Projet de Programmation en C 2022-2023

ABENSOUR Liam – CREMOUX Mathéo

Sommaire

1.Préambule :

Aujourd'hui la quasi-totalité de l'ideosphère est accessible en seulement quelques clics. Cette abondance d'informations -symptomatique d'une époque qui est la nôtre- génère, a n'en pas douter, autant de problèmes que de solutions. Mais c'est surtout la facilité avec laquelle nous avons accès à cette abondance, qui est le plus à même de générer des dérives. En effet, l'information n'a plus le temps ni d'être digérer, ni d'être comparé et d'être encore moins critiquer. Mais surtout cette dernière peut être copier, et donc se voir approprier sa filiation par une personne qui n'en a même pas compris l'essence.

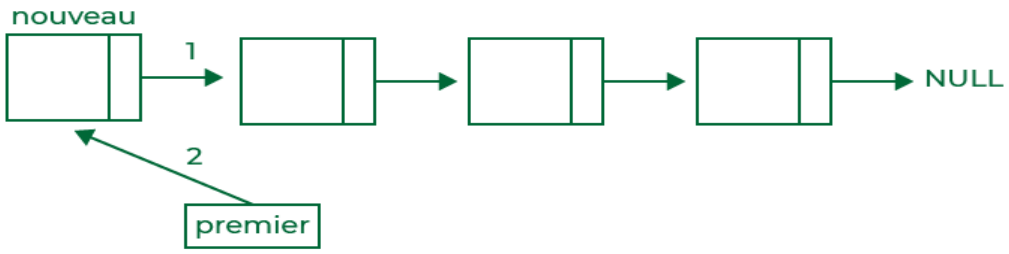
C'est pour cela que la détection de plagiat est un défi qui pour nous - étudiants en proie à la facile combinaison crtl C + crtrl V, et surtout scientifiques de demain - revêt d'une importance capitale.

Le but de ce projet programmation C 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.

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

2.Pré-traitement et Découpage en bloc/segments

Le prétraitement a été une étape assez compliquée à traiter du fait de la multitude des taches à réaliser. Nous avons décidé de l'implémenter de la manière suivante. Tout d'abord nous avons créé une structure de liste chainée dans laquelle chaque élément comprend deux champs : une chaine de caractère qui correspondra à une ligne d'un fichier, et un pointeur sur l'élément suivant. Une fois ces structures mise en place il a été nécessaire d'implémenter des fonctions permettant de les manipuler (comme l'ajout d'un élément ou la suppression de celle-ci). Toutes les fonctions de gestion des listes sont inspirées des méthodes vu en TP avec Monsieur Lazard ainsi qu'un cours de OpenClassrooms (lien suivant).

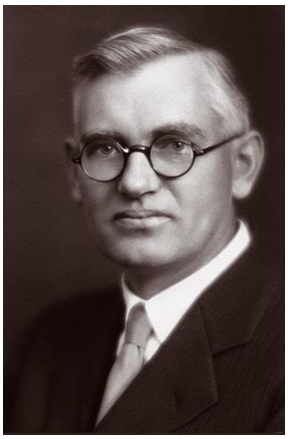


https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19733-stockez-les-donnees-avec-les-listes-chainees

Nous avons alors implémenté les fonctions intermédiaires nécessaire au prétraitement (toute en O(n) où n est la taille de la chaine de caractère). Cela a permis d'implémenter la fonction qui traite les éléments d'une liste chainé. Contrairement à ce qui est suggérer dans l'énoncé, on supprime tout d'abord le contenu des chaines de caractères (une suite de caractère encadrée par des guillemets, et ensuite on supprime les commentaires. Car si on suit l'ordre inverse ce pose le problème de faux commentaires dans une chaine de caractère (printf("//je suis un piège")). Puis remplace chaque lettre par des « w », ensuite on supprime tous les "w" consécutifs : par exemple " if je valide mon année" devient "ww ww wwwwww www wwwww" puis "w w w w w". Puis on supprime enfin les espaces (l'ordre est important car la transformation des mots en w est bien plus difficile si les espaces ont déjà été supprimé). Et pour finir l'étape la plus capitale (et qu'on longtemps sous-estimé) on supprime les lignes vides (ou avec juste un retour à la ligne). Cette étape est essentielle car sinon, les résultats sont compléments biaisés. Par exemple si on ne supprime pas les lignes vides, on obtient une distance finale supérieure à 0.5 entre deux documents identiques .Cet algorithme a une complexité en $O(n^2)$ (où n est la taille de liste chainée).

