VAMOS A LA PLAGIAT*

Projet Programmation C, L2 MIDO 2022-2023

Résumé

Le but de ce projet est de détecter le plagiat entre deux fichiers contenant du code source C. Pour cela, une chaîne de traitement est appliquée, jusqu'à la production finale d'un rapport de similarité entre les deux fichiers.

Note : ce projet est à but pédagogique [1] : un vrai détecteur de plagiat sera appliqué sur vos projets.

1 Dates importantes

- ▷ Deadline composition des groupes (Excel dans le Teams) : 16 décembre 2022 à 23h59.
- ▷ Rendu du projet : vendredi 3 février 2023 à 23h.
- ▷ Soutenances : à définir, probablement semaine du 6 février 2023.

2 Versions

Ce sujet est susceptible d'être modifié.

- ▷ 8 décembre 2022 : première version du projet.

3 Description

Pour détecter le plagiat entre deux fichiers de code source C, il faut, entre autres, être capable de repérer :

- ▷ le renommage de variables ou fonctions ;
- ▷ le déplacement des variables ou fonctions à différents endroits dans le code ;
- ▷ l'ajout, la modification ou la suppression d'information non essentielle.

Pour cela, on va effectuer une chaine de traitement sur les deux fichiers, chaîne dont chaque élément sera détaillé plus loin :

1. pré-traitement
2. découpage
3. mesure de similarité
4. couplage
5. filtrage
6. affichage

*https://www.youtube.com/watch?v=FH1BiCKCEGA

1. Ce projet est inspiré de l'article http://hal.archives-ouvertes.fr/hal-01066127

4 La chaîne de traitement

Note : votre programme ne doit jamais modifier le contenu des fichiers donnés en entrée.

Input

Le projet sera lancé en ligne de commande en passant comme arguments les noms des deux fichiers que l'on souhaite comparer.

Exemple : `./a.out fichier1.c fichier2.c`

Bonus (moyen) : On souhaite maintenant être capable de comparer deux-à-deux plus de deux fichiers sources. Votre programme sera alors lancé en listant en paramètre de la commande les noms de tous les fichiers comme par exemple :
`./a.out fichier1.c fichier2.c fichier3.c main.c`
Si n fichiers sont indiqués, on effectuera les $\frac{n(n-1)}{2}$ comparaisons deux-à-deux et l'affichage final sera la matrice $n \times n$ de toutes les distances obtenues.

4.1 Pré-traitement

La première étape du projet est de transformer chaque ligne non vide des fichiers de façon à éliminer les informations non pertinentes pour l'analyse et ne garder que la « structure » du programme. Le projet doit donc effectuer sur chaque ligne des fichiers les opérations suivants :

- ▷ suppression des commentaires (on se limitera aux commentaires débutant par `//`) ;
- ▷ suppression du contenu des chaînes de caractères ;
- ▷ chaque mot (suite de caractères alphanumériques) est transformé en un unique caractère « `w` ». Cela permet de s'affranchir du renommage des variables ou des fonctions puisque chaque identificateur (et chaque mot-clé) se retrouve transformé en une unique lettre ;
- ▷ suppression de tous les espaces et tabulations.

Il ne doit donc plus rester dans chaque ligne que les caractères de ponctuations de code (parenthèses, crochets, points-virgules, opérateurs, etc.) et les 'w' qui marquent l'emplacement des mots.

Ainsi par exemple la ligne :

```
if (a>2) { printf("%d\n", a*T[i]); return 0; } // fin
```

va se voir transformée en :

```
w(w>w){w("",w*w[w]);ww;}
```

Bonus (facile) : Afin de mieux capter la structure du code, chaque mot est remplacé par « `w` » SAUF s'il s'agit d'un mot-clé du C [2] qui est alors remplacé par « `m` ». L'exemple précédent donnera donc la ligne :

```
m(w>w){w("",w*w[w]);mw;}
```

L'affichage final donnera les deux distances obtenues, avec ou sans mise en valeur des mots-clés.

2. On prendra la liste des mots-clés suivants : `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `if`, `int`, `long`, `return`, `short`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `while`

4.2 Découpage en blocs/segments

La deuxième étape consiste à découper chaque fichier en blocs élémentaires, appelés segments, qu'on va pouvoir comparer entre eux. L'idéal est de définir la taille d'un segment pour qu'elle corresponde à peu près à l'unité sémantique du langage et que l'on puisse donc comparer les instructions élémentaires de chaque fichier entre elles.

Pour simplifier, et parce que cela correspond à peu près à une instruction C, chaque ligne formera un segment.

À la fin de cette étape, on en est au point où chaque fichier à comparer est résumé par un certain nombre de lignes non vides, chaque ligne ayant été pré-traitée comme indiqué dans la section précédente.

