Rapport de Projet : Programmation C

1 Présentation

Le but de ce projet est de créer un programme capable de détecter le plagiat entre deux fichiers contenant du code source C. Nous allons présenter dans ce document les différentes phases de développement de ce projet. Ce projet est découpé en plusieurs parties formant la chaine de traitement sur les deux fichiers : pré-traitement, découpage en segments, similarité entre segments, couplage des segments, post-filtrage et l'output des résultats.

2 Structures nécessaires : `cellule` et `chained_list`

3 Pré-traitement

Dans cette partie, on procède à l'élimination des informations non pertinentes dans les codes sources respectifs. Entre autres, on supprime les commentaires ainsi que les espaces, les tabulations, le contenu des chaines de caractères et on renomme tous les mots du fichier, en particulier les identificateurs en la lettre "`w`".

Pour cela, nous avons choisi de diviser cette tâche en plusieurs sous-taches réparties dans différentes fonctions afin de répartir les différentes tâches à réaliser de manière claire et lisible dans le code. La réalisation de ces sous-taches s'effetuera alors dans la fonction principale de cette partie `filtering()`.

3.1 Fonction de filtrage : `filtering()`

Cette fonction `filtering()` est celle qui va effectuer la partie de pré-traitement. Celle-ci sera effectuée pour les deux fichiers, séparément de sorte à ce que la même opération effectuée sur deux fichiers ne soit pas répétée dans le code, nous appliquons simplement la fonction de filtrage aux deux fichiers (plutôt que d'effectuer une fonction de filtrage qui filtre les deux fichiers en même temps rendant les choses beaucoup moins lisible dans le code).

Nous allons donc appliquer cette fonction dans la fonction `main()` aux deux fichiers à traiter séparément.
La fonction `filtering()` prend en argument le fichier à traiter (`file_name_e`), le nom que l'on va donner au fichier (`file_name_o`) ainsi qu'une liste chainée (`list`) qui contiendra, à l'issue de l'éxécution de la fonction, une liste chainée dont les éléments sont les lignes du fichier après traitement.

Dans `main()`, cette fonction est éxécutée après la vérification des fichiers : on vérifie si les deux fichiers s'ouvrent bien à l'aide d'une fonction auxiliaire `opening_test()` et on vérifie également si les fichiers en entrée sont vides à l'aide d'une autre fonction `count_lines()`. Si un fichier n'existe pas, ou bien si un fichier est vide, on a choisi d'afficher un message d'erreur indiquant l'erreur pour en informer l'utilisateur et le programme s'arrête dans ce cas.

La fonction `filtering()` lit le fichier de manière séquentielle, nous avons choisi de lire les lignes du fichier une par une à l'aide d'une boucle `for` sur le nombre de lignes du fichier (obtenu par la fonction `count_lines()` qui compte le nombre de lignes d'un fichier), et d'appliquer les fonctions qui effectuent les différentes sous-taches du pré-traitement ligne par ligne. Nous avons choisi d'initialiser une variable `current_line` qui stocke la ligne courante (obtenue par `fgets()`). Cependant, on ne connaît pas à l'avance le nombre de caractères maximum que pourraît contenir une ligne du fichier, en l'occurrence `current_line`. C'est pourquoi, nous avons fixé une taille maximale de 200 caractère à `current_line`, puis nous faisons un test sur la chaine de caractère récupérée. Si toute la ligne a été récupérée, alors on continue le programme. Sinon, on double la taille allouée pour `current_line` et on continue tant que la ligne n'est pas entièrement stocké. Cela nous permet donc d'avoir une allocation dynamique. On appliquera alors successivement les différentes fonctions qui effectuent les différentes taches de filtrage sur cette ligne courante, dans le but de ne pas modifier le fichier en entrée.

3.2 Fonctions de traitement des lignes

Ces fonctions ont des entrées et sorties simples, et effectuent une seule tache ciblée et indiviudalisée ce qui rend plus facile la compréhension du code global :

- prennent tous en argument une chaine de caractères (dans notre cas, la ligne courante non traitée)
- effectuent le traitement de filtrage ciblé sur la chaine : `str_without_comments()` supprime les commentaires présents sur la ligne, `str_without_spaces_tabs()` effectue la suppression des espaces et tabulations sur la ligne, `str_without_strings()` supprime les chaines de caractères présents sur la ligne et `words_to_w()` change toute suite de caractères alphanumériques présents sur la ligne en la lettre "`w`".
- renvoient tous une chaine de caractère (dans notre cas, la ligne courante traitée par la tache en question)