3.Similarité entre les segments

Lee R. Dice (1887-1977)



On entre alors dans la partie suivante qui consiste à calculer les distances de Dice (ou indices de Sørensen). Nous avons d'abord implémenté une fonction qui permet de compter le nombre de digramme en commun entre deux chaines (en O(n*m) avec n et m les tailles de chaines). Cet algorithme compare toujours la plus petite chaîne à la plus grande par pairs de caractères, en effet après plusieurs testes nous nous sommes rendu compte que le résultat pouvait être erroné si jamais un digramme commun apparait plusieurs fois dans l'une des chaines. Exemple dans «www=w;» et « ww=w;», en comparant la plus grande chaine à la plus petite on obtient 5 digramme commun pour un nombre de digrammes total de 9. La distance sera négative. Ainsi le fait de comparer la plus petite à la plus grande permet de mettre fin au problème. A partir de cette fonction nous avons pu implémenter la fonction qui calculera la distance de Dice entre deux segments obtenus après prétraitement.

Ces fonctions étant créé nous avons pu mettre en place l'algorithme qui, étant donné deux listes chaînées et leur taille (nous avons donc du crée une fonction qui renvoie la taille d'une liste), calculera les distances de Dice entre chaque segment des listes et renverra le résultat sous forme d'un pointeur sur une matrice de format (N x M). Cet algorithme est donc en O(n*m*d) où d est le temps d'exécution de digramme, et n et m les tailles des listes chainées. Pour mieux voir le problème nous avons aussi afficher la matrice afin de vérifier la véracité des algorithmes.

Pour afficher la matrice dans un fichier .pgm nous avons alors crée une fonction qui multiplie les distances par 100 pour les faire correspondre à des pixels, puis formate le fichier renvoyé pour qu'il puisse être interprété correctement. Pour ne pas réécrire une fonction pour le fichier filtrage.pgm nous avons rajouté en argument une chaîne de caractère qui correspond au nom du fichier. En fonction de ce dernier il traite de manière différente le résultat et renvoie un fichier pgm adapté.

4.Couplage des segments

Nous avons ensuite implémenté l'algorithme glouton. Pour cela il a fallu définir une structure trio_seg qui permet de stocker l'indice de ligne, celui de la colonne et la valeur du minimum de la matrice à chaque étape. Ensuite nous avons décidé de modifier dynamiquement la matrice. Ainsi la matrice se réactualise sans cesse à chaque itération du processus décrit dans l'algorithme glouton. Pour faciliter son implémentation chaque minimum était gardé en mémoire (indices et valeur) dans un tableau de trio_seg, dont la taille et le nombre de ligne du fichier le plus court passé en argument. Quand le nombre suffisant de couple est atteint ; il suffit de réactualiser la matrice une dernière fois.

Illustrons cela avec l'exemple suivant :

1ere etape:

0.2 0.3 0.7 1 0.3 0.7

0.1 0.4 0.8 ---->1 1 1 on stock (2,1,0.1) dans la première case du tableau

0.3 0.5 0.6 1 0.5 0.6

2eme etape:

1 1 1

1 1 1 ----> on stock (1,2,0.3) dans la 2 eme case du tableau

1 1 0.6

3 eme etape:

On stock (3,3,0.6) dans la dernière case du tableau. Enfin on parcourt la matrice en modifiant les 1 qui se trouvent à des positions mémorisées dans le tableau.

