Vincent Alves
Anthony Fournier
Marc-Antoine Testier

Rapport de projet de parcours

13 juin 2014

Remerciements

Nous tenons à remercier toute l'équipe éducative qui nous a encadrée durant ce projet.

Table des matières

1 Introduction

Avec le développement d'internet et des nouvelles technologies, le phénomène de plagiat s'est beaucoup développé, en particulier dans le milieu universitaire. De nombreux enseignants cherchent alors des moyens efficaces pour lutter contre le plagiat. Il existe plusieurs solutions. Dans ce projet, nous étudierons les méthodes utilisant des logiciels de détection de plagiat pour les codes sources. Les outils pour détecter la similarité entre deux codes sources utilisent plusieurs techniques différentes :

- Compter le nombre de variables, nombre d'opérateurs uniques
- Comparaison de chaîne de caractères.
- Comparaison de la structure du code source.

L'objectif final étant la réalisation d'un programme C pouvant détecter le plagiat entre deux extraits de code C fournis.

2 Etat de l'art

Le plagiat dans les codes sources informatiques est assez fréquent, et celui-ci demande d'utiliser des outils différents que ceux utiliser pour la comparaison de texte dans les documents. Un aspect différent du plagiat de code source est qu'il n'existe pas "d'usine à dissertations" que l'on peut trouver dans le plagiat classique. Puisque les devoirs de programmation attendent des étudiants d'écrire des programmes avec des exigences précises, il est difficile de trouver des programmes qui répondent déjà à ces exigences. Puisque intégrer un code externe est souvent plus difficile que de l'écrire à partir de zéro, la plupart des étudiants qui plagient choisissent de le faire avec les code-sources de leurs camarades.
Les algorithmes de détection de plagiat dans les codes sources peuvent se faire sur différents niveaux :

- On peut comparer le code par segment de texte, mais cet algorithme peut être contré en changeant les noms des variables et des fonctions.

- On peut parser le code, afin de retirer les espaces, les commentaires et d'ignorer le nom des variables et fonctions. On peut ainsi comparer les structures de code. Mais cet algorithme peut être contré en rajoutant des boucles ou des conditions inutiles.

- On peut aussi placer la structure dans des arbres et comparer les sous arbres des codes sources.

- On peut comparer les types d'affectations et de déclarations des variables et des fonctions mais la complexité du programme en devient beaucoup plus grande.
- On peut compter le nombre de boucles, conditions et affectations par segment et en déterminer un score selon la similarité de ces nombres par segment entre deux codes. Cependant cela peut entraîner des faux positifs car deux segments de code peuvent avoir les mêmes nombre de boucles, conditions et affectations.
- Un autre algorithme consiste à créer les arbres des suffixes des mots des codes et de les comparer.

Il existe déjà différents logiciels répondant à ce problème gratuits ou non tel que : MOSS, JPlag, The Sherlock Plagiarism Detector et beaucoup d'autres.

3 Analyse du sujet, conception et implémentation

3.1 Concept général

En ce qui concerne notre projet, nous avons décider de réaliser un algorithme de comparaison des structures en parsant les codes, puis en les comparant.

Pour cela, il nous a tout d'abord fallut récupérer le code d'un fichier et de le mettre dans une chaîne de caractère. On modifie ensuite cette chaîne en espaçant le code et en éliminant les lignes comprenants des include, define ou des commentaires, ainsi qu'en éliminant les chaînes de caractères lors des définitions de variable ou de printf.

Nous avons ensuite réduit la chaîne de caractère en ne gardant que la structure du code, sous forme de "boucle", "condition", "affectations" et accolades. "Boucle" ne faisant pas de différence entre une boucle for, while ou do et "condition" ne faisant pas la différence entre un if et un switch .

On rentre ensuite la chaine résultante dans un tableau, chaque ligne du tableau étant une fonction du code, dans laquelle se trouve la structure de la fonction sous forme de tableau contenant chaque mot de la structure de la fonction.

On va alors comparer toutes les fonctions entre elles suivant le fichier auquelles elles appartiennent et en déterminer un score de similitude.

Afin de comparer deux fonctions, nous comparons chaque mot des deux fonctions et incrémentons un score si les mots sont identiques. Si les fonctions ne font pas la même taille, nous faisons de même tout en décalant la fonction la plus courte afin de voir si la structure interne de la fonction la plus grande ressemble à celle de la plus petite fonction.

3.2 Analyse, voies empruntées et abandonées

Le parseur a été implémenté sans problème, le problème principal tenait dans la structure de données à utiliser pour stocker les données parsées.

Avant d'arriver à notre implémentation actuelle, nous sommes passés par plusieurs étapes, que nous avons par la suite abandonées :

Au début, en nous basant sur l'énoncé qui nous a été donné, nous avons essayé de séparer les fonctions parsées sous la forme d'arbres. Ceci nous permettrait de tester pour les échanges de fonctions plus facilement(Une méthode de plagiat consistant à inverser la place des fonctions dans le code), ainsi que d'avoir éventuellement une sortie graphique facile à comprendre.
Cependant, la structure des arbres s'est montrée trop difficile à implémenter en C, et elle nous aurait pris longtemps à appliquer à notre projet. Au vu de la courte durée d'implémentation(4 jours), nous avons décidé d'abandonner cette solution.