Cependant, quelques difficultés ont été recontrées lors de l'écriture de ces fonctions car la manipulation de chaines de caractères est assez difficile en `C`. Lors de la manipulation de chaines de caractères, il faut être très minutieux et attentif. Il fallait notamment faire attention à ne pas oublier le caractère nul `\0` à la fin des chaines de caractères/lignes, sans quoi rien ne fonctionnait à cause des `segmentation fault.`

4 Découpage en segments

L'étape de pré-traitement de chaque ligne de fichier dans la fonction `filtrage()` est suivie d'un découpage. Après avoir traité la ligne courante dans la fonction, nous avons décidé d'ajouter chaque ligne traitée du fichier dans une liste chainée (en utilisant une structure de liste chainée `chained_list` définie en début de programme). Cette liste chainée est donnée en argument et initialisée dans la fonction `main()`. Etant donné que dans le langage `C`, chaque ligne correspond à peu près à une instruction, donc dans toute la suite, chaque ligne représentera un segment.

Ainsi, à la fin du processus de pré-traitement de chaque fichier, on disposera de deux listes chainées respectivement associées aux deux fichiers, contenant chacune exactement les segments traités, avec lesquels on va travailler pour déterminer la similartié entre ces deux fichiers. Ces deux listes chainées sont alors une représentation différente des fichiers avec lesquels on travaille. On a attribué un champ `length` dans notre structure de liste chainée afin de toujours avoir accès au nombre de segments pour chaque fichier.

On se retrouve alors avec deux listes chainées `list1` et `list2` de tailles respectives n et m, avec $n, m \in \mathbb{N}^*$.

5 Similarité entre segments

On compare deux à deux les mn segments (S_i, S_j) formés par les deux fichiers (S_i est un segment de la liste chainée `list1` associée au premier fichier et S_j est un segment de la liste chainée `list2` associée au deuxième fichier) en calculant leur distance de dice. Ce calcul est effectué par la fonction `dice_distance()`.

5.1 Distance de dice

Le calcul de la distance de Dice requiert préalablement le calcul du nombre de digrammes contenus dans chacun des segments et le calcul du nombre de digrammes communs aux deux segments. Nous avons décidé de créer deux fonctions réalisant séparément ces deux tâches :

- `number_of_digrammes()` qui prend en argument un segment et renvoie le nombre de digrammes relatifs à ce segment. Pour cette fonction, nous avons du traiter les différents cas, lorsque la longueur de la chaine est strictement inférieure à 2, alors comme on ne peut pas former de digramme, alors on renvoie la valeur zéro. Sinon, on traite le cas général, pour un chaine de longueur N, on peut trouver $N-1$ digrammes, excepté si `\n` est l'avant dernier caractère (avant `\0`), on ne doit donc pas compter ce digramme.

- `compare_digrammes()` qui prend en argument deux segments `str1` et `str2` et renvoie le nombre de digrammes communs à ces deux segments. Cette fonction range les digrammes des deux segments dans deux tableaux respectifs `tab_digrammes1` et `tab_digrammes2`. On effectue donc une boucle pour chaque segment. Sur ces boucles, plutôt que de donner une condition d'arrêt sur le nombre de digrammes ou le nombre de caractères du segment, on a préféré donner une terminaison sur le caractère courant : on parcourt les caractères du segment et on s'arrête lorsque le caractère courant est soit `\0`, soit `\n`. Cela a l'avantage de savoir si on s'arrête bien à la fin de la chaine. Dans une version antérieure de la fonction, on avait effectué une boucle `for` sur la longueur du segment, mais il s'est avéré pour nous plus difficile à manier, on a notamment eu quelques difficultés, bugs au niveau de la gestion des caractères `\0` et `\n` et cela devenait pas lisible. A chaque itération de boucle, on sauvegarde le digramme courant dans une variable `current_digramme` (initialisée de longueur 3) et on l'ajoute à la liste des digrammes. Il fallait faire attention à ne pas oublier le caractère nul `\0` en troisième position du digramme courant, initialement oublié. Après avoir mis les digrammes des segments dans leur liste respective, on a compté le nombre de digrammes communs à ces deux segments à l'aide des listes de digrammes que l'on avait établies. Le nombre de digrammes communs est sauvegardé dans la variable `count` et sera renvoyée à la fin de la fonction. Etant donné qu'on avait alloué dynamiquement les listes, il a fallu à la fin de la fonction, libérer l'espace mémoire alloué à ces deux listes en utilisant `free()`.

