PROJET DE PROGRAMMATION C

Rapport de Projet

Sujet:
VAMOS A LA PLAGIAT

Elaboré par:
Sabana SURESH
Oscar PERIANAYAGASSAMY
Groupe de TD6

Encadrants:
Emmanuel Lazard
Florian Sikora

Université Paris Dauphine PSL – L2 MIDO 2022/2023

Remerciements

Nous tenous à remercier M.Lazard et M.Sikora pour vos explications, vos conseils et le temps que vous nous avez consacré pour répondre à nos questions.
Nous vous remercions également car c'est grâce à vous et votre enseignemnent que nous avons la capacité de coder en C et de pouvoir effectuer ce projet.

SOMMAIRE

I. INTRODUCTION

Pour rappel, le projet a pour objectif l'implémentation d'un algorithme permettant la quantification du plagiat entre deux fichiers sources écrit en langage C. La valeur est réelle et comprise entre 0 et 1, 0 assurant une très forte proximité entre les fichiers et 1 l'inverse.

Nous avons décidé de nous partager la tâche en deux en suivant l'ordre proposé par le sujet :

- Sabana : pré-traitement, découpage et mesure de similarité.
- Oscar : couplage, filtrage et affichage.

La dynamique suivie a été de travailler chacun de notre côté sur nos parties respectives pendant la majorité du temps avant de réunir le tout et de discuter les possibilités d'amélioration et de correction de bugs.
On a préféré essayer de résoudre tous les bugs et de vérifier que tout fonctionnait bien avant de réaliser des bonus. Le fait que le programme fonctionne pour différents cas et soit bien correcte et lisible était primordial.

En termes de bonus, nous avons tenté d'en traiter deux : l'implémentation de l'algorithme hongrois (dit de Kuhn-Munkres) et le « bonus affichage » consistant à mettre en couleurs les lignes plagiées. Nous avons également fait le bonus qui consiste à traduire les 27 mots-clés du C par 'm' *(en Annexes p.14)* mais nous n'avons pas eu le temps de l'incorporer au projet.

II. Caractéristiques techniques

a. Structure générale

Trois fichiers composent le projet : $partieSabana.c$, $partieOscar.c$ et $partieOscar.h$. Le $main()$ se situe dans $partieSabana.c$. Cette répartition a été motivée par le fait que chaque partie a été construite indépendamment de l'autre. Il paraît assez naturel de diviser ainsi un projet en binôme afin d'avoir une lecture plus claire du travail de chacun. Le fichier $header$ permet la jointure entre les deux fichiers sources.

b. Principaux algorithmes et leur complexité

i. Calcul de la distance de Dice

Voici l'implémentation de l'algorithme du calcul de la distance de Dice.

```
double distance_dice(char *word1, char *word2) {

  char **list1 = malloc(200 * sizeof(char *));
  char **list2 = malloc(200 * sizeof(char *));

  if (word1[1] == 0 && word2[1] == 0)

    return 0.0;

  digramme(word1, list1);
  digramme(word2, list2);

  int nx = compute_common_digramme(list1, list1);

  int ny = compute_common_digramme(list2, list2);

  int nc = compute_common_digramme(list1, list2);

  double d = 1 - ((double)(2 * nc)) / ((double)(nx + ny));


  liberate_digramme(list1);
  liberate_digramme(list2);

  return d; }
```

On peut remarquer que les complexités qui vont nous intéresser sont celles des algorithmes $digramme()$ et $compute\_common\_digramme()$. La première ajoute dans la liste passée en paramètre tous les digrammes du mot passé également en argument de la fonction. La seconde calcule le nombre de digrammes communs aux deux listes en entrée. Les autres opérations étant constantes, il n'y a pas de raison de s'attarder dessus.

La fonction $digramme()$ a une complexité particulière à calculer. En effet, il s'agit d'effectuer des opérations de temps quasi-constants (allocation d'un espace mémoire pour un pointeur sur $char$ de taille 3 et complétion des valeurs des cases de l'espace) n fois où n est le nombre de parties à deux éléments consécutifs du segment courant. Autrement dit :

$$n = card(S) - 1$$

où S représente le segment courant considéré.

La complexité serait alors dans le cas général : $\boldsymbol{O}(\mathbf{max}\{\ \boldsymbol{card}(\boldsymbol{S})\})$
où S appartient à l'ensemble des segments générés par notre programme.

Or puisque aucune hypothèse n'est faite sur la taille des lignes, il est compliqué de pouvoir estimer cette quantité.

Dans le cas où l'on a des 1-grammes (modification applicable dans le bonus « n-gramme »), le calcul de la complexité reviendrait à trouver la taille du plus grand segment généré. Plus généralement pour des k-grammes, on aurait $n = card(S) - (k - 1)$ où S est le segment considéré.
Posons $A = \ max\{card(S)\}$, S dans l'ensemble des segments généré.

Passons à la fonction $compute\_common\_digramme()$. Brièvement, elle renvoie le nombre de digrammes communs aux deux listes passées en paramètre. Pour cela, elle compte le nombre d'éléments dans chacune et appelle la fonction $digramme\_verif()$ en les mettant en argument. Celle-ci parcourt la plus petite des deux listes et vérifie si le digramme courant est présent dans l'autre liste grâce à la fonction $in()$.