Par la suite, nous avons pensé à implémenter la structure des listes, qui était plus simple que les arbres, et qui nous aurait permis d'arriver à un résulat similaire.
L'implémentation de la structure a été faite, mais nous n'avons pas réussi à la faire concorder avec le parseur des fonctions. Un reste du code essayé se trouve dans le fichier *List.c* Ces deux types de structure s'étant révelés non concluants, nous avons essayé une implémentation basée sur le type du tableau, plus simple et déjà implémenté de base.
Le fonctionnement de notre programme avec ce type a été expliqué dans la section précédente.

3.3 Implémentation

Une fois les fonctions des codes espacées et parsées dans des tableaux, nous pouvons commencer à vérifier les similitudes entre ces tableaux pour calculer des pourcentages de ressemblance.
Pour ce faire, nous comparons mot à mot (chaque case d'une ligne du tableau correspond à un mot) toutes les fonctions des deux codes, deux par deux.

Si deux cases sont les mêmes, le score de la comparaison est incrémenté de 1. A la fin de la comparaison, le score total est divisé par la longueur de la fonction pour obtenir un pourcentage.
Si les deux fonctions n'ont pas la même longueur (un cas très fréquent), le score total est divisé par le minimum. Nous effectuons alors également plusieurs comparaisons, en décalant la fonction la plus petite dans la plus grande, afin de vérifier si une fonction n'est pas inclue dans la deuxième.

Lors d'une comparaison ou les deux fonctions n'ont pas la même taille, il est possible qu'une des fonctions soit bien plus petite que la seconde. Cela peut entraîner des faux résultats : En effet, si une fonction contenant juste une boucle est comparée à une fonction qui contient entre autres une boucle, le pourcentage renvoyé sera proche de 100%(vu que nous divisons par la taille minimale).

Pour éviter cela, nous pondérons le résultat afin de réduire le pourcentage si la différence entre les fonctions est trop grande. En effet, si cette différence est plus grande que les $\frac{2}{3}$ de la taille du plus grand fichier, nous multiplions le score obtenu par $\frac{tailleMin}{tailleMax}$.

Une fois nos comparaisons faites, nous définissons un seuil arbitraire à partir duquel la fonction est considérée plagiée.
Si le pourcentage est supérieur à 80%, nous marquons la fonction comme plagiée.

4 Conclusion

4.1 La compétition - Résultats concrets

Au début de la compétition, le seuil que nous avions précisé était de 70%. Or, en testant avec le jeu de test fourni pour la compétition, nous avons alors trouvé beaucoup de faux positifs. Pour résoudre cela, nous avons monté le seuil à 80%, et avons obtenu un résultat plus concluant, mais avec tout de même plusieurs faux positifs.

En effet, le seul test que nous avons eu le temps d'implémenter, celui des comparaisons structurelles, n'est pas très fiable : En effet, même si anonymiser les instructions permet de contrer les méthodes de changement de variable et de nom, le code comparé que l'on obtient est trop vague et amène de ce fait trop de faux positifs.
En effet, imaginons un code 1 :

```
int i=0;
for (i;i<5,i++)
{
unTruc();
}
```

et un code 2 :

```
int j=0;
do{
unTrucQuiN'aRienAVoir();
j++;
} while (j<200);
```

Ces deux codes seront interprétés comme aff boucle { } , alors qu'ils ne font absolument pas la même chose.

4.2 Conclusion finale

Nous sommes finalement plutôt satisfaits de notre projet, celui-ci correspondant à nos attentes. Nous aurions aimé améliorer quelque peu notre programme et nous regrettons donc de ne pas avoir plus de temps pour le réaliser. Avec plus de temps, nous aurions pu créer les arbres des structures des codes, ainsi que d'autres tests de détections plus complexes et fiables.

A la base, nous avions prévu plusieurs tests en plus de celui-ci, qui contribueraient à affiner le score final. Cependant, ayant perdu trop de temps sur les structures de données, nous n'avons pas eu le temps d'en implémenter d'autres.

5 Bibliographie

Pour réaliser ce projet, nous avons utilisé la documentation suivante :
`http://en.wikipedia.org/wiki/Plagiarism_detection`
Page Wikipédia sur la détection de plagiat.

`http://en.wikipedia.org/wiki/Parse_tree`
Page Wikipedia sur les arbres syntaxiques.

`http://en.wikipedia.org/wiki/Suffix_tree`
Page Wikipédia sur les arbres des suffixes.

`ftp://ftp-developpez.com/sjrd/tutoriels/compilation/analyseurs-syntaxiques.pdf`
Analyseurs syntaxiques, Leur fonctionnement par l'exemple
(Sébastien Jean Robert Doeraene)

`http://llvm.org/docs/tutorial/LangImpl2.html`
Implémentation d'un AST.

`http://en.wikipedia.org/wiki/Abstract_syntax_tree`
Page Wikipédia sur les AST.

`http://sydney.edu.au/engineering/it/~scilect/sherlock/`
Sherlock : Détecteur de plagiat.

6 Manuel d'utilisation

Pour lancer le programme, il suffit d'ouvrir un terminal et d'accéder au dossier où se situe le main. Il faut ensuite soit utiliser la commande "./compare.sh" qui compare tous les fichiers de tests contenus dans le dossier CompeteDeLaMuerte. Ou alors utiliser la commande "./main fichier1 fichier2", qui compare uniquement 2 fichiers ensemble.