Le calcul de la distance de Dice pour tous les segments $\{(S_i, S_j)_{i,j\in\mathbb{N},\ i\leq n,\ j\leq m}\}$ s'effectue alors dans la fonction `dice_distance()` dont les arguments sont les listes chainées (on a choisi de les passer en argument car on a besoin de leur taille pour calculer les distances pour l'ensemble des segments possibles). On donne également en argument deux listes de flottants `C` et `D` qui représenteront notre matrice de Dice à la fin de l'éxécution de la fonction. Elles ont été initialisées par allocation dynamique dans `main()` à la taille mn.

On a choisi dans un souci de simplification de représenter nos matrices de valeurs par des tableaux de flottants, beaucoup plus simples à manipuler que les matrices de flottants : cela évite largement le recours à l'utilisation de pointeurs. Dans toute la suite, on utilise alors l'équivalence des représentations `M[i][j]=T[i*m+j]` où `M` est une matrice de taille $n \times m$ et `T` est une liste de flottants de taille nm, m représentant le nombre de colonnes et n le nombre de lignes. A partir de `T[k]` pour un certain entier naturel $k \leq mn$, on peut facilement trouver les indices correspondant aux colonnes et aux lignes en utilisant respectivement l'opérateur modulo `%` et l'operateur quotient.

Dans la fonction principale `dice_distance()`, on parcourt les segments des deux listes chainées, on calcule leur distance de Dice. On a également fait le choix de traiter les différents cas spécifiques pour aller plus vite, si les segments sont identiques, on associe leur distance à zéro, si l'un des segments n'a pas de digramme, on associe leur distance à un, sinon on procède au calcul de la distance de dice, en utilisant `compare_distance()` dans la formule de la distance de Dice. On remplit à chaque fois notre liste avec les distances aux indices correspondant aux lignes et aux colonnes : si on a calculé la distance $d(S_i, S_j)$ entre le i-ème segment de la première liste chainée et le j-ième segment de notre seconde liste chainée alors la case d'indice (i, j) contient $d(S_i, S_j)$, il s'agit de la case d'indice $im + j$ pour notre tableau.

A chaque itération, on ajoute la valeur de la distance dans un fichier `dice.pgm`, initialisé au début de la fonction. Quelques problèmes sont survenus lors de la réalisation des fichiers `.pgm` : on avait oublié d'écrire `P2` et la taille de la matrice au début du fichier et de plus, le fichier ne s'ouvre pas forcément si les valeurs ne sont pas entières (`float`), cela dépend du logiciel utilisé (pour `Gimp` cela fonctionne, mais cela ne fonctionne pas pour `Libre Office Draw`). Il fallait également ne pas oublier d'effectuer l'opération 255-*valeur* pour avoir inverser la couleur du pixel.

A la fin de l'éxécution, les listes `C` et `D` contiennent les valeurs des distances et un fichier `dice.pgm` est généré et représente notre matrice en image de pixels en niveau de gris.

5.2 Distance de Levenshtein (bonus)

Nous avons également implémenté l'algorithme pour calculer les distances entre segments à l'aide d'une autre distance, la distance de Levenshtein ; dans la fonction `levenshtein_distance()`. Nous avons utilisé l'algorithme de programmation dynamique présent sur la page Wikipedia. Il fallait faire attention car les indices de boucles n'étaient pas forcément les mêmes, il y'avait au départ un bug (maintenant résolu) au niveau du test d'égalité : on effectuait le test sur les chaines de caractères au mauvais indice (test effectué sur l'indice i au lieu de l'indice $i - 1$).

La distance de Levenshtein étant à valeurs dans $\mathbb{N}$, il a fallu normaliser la distance : c'est-à-dire la ramener à l'intervalle de valeurs $[0, 1]$. Pour cela, plusieurs pistes étaient envisagées. Notons d_L la distance de Levenshtein.

- Première solution : passer à la fonction inverse, c'est-à-dire, prendre $1/d_L$. Cependant, si la distance est grande, alors la distance normalisée devient proche de 0. Or, dans ce cas, nous voulons que elle soit proche de 1.
- La deuxième solution est donc de prendre $1-1/d_L$. On s'affranchit alors du problème précédent. Si la distance est grande, la distance normalisée se rapproche de 1. Si la distance est proche de 1, alors on se rapproche de zéro. Si la distance est égale à 1, la distance normalisée vaut 0 et si la distance est égale à 0, on a un problème de division par zéro, on traitera alors ce cas spécifiquement dans la fonction (en renvoyant directement 0).
- Une autre solution est de prendre une fonction f telle que $f(d_L) = \exp(-Ad_L + A)$ où $|A|$ est suffisamment petit, c'est-à-dire $|A| < 1$. Avec par exemple, $A = \max(l_1, l_2)$ où l_1 et l_2 sont les longueurs de nos segments.