4.3 Similarité entre segments

On se retrouve maintenant avec n segments dans le premier fichier et m dans le second. On cherche à comparer ces segments deux à deux pour détecter les similarités.

Pour chaque couple de segments (S_i^1, S_j^2), où le premier segment est dans le premier fichier et le second segment dans le second fichier, on calcule la distance de Dice [3] (voir ci-dessous), comprise entre 0 (segments identiques) et 1 (segments totalement différents).

On obtient ainsi une matrice D de taille $n \times m$ de coefficients compris entre 0 et 1 donnant la similarité entre deux segments.

On génèrera un fichier image `dice.pgm`, en niveau de gris comme vu en TP, représentant la matrice D, le pixel étant d'autant plus clair que les segments sont proches.

Distance de Dice

Étant donnés deux chaînes `x` et `y`, on peut calculer leur distance de Dice comme suit :

$$d = 1 - \frac{2n_c}{n_x + n_y}$$

où n_c est le nombre de digrammes (formés de deux caractères consécutifs) communs aux deux chaînes, n_x est le nombre de digrammes dans `x` et n_y le nombre de digrammes dans `y`.

Par exemple, pour calculer la similarité entre `richesse` et `chasse`, on calcule les digrammes de chaque mot :

```
ri ic ch he es ss se
ch ha as ss se
```

Il y a sept digrammes dans le premier ensemble et cinq dans le second ; leur intersection est formée des trois digrammes `ch`, `ss` et `se`.

On obtient donc une distance $d = 1 - 2 \times 3/(7 + 5) = 0,5$.

Bonus (facile) : Comparer les résultats avec d'autres longueurs possibles de n-grammes et discuter n le plus pertinent (le sujet étant donné pour $n = 2$).

Bonus (difficile) : Il existe une autre distance mesurant plus naturellement notre impression de distance entre deux chaines : il s'agit de la distance de Levenshtein ou distance d'édition ; elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer

3. http://fr.wikipedia.org/wiki/Indice_de_S%C3%B8rensen-Dice

pour passer d'une chaîne à l'autre [4]. La définition est donnée de manière récursive mais malheureusement, l'algorithme récursif évident est beaucoup trop lent en raison de sa complexité exponentielle. Il faut donc implémenter l'algorithme de programmation dynamique donné sur la page wikipedia. Il faudra ensuite normaliser la distance obtenue pour la ramener dans l'intervalle $[0, 1]$.

4.4 Couplage des segments

Nous avons maintenant une matrice des distances entre chaque segment de chaque programme. Afin de détecter le plagiat, il faut repérer, entre les deux programmes, quels sont les segments qui se ressemblent (ils ne sont pas forcément au même endroit car il y a pu avoir des permutations de lignes). Pour ce faire nous allons associer deux à deux les segments et obtenir une liste de couples (s^1, s^2). Bien sûr, nous souhaitons associer des segments semblables, qui sont à une faible distance. Nous cherchons donc une liste de couples, la plus grande possible, telle que la somme de toutes les distances de chaque couple soit la plus petite possible. Il s'agit du problème du couplage maximum de poids minimum dans un graphe biparti, aussi appelé problème d'affection [5].

Dans notre cas, cela revient à prendre le plus petit des deux programmes (celui qui a le moins de segments) et à chercher à associer chaque segment avec un segment qui lui ressemble dans le second programme.

Prenons un exemple : supposons que nous avons deux programmes, de 3 et 4 segments, et que la matrice des distances soit :

$$\begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \end{array} \begin{array}{c} \begin{array}{cccc} b_1 & b_2 & b_3 & b_4 \end{array} \\ \begin{pmatrix} 0,1 & 0,2 & 0,3 & 0,3 \\ 0,5 & 0,8 & 0,7 & 0,6 \\ 1 & 0,2 & 0,9 & 0,4 \end{pmatrix} \end{array}$$

Le but est de coupler les trois segments a_1, a_2 et a_3 avec trois segments différents du second programme de tel sorte que la somme de leur distance soit minimale. On peut avoir le couplage $(a_1, b_3); (a_2, b_1); (a_3, b_2)$ qui a pour somme totale des distances $0,3 + 0,5 + 0,2 = 1$, mais le couplage $(a_1, b_1); (a_2, b_4); (a_3, b_2)$ a pour somme $0,1 + 0,6 + 0,2 = 0,9$ est plus intéressant. On s'aperçoit assez rapidement que ce dernier couplage est celui donnant la somme la plus petite, c'est donc celui recherché.

Algorithme glouton de couplage minimal

Trouver le couplage maximum de poids minimum se fait avec un algorithme assez compliqué (voir bonus) mais nous pouvons utiliser ici un algorithme glouton qui donne une assez bonne approximation du résultat.

