandes...)

Processus de recherche de similarité

1. Obtenir la représentation brute
2. L'abstraire et la normaliser pour la rendre insensible à certaines opérations d'obfuscation
3. Éventuellement la filtrer pour réduire la complexité (conservation des éléments les plus spécifiques)
4. Recherche de correspondances sur la représentation transformée
   - Exactes (indexation de suffixes, méta-lexémisation progressive)
   - Approchées (alignement local)
5. Consolidation des correspondances proches

Méthodes de recherche

- Sur une paire de projets
  $\longrightarrow$ recherche de correspondances approchées possible
- Sur un jeu fixe de projets
- Sur une base évolutive de projets
  $\longrightarrow$ recherche de k-correspondances exactes

Comparaison de paires de projets lexémisés

À la recherche de sous-chaînes partagées

Tuilage glouton à base de méta-lexémisation progressive (RKR-GST) [Wise1993]
Complexité expérimentale linéaire

À la recherche de sous-séquences partagées (fossés)

Alignement local (Smith-Waterman) par programmation dynamique [Irving2004]
Complexité : $\Omega(n^2)$

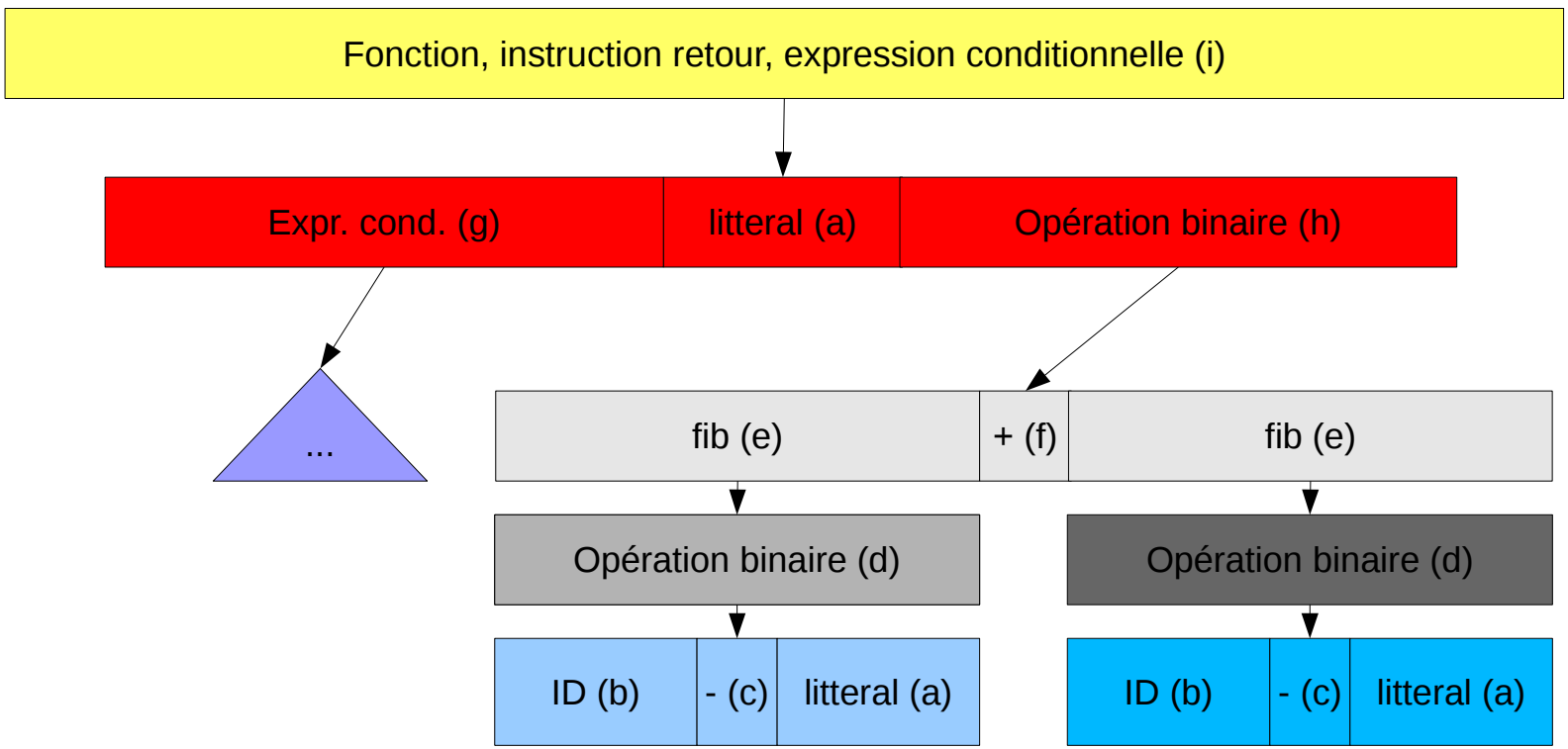
Des arbres de syntaxe aux chaîne de lexèmes

- Sous-arbre = valeur de hachage
- Considération des chaînes de sous-arbres frères en tant que chaînes de lexèmes pour la recherche