On a finalement opté pour la deuxième solution, elle n'est pas optimale : on peut faire mieux. Mais on a préféré rester sur un normalisation simpliste, accessible. L'implémentation du calcul pour deux segments s'effectue dans la fonction `levenshtein_distance()` et la réalisation de la matrice des distances et du fichier `.pgm` associé s'effectue dans la fonction `levenshtein_distance_matrix()`, de manière analogue à la fonction `dice_distance()`.

6 Couplage des segments : la fonction `minimal_coupling()`

Etant donné notre matrice de distances, on cherche les couples de segments suspects : il faut repérer les segments semblables, à faible distance. On utilise alors un algorithme glouton de couplage des segments de somme minimale.

Nous avons implémenté l'algorithme de couplage minimal en cherchant la valeur minimale de la matrice, si le minimum est atteint en la ligne i et la colonne j, on modifie toutes la valeurs des lignes i et des colonnes j à la valeur 2 : on a pas choisi de mettre la valeur 1 à ces cases car la matrice des distances obtenue précédemment contient éventuellement des 1 (le choix de la valeur 2 n'influe pas les résultats finaux, on remet ces cases à 1 dans l'étape de post-filtrage). On recommence tant que on a un minimum à trouver (par ligne et colonne simultanément). La principale difficulté lors de l'implémentation de cet algorithme est sa terminaison. Lors de nos précédentes tentatives/versions, l'algorithme ne terminait pas forcément et ce, pour plusieurs raisons :

- il faut marquer les minimums sur lesquels on est déjà passés, sinon on peut rester sur le même minimum indéfiniement et l'algorithme boucle indéfiniement. On a fait le choix de modifier la valeur de la case minimale à 2, pour pouvoir la marquer. On sauvegarde néanmoins ses indices afin de ne pas perdre son emplacement.
- il faut faire attention à la terminaison de l'algorithme, on a eu beaucoup de problèmes car dans beaucoup de cas l'algorithme ne terminait pas, quand on a le nombre de colonnes m supérieur au nombre de lignes n (ou réciproquement). L'idée de l'algorithme est de trouver $\min(m, n)$ minimums dans la matrice : à la fin de l'exécution on doit trouver au moins un minimum par ligne et par colonne.

Voici une ancienne version de l'algorithme (il s'agit d'une partie de code ne figurant pas dans le projet, le but de ce paragraphe étant d'expliquer les difficultés rencontrés lors de l'écriture de cette fonction)

La fonction `is_matrix_of_1()` représente les conditions de terminaisons de l'algorithme. La fonction `minimal_coupling()` telle qu'écrite ici, est incorrecte. En fait, elle ne fonctionne pas dans le cas $n > m$ (nombre de lignes strictement supérieur au nombre de colonnes). En effet, la condition d'arrêt de l'algorithme repose sur l'existence d'une unique valeur différente de 1 sur chaque ligne. Mais elle ne prend pas en compte les colonnes. On peut donc avoir plusieurs minimums trouvés sur un même colonne de la matrice (appartenant toutefois à des lignes différentes). Pour que l'algorithme fonctionne, il aurait fallu ajouter à la condition d'arrêt l'existence d'une unique valeur minimale par colonne. De plus, cette fonction `minimal_coupling()` ne donnait pas forcément les bons résultats au départ car le marquage des minimums effectué à l'aide de la liste `list_of_min` était effectué dans la boucle (dans le code ci-dessous, il est maintenant effectué après la boucle). Cela faussait le résultat car l'algorithme loupait des minimums et s'arrêtait trop vite. On aurait peut-être pu se ramener dans le cas $n > m$ au cas $m > n$ qui lui fonctionnait en effectuant une transposée.