Le principe de l'algorithme est de systématiquement chercher les deux segments qui se ressemblent le plus dans la matrice D (donc le plus petit coefficient).

1. chercher le plus petit coefficient de la matrice ; il s'agit de $d_{i,j}$, à l'intersection de la ligne $0 \leq i < n$ et de la colonne $0 \leq j < m$;

4. https://fr.wikipedia.org/wiki/Distance_de_Levenshtein
5. https://fr.wikipedia.org/wiki/Probl%C3%A8me_d%27affectation

2. On associe alors les segments i du premier programme et j du second, et on les « retire » de la matrice en laissant $d_{i,j}$ mais en mettant tous les autres coefficients de la ligne i et la colonne j à la valeur 1 et en mémorisant que cette ligne et cette colonne ont déjà été utilisées ;
3. on recommence en cherchant le plus petit coefficient dans le reste de la matrice tant que le plus petit programme a encore un segment à associer.

On se retrouve au final avec une matrice C remplie de 1 sauf lorsque les segments i et j ont été couplés et pour lesquels le coefficient $d_{i,j}$ donne la distance entre ces deux segments.

On génèrera un fichier image `couplage.pgm` représentant la matrice C, le pixel étant d'autant plus clair que le coefficient est petit (et donc les segments proches).

Bonus (très difficile) : l'algorithme glouton ne donne en général pas la valeur optimale du couplage. Il existe cependant un algorithme polynomial permettant de trouver l'optimum : l'algorithme hongrois[6]. On essayera donc de le programmer pour améliorer le calcul du couplage entre segments.

4.5 Post-filtrage

Nous faisons l'hypothèse que deux documents sont suspects s'il a des blocs de segments *contigus* suspects, c'est-à-dire des segments consécutifs couplés à l'étape précédente. Le but de cette étape est donc de supprimer les couplages de segments isolés qui sont probablement plus des coïncidences que du plagiat.

Pour ce faire, nous allons recalculer chaque coefficient $c_{i,j}$ de la matrice précédente comme une moyenne avec les quatre coefficients adjacents situés sur la même diagonale[7] (et correspondant donc à cinq segments consécutifs dans les deux fichiers) et ne garder que les coefficients suffisamment petits via une fonction de seuil. Si des segments consécutifs de chaque fichiers sont plagiés, cela signifiera que plusieurs coefficients consécutifs (sur une diagonale) seront proches de 0 et leur moyenne également. En revanche, un couplage isolé ne produira qu'un seul petit coefficient sur les 5, donc une moyenne élevée qui sera ensuite éliminée par le seuil.

Calcul post-filtrage

À partir de la matrice C obtenue à l'étape précédente, on calcule une nouvelle matrice F comme suit :

$$f_{i,j} = \frac{1}{5} \sum_{k=-2}^{2} c_{i+k,j+k} \text{ (convolution), puis } f_{i,j} = 1 \text{ si } f_{i,j} \geq 0.7 \text{ (seuil)}$$

On génèrera un fichier image `filtrage.pgm` représentant la matrice finale F, le pixel étant d'autant plus clair que le coefficient est petit.

6. https://fr.wikipedia.org/wiki/Algorithme_hongrois
7. Mathématiquement, il s'agit du produit de convolution de C avec la matrice identité 5×5.

4.6 Output des résultats

Nous pouvons maintenant calculer la « distance » entre les deux fichiers en sommant les distances entre les segments couplés.

$$\Delta = 1 - \frac{1}{\min(n, m)} \sum_{\substack{0 \leq i < n \\ 0 \leq j < m}} (1 - f_{i,j})$$

La sortie du programme sera donc la valeur finale de distance obtenue avec deux chiffres après la virgule.

Bonus (moyen) : On affichera également les couples de lignes originales des deux fichiers ayant une distance de 0 (donc les lignes normalement « identiques »), avec leurs numéros de ligne dans le fichier original. En pourra également envisager d'afficher des lignes avec distance plus grande que 0 en mettant en évidence (avec de la couleur ?) les zones plagiées.

Bonus (moyen) : Si on compare plus de deux fichiers sources (bonus « input »), l'affichage final sera la matrice $n \times n$ de toutes les distances obtenues, triée de manière pertinente.

5 Ce qui est demandé

Vous devez mettre en œuvre l'algorithme décrit pour calculer et afficher la distance entre deux fichiers sources C qui seront indiqués en ligne de commande, ainsi que générer les matrices demandées sous forme d'image niveaux de gris `.pgm`. On ne fera aucune hypothèse sur la taille maximale des fichiers ou des lignes. **Chaque fichier sera stocké dans votre programme sous la forme d'une liste chaînée de lignes.**

Une fois le projet de base parfaitement fonctionnel, on pourra chercher à implémenter les bonus des différentes sections.