Exemple : un arbre de syntaxe et ses chaînes de sous-arbres frères

`int fib(n) { return (n ≤ 2)?1:(fib(n-1) + fib(n-2)); }`

Chaînes considérées : $\{i, gah, efe, d, bca, \ldots\}$



Recherche de k-correspondances exactes

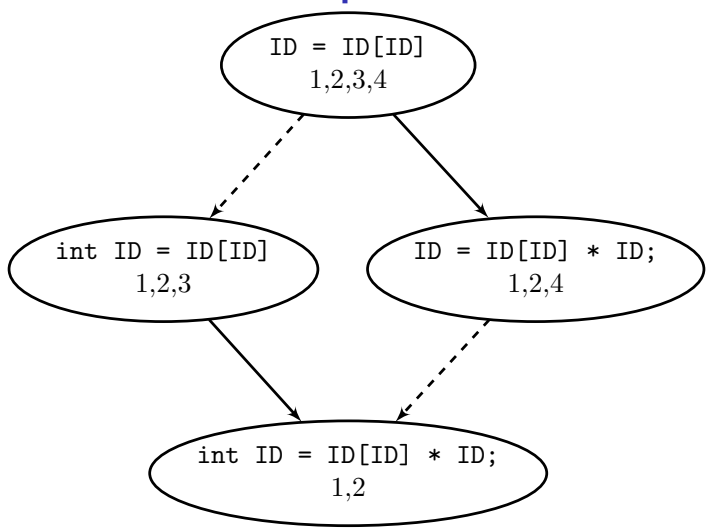
- Méthode : indexer les suffixes de chaînes de lexèmes ou chaînes de sous-arbres frères
- Pour l'indexation : utilisation de table de suffixes, arbre de {suffixes, intervalles}
- Modélisation d'éventuelles relations d'imbrication entre correspondances
- Complexité temporelle linéaire

Exemple : k-correspondances avec relations de chevauchement

Quatre instructions...

```
1  int a = tab[i] * i ;
2  int b = tab[j] * j ;
3  int a = tab[i] ;
4  byte b = tab[j] * j ;
```

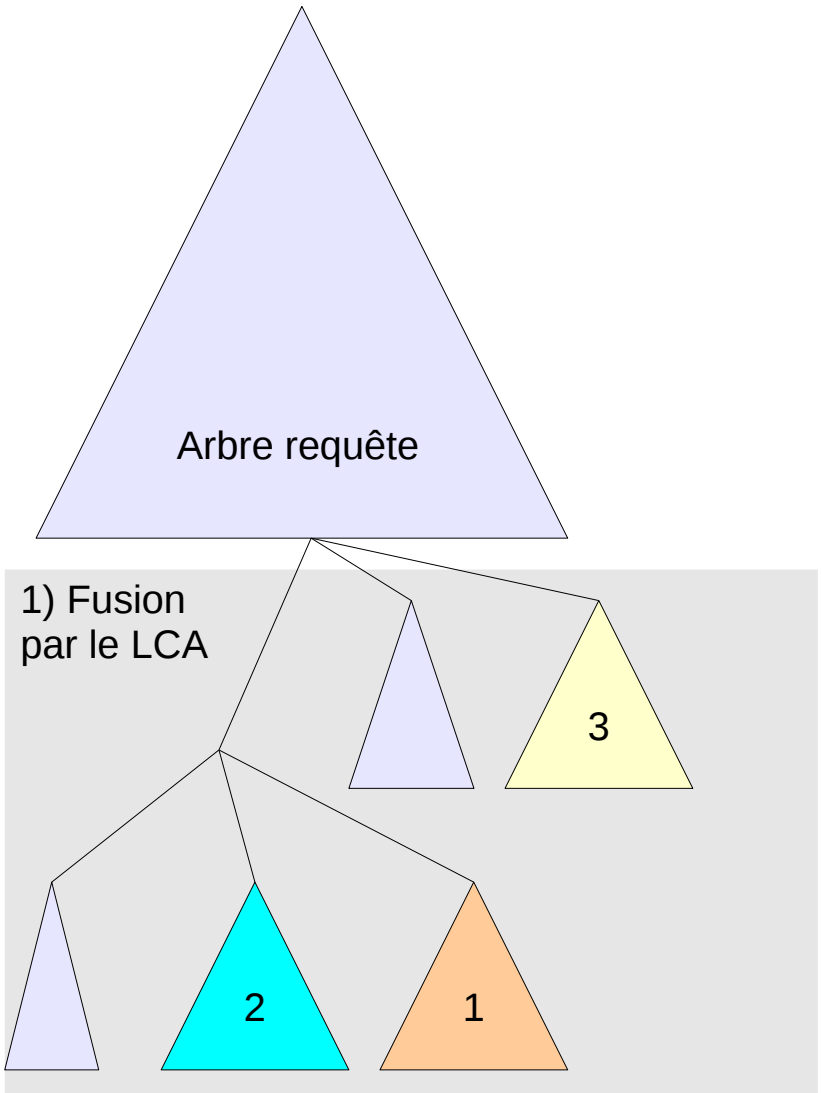
...Et leur graphe de facteurs répétés maximaux [MC2010]