Parlons complexité :

- $in()$ s'exécute en $O(A) + \Theta(1) = O(A)$.
- $digramme\_verif()$ en $A \times O(A) + \Theta(1) = O(A^2)$. La multiplication par A vient du fait que nous exécutons une boucle for avec i variant de 0 à la taille de la liste de digramme courant. Vu que l'on cherche seulement à

majorer, A convient pour le nombre d'itérations. $O(A)$ provient des appels à $in()$ dans la boucle.

Ainsi, $compute\_common\_digramme()$ s'exécute donc en

$$O(A) + O(A) + O(A^2) + \Theta(1) = O(A^2).$$

En combinant ces informations, nous pouvons avancer que la fonction $distance\_dice()$ a pour complexité $O(A) + O(A^2)$ c'est-à-dire $\boldsymbol{O(A^2)}$.

Remarque : on peut dire plus généralement que la complexité est en $O(n^2)$ où n représente la taille des données. Mais il faut faire attention car dans la suite, on ne pourra plus parler du même n. Les données en entrée ne seront en effet plus de la même nature. D'où l'utilisation de A.

ii. Couplage minimal

```
Couple **minimal_coupling(double **matrix, int n, int m, int *couple_counter){
    int min = (n > m)? m : n;
    int max = (n > m)? n : m;
    Couple *C;
    Couple **couple_array = malloc(max*sizeof(Couple));

    while (*couple_counter < min){

        C = find_min(matrix, n, m, couple_array, *couple_counter);
        replace_row_column(matrix, n, m, C);
        couple_array[*couple_counter] = C;
        (*(couple_counter))++;
    }
    return couple_array;
}
```

On suppose ici que la taille des données est représentée par n et m tel que $n < m$ où n est le nombre de segments du plus petit fichier et m celle du plus grand. Au niveau du $running\ time$:

- La boucle $while$ a pour complexité $O(n)$;
- La fonction $find_$min () renvoie le couple d'indices $\{i, j\}$ tel que leur valeur associée dans la matrice des distances soit la plus faible. Dans cette optique, nous avons choisi de parcourir la matrice d'élément en élément (opération en $O(n \times m)$) tout en vérifiant que le couple trouvé n'est pas

déjà répertorié (opération coûtant $O(\frac{n}{2})$ en moyenne). En effet, la vérification consiste à parcourir le tableau de couples auquel on ajoutera un élément à chaque itération de boucle jusqu'à atteindre une taille de n. D'où une complexité $O(n^2 \times m)$ pour cette étape.

- $replace\_row\_column()$ a pour but de parcourir la ligne et la colonne où est présent notre couple de poids minimal afin de la remplir de 1. En résulte un temps d'exécution en $O(n) + O(m) = O(n + m)$.

D'où, $minimal\_coupling()$ s'exécute en

$$O(n) \times \big(O(n^2 \times m) + O(n + m)\big) = O(n^3 m) + O(n^2 + nm)$$

c'est-à-dire en

$$O(n^3 m + n^2 + nm).$$

Si l'on suppose qu'asymptotiquement la différence entre la taille des deux fichiers est négligeable, on peut dire que l'algorithme est en $O(n^4)$ avec n la taille des données.

c. Choix effectués

Plusieurs points du sujet ont conduit à un choix de notre part.

Etape de pré-traitement et de découpage en segment

Pour le pré-traitement des fichiers, nous commençons par retirer les commentaires. Pour cela, nous ouvrons un fichier temporaire nommé $OutputAfterRemovingComments.c$ où nous recopions le fichier original sans les zones commentées. Ensuite, la fonction $lettersToW$ crée un nouveau fichier appelé $OutputAfterReplacingCaractersWithW.c$ dans lequel tous les caractères alphanumériques de $OutputAfterRemovingComments.c$ sont remplacés par le caractère w. Dès que cela est fait, ce dernier fichier est supprimé avec la fonction $remove$ de la bibliothèque $stdio$. Une avant dernière fonction ouvre un nouveau fichier où on recopie le dernier en date tout en ne recopiant qu'un seul w par séquence de w consécutifs. Enfin, les lignes vides sont retirées et le fichier prétraité $FileFinalOutput.c$ est prêt à être stocké sous forme de liste chaînée.

Le principal avantage de cette méthode a été une meilleure détection des potentielles erreurs. En effet, à chaque étape, un nouveau fichier apparaît ce qui nous a permis d'avoir un suivi très précis.

Cependant, elle comporte deux inconvénients majeurs. Elle demande un certain nombre de fonctions pour être appliquée. Il aurait peut-être fallu combiner des étapes (par exemple le retrait de commentaires et de lignes vides) afin d'avoir une meilleure lisibilité sur cette partie du programme. L'autre inconvénient est qu'après la phase de pré-traitement nous n'avons plus aucun lien visible entre les segments et les lignes originales. Cela pose notamment problème pour le bonus « display » où il est demandé d'afficher les lignes originales. Il aurait été judicieux de créer un nouvel attribut à la structure $Line$, qui représente un segment en stockant une chaîne de caractères qui respecte le pré-traitement demandé et l'adresse du segment suivant, afin d'y ajouter la ligne originale du fichier.

Etape de couplage minimal

Le seul choix notable sur cette partie est son implémentation (décrite dans la partie « principaux algorithmes et complexité associé »). On peut par exemple noter que nous avons construit une structure de couple avec deux attributs, correspondant aux indices $\{i, j\}$ dans la matrice de Dice, afin de stocker nos couples de poids minimal dans un tableau de $Couple$.

Etape de calcul post-filtrage

On