1 0.3 1

0.1 1 1

1 1. 0.6

5.Post-filtrage

De là nous avons réalisé la fonction qui calculera la deuxième matrice à partir de l'ancienne. Un test sera réalisé pour chaque coefficient, s'il est sur l'extrémité de la matrice alors on ne fera la moyenne qu'avec les valeurs possible. Par exemple pour le premier coefficient de la matrice, à la suite du test on ne calculera la moyenne qu'avec les deux coefficients suivant de la diagonale puisqu'il n'en existe aucun avant. Il suffit alors de parcourir une nouvelle fois la matrice et de mettre à 1 tous les coefficients au-dessus de 0,7. Cet algorithme est en O(n*m) où n et m sont les dimensions de la matrice.

Une approche différente avait d'abord été abordé. Nous voulions séparer les différents cas possibles pour le calcul de la moyenne mais il nous est vite apparu difficile de considérer cette approche. Le nombre de cas à couvrir était assez grand et nous avons donc chercher une solution alternative. En comparant la somme de l'indice de la somme k (sigma) et des indices du coefficient (i et j) de la matrice avec 0 nous avons remarqué que c'était un moyen efficace d'obtenir le résultat escompté.

Pour crée ce deuxième fichier filtrage.pgm nous avons rappelé la fonction qui crée un fichier en passant son nom en argument. Cette dernière traitera sa création différemment puisqu'on désire que le pixel soit d'autant plus clair que le coefficient est petit (0 étant noir et le max étant blanc). On soustraira à 100 le coefficient que l'on aura multiplié par 100. Pour un coefficient à 0.3, on affichera dans le fichier pgm 70 (=100-100*0.3).

6.Output des résultats

Nous pouvons alors calculer la distance finale en calculant d'abord la somme de tous les coefficients de la matrice et stocker la valeur dans une variable pour ensuite utiliser la formule énoncée dans le sujet.

7.Bilan de notre projet

Tout au long de notre projet le maître-mot était « clarté ». En effet conscient de la difficulté du projet et de nos compétences limiter en C ; l'objectif était d'avoir le code le plus claire possible avant de pouvoir repérer aisément les erreurs, mais surtout d'offrir la possibilité à un lecteur avisé de faire évoluer notre projet (par exemple affiner les couplages des segments avec la méthode hongroise).

L'intérêt de ce projet a été double pour nous :

-Il nous a offert une compréhension bien plus précise du langage C, ce qui nous a permis de renforcer nos connaissances. Être confronté à des bugs et d'être obligé de trouver la solution, en ajustant, en corrigeant tel ou tel détail, a été très formateur. Cela permet aussi de comprendre les erreurs auxquelles on peut être confronté (les limites de programmer sur une machine Apple/Windows plutôt que Linux, l'allocation de mémoire et la fuite de celle-ci).

-Le projet étant inscrit dans des sujets de recherche récents c'est avéré très stimulant, car on comprenait aisément les champs d'application de celui-ci.

Il est temps de porter un regard critique sur notre production :

-La gestion de la mémoire fut, pour nous, la tâche la plus difficile. D'où l'importance de programmer sur une machine linux car valgrind n'est plus disponible sur les machines Apple. Mais après plusieurs heures à analyser notre code, nous avons trouvé le bug. Cela nous apporter une compréhension beaucoup plus fine de l'allocation dynamique de la mémoire, et le fonctionnement de la fonction strdup.

-Nous ne nous sommes pas concentrés sur les bonus, car notre objectif était d'avoir un code, certes simple, mais efficace, plutôt que de multiplier les résultats approximatifs.

-L'anglais aurait été plus judicieux pour un projet informatique.

-On a supposé que la taille Max d'une ligne fût inférieure à 20000 caractères (nécessité pour le remplissage de notre liste avec la fonction fgetc d'avoir une taille précise pour le buffer).