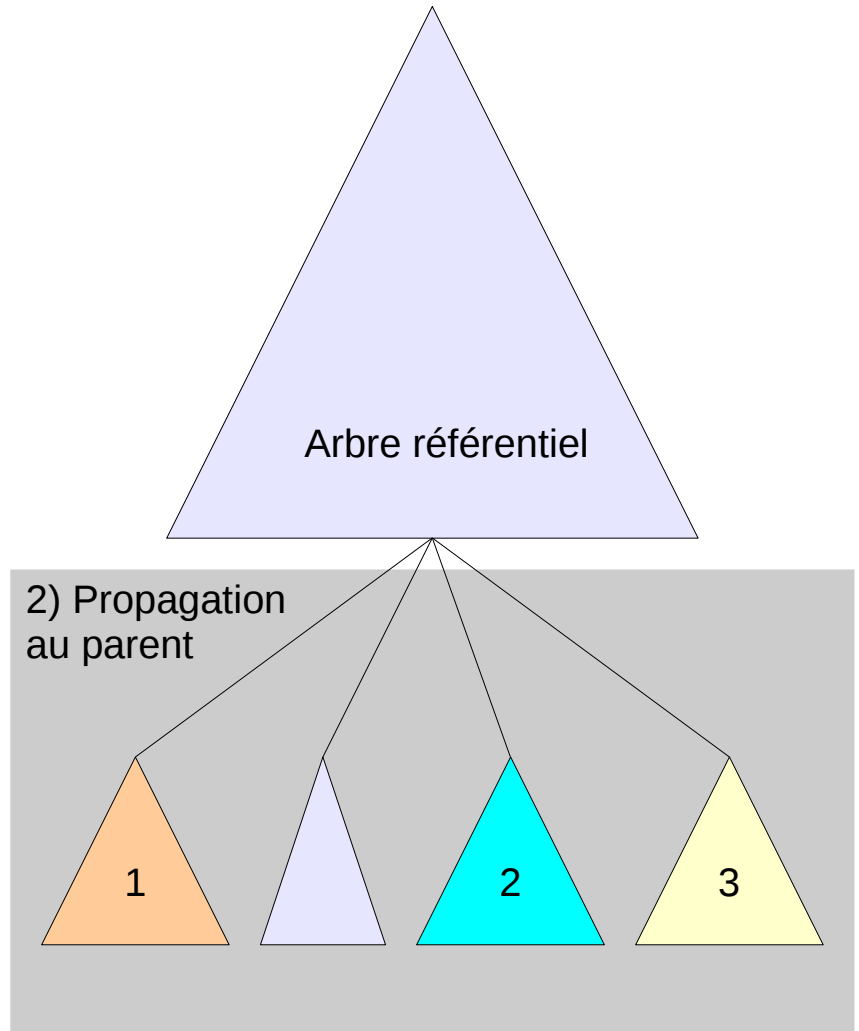


Consolidation en paires d'exemplaires approchés

- Idée : les correspondances exactes spatialement proches sont potentiellement assemblables
- Proposition #1 : hachage *dégradé* dans des bacs et calcul de distance d'édition [Baxter1998]
- Proposition #2 : heuristique gloutonne ascendante **ensembliste** [MC2010]
  - Entrée : correspondances exactes
  - Résultat : forêt d'arbres de correspondances (feuilles = correspondances exactes)

Exemple : assemblage de correspondances exactes





Pour finir : quelques remarques sur les arbres de syntaxe

- Nécessite une analyse syntaxique
- S'adapte aux algorithmes sur chaînes de lexèmes
- Bonne délimitation de correspondances
- Coût temporel favorable pour des duplications de haut niveau
- Abstraction et normalisation aisée
- Consolidation structurelle possible
- Ajout possible d'arcs supplémentaires (liens d'appel, dépendances instructions...)

Quelques références utiles

M. Wise

String similarity via greedy string tiling and running Karp-Rabin matching

Tech. report of the University of Sydney, 1993.

R. Irving

Plagiarism and collusion detection using the Smith-Waterman algorithm

Tech. report of the University of Glasgow, 2004.

I. Baxter et al.

Clone Detection Using Abstract Syntax Trees

Proc. of the Int. Conf. on Software Maintenance, 1998.

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, 2002.

M. Chilowicz

Recherche de similarité dans du code source

Thèse de doctorat de l'Université Paris-Est, 2010.

Une sélection d'outils de recherche de similitudes

Chaînes de lexèmes

- JPlag (service web) : tuilage glouton avec empreintes Karp-Rabin
- Moss (service web) : méta-lexémisation et sélection
- CCFinderX (licence MIT) : indexation par arbre de suffixes

Arbres de syntaxes

- CloneDr (logiciel propriétaire) : recherche de sous-arbres dupliqués avec hachage degradé
- CloneDigger (licence GPL) : arbres avec motifs de haut-niveau similaires (anti-unification)

Plate-forme multi-approches

Plade (licence libre) : en cours de développement

Merci de votre attention