Annexe : Version antérieure de la fonction de couplage

```
int is_matrix_of_1(float *C, int lin, int col){
    int num = 0 ;
    for(int i=0; i<lin*col; i++){
        if ((i%(col))==0){
            num = 0 ; // if we change of line, we set the counter at zero
        }
        if (C[i]!=1){
            num ++ ; // we count the number of elements other than 1
        }
        if (num > 1){
            return num+i+2023; // we return a number everytime different of 1
```

```
        }
    }
    return 1; // if at least, the last line elements are all equal to 1 or for every line, there exists
            // exactly one element equal to 1
}

void minimal_coupling(chained_list* list1,chained_list* list2, float* C){
    // il faut appliquer la fonction à la matrice D (obtenue par dice)
    // (au pire en faisant une copie)
    int n=list1->length; // number of lines
    int m=list2->length; // number of columns
    int list_of_min[m*n]; // matrix of marking for minimums
    while(is_matrix_of_1(C,n,m)!=1){ // while there is a non-unique non-1-element in the matrix
        int min_index = 0 ;
        float min = C[0] ;
        for(int i=0; i<m*n; i++){
            if ((C[i] < C[min_index])&&(list_of_min[i]!=1)){
                min = C[i];
                min_index = i; // we want to find the minimal element of the matrix
            }
        }
        list_of_min[min_index]=1; // we mark the index of min AFTER finding the minimum
        printf("\n a: %d \n",min_index);
        int min_i = min_index / m ; // this is the line coordinate of the minimum (using division)
        int min_j = min_index % m ; // this is the column coordinate of the minimum (using modulo)
        for(int j=0; j<m; j++){
            if (j != min_j){
                C[min_i*m+j]=1; // we set all the elements of the line of the minimum (except himself) at
            }
        }
        for(int k=0; k<n; k++){
            if(k != min_i){
                C[k*m+min_j]=1; // we set all the elements of the colonmn of the minimum (except himslef)
            }
        }
    // printf("Affichage de la matrice : \n \n \n");
    //for(int i =0; i<100; i++){
        //printf("%f | ",C[i]);
        //if(((i+1)%5)==0){
            //printf("\n");
        //}
    //}
    //printf("\n is matrix : %d \n", is_matrix_of_1(C, n,m));
    }
}
```

7 Post-filtrage et output des résultats

Dans cette partie, on fait l'hypothèse que les segments isolées des autres ne sont pas révélateur de plagiat, et c'est pourquoi on va calculer une matrice `filtrage.pgm` qui va mettre en valeur les segments adjacents les un aux autres. Cela est implémenté par la fonction `post_filtering()` prenant en les deux listes chainées et les matrices C et F. L'algorithme consiste tout d'abord à calculer tous les coefficients que l'on qualifiera de "internes" de F. Cela veut dire que ces coefficients admettent une diagonale de 5 éléments et peuvent être calculés par la formule donnée dans le sujet. Les autres coefficients, dits "externes", admettent moins de 5 coefficients sur leur diagonale. Nous avons décidé de calculer ces coefficients par la même formule que précédèmment mais en modifiant le nombre d'élèment sommé, et donc le diviseur n'est plus 5 mais correspond au nombre de coefficients disponible sur la diagonale de l'élément (entre 2 et 4).

8 Répartition, avis et complexités des algorithmes principaux

Le travail a été réparti comme suit : le travail de pré-traitement a été réalisé par Kevin, les parties du travail sur la similarité, le couplage des segments ont été réalisés à deux, parfois travaillées séparément et on a pris l'algorithme qui fonctionnait le mieux, qui était le plus efficace. La rédaction du développement a été réalisé par Yousouf et la rédaction du fichier d'expérience a été réalisée par Kevin. La seule partie bonus traitée est celle sur la distance de Levenshtein, nous avons attendu d'avoir un code fonctionnel et de faire des tests avant de faire les parties supplémentaires en bonus. On a également essayé d'être suffisamment exhaustifs sur la documentation du code.

8.1 Complexités

1) **Fonction** `compare_digrammes()` possède deux boucles imbriquées, et on effectue `str1` $\times$ `str2` itérations de boucles. La complexité est donc en $\Theta($`str1` $\times$ `str2`$)$ soit $\Theta(mn)$

2) **Fonction** `dice_distance()` possède deux boucles imbriquées, et on effectue $n \times m$ itérations de boucles dans le meilleur des cas on ne passe jamais dans la fonction `compare_digrammes()`. Si on passe à chaque fois dedans, dans le pire des cas, on effectue `str1` $\times$ `str2` fois `str1` $\times$ `str2` itérations donc la complexité est en $\Theta(m^2n^2)$

3) **Fonction** `minimal_coupling()` possède au maximum trois boucles imbriquées de tailles n, m et $\min(n, m)$: la complexité est donc en $\Theta(mn\min(n, m))$

4) **Fonction** `post_filtering()` possède trois boucles imbriquées de tailles n, m et 5 : la complexité est donc en $\Theta(mn)$