6 Conditions de rendu

Le projet est à effectuer en **binôme**. En cas de nombre impair d'étudiants, **un seul** groupe sera autorisé à effectuer le projet en **trinôme** (notation plus sévère). Pour former les groupes, utiliser le fichier Excel dans le canal Teams correspondant. **Les binômes doivent être différents de ceux utilisés pour le projet d'Archi !**

Toute question sur le projet devra être posée dans l'équipe Teams (et non par mail ou en individuel) afin de limiter les doublons de questions.

Le projet est à rendre avant la date et l'heure indiquées plus haut sur l'espace Moodle dédié. **Chaque heure de retard sera pénalisée d'un point sur la note finale** (une heure entamée étant due). Le format de rendu est une archive au format ZIP contenant :

- ▷ Le code-source de votre projet (éventuellement organisé en sous-dossiers).
- ▷ Un fichier `README` à la racine, indiquant comment compiler et exécuter votre projet.
- ▷ Un document `dev.pdf`, devant justifier les choix effectués, les avantages et inconvénients de vos choix, expliquer les algorithmes et leur complexité, indiquer quelles ont été les difficultés rencontrées au cours du projet ainsi que la répartition du travail entre les membres du binôme. Un programmeur averti devra être capable de faire évoluer

facilement votre code grâce à sa lecture. En aucun cas on ne doit y trouver un copier/coller de votre code source. Ce rapport doit faire le point sur les fonctionnalités apportées, celles qui n'ont pas été faites (et expliquer pourquoi). Il ne doit pas paraphraser le code, mais doit rendre explicite ce que ne montre pas le code. Il doit montrer que le code produit a fait l'objet d'un travail réfléchi et minutieux (comment un bug a été résolu, comment la redondance dans le code a été évitée, comment telle difficulté technique a été contournée, quels ont étés les choix, les pistes examinées ou abandonnées...). Ce rapport est le témoin de vos qualités scientifiques mais aussi littéraires (style, grammaire, orthographe, présentation).

- ▷ Un court document `exp.pdf` étant un rapport d'expériences faites avec votre programme pour différents types de plagiat ou de non-plagiat et les conséquences sur votre programme.
- ▷ Optionnellement un makefile.

L'archive aura pour nom `Nom1Nom2.zip`, où Nom1 et Nom2 sont les noms des membres du binôme par ordre alphabétique. L'extraction de l'archive devra créer un dossier Nom1Nom2 contenant les éléments précisés ci-dessus.

Il va sans dire que les différents points suivants doivent être pris en compte :

- ▷ Projet compilant sans erreur ni warning avec l'option `-Wall` de gcc.
- ▷ Uniformité de la langue utilisée dans le code (anglais conseillé) et des conventions de nommage et de code.
- ▷ Les sources doivent être commentées, dans une unique langue, de manière pertinente (pas de commentaire "fait un test" avant un `if`).
- ▷ Le code doit être propre et correctement indenté.
- ▷ Bonne gestion de la libération de la mémoire (utilisez valgrind pour vérifier que le nombre de `free` est égal au nombre de `malloc` : il faut d'abord compiler avec l'option `-g`, puis lancer valgrind avec en argument le nom du programme. Par exemple :

  ```
  gcc -g main.c
  valgrind ./a.out
  ```

  Valgrind peut également vous indiquer la ligne où un segmentation fault a eu lieu, pratique ! – vous pouvez aussi compiler avec l'option `-fsanitize=address` pour savoir cela.
- ▷ Le projet doit évidemment être propre à chaque binôme. **Un détecteur automatique de plagiat sera utilisé**. Si du texte ou une petite portion de code a été emprunté (sur internet, chez un autre binôme,...), il faudra l'indiquer dans le rapport, ce qui n'empêchera pas l'application éventuelle d'une pénalité. Tout manque de sincérité sera lourdement sanctionné (conseil de discipline) – c'est déjà arrivé.

La documentation (rapports, commentaires...) compte pour un quart de la note finale.

Soutenance

Une soutenance d'une dizaine de minutes aura lieu pour chaque binôme, à la date et à l'heure indiquées plus haut. Elle doit être préparée et menée par le binôme (*i.e.* fonctionnant parfaitement du premier coup, avoir préparé des jeux de tests intéressants, etc.).

Pendant la soutenance, ne perdez pas de temps à nous expliquer le sujet : nous le connaissons puisque nous l'avons écrit. Essayez de montrer ce qui fonctionne et de nous convaincre que vous avez fait du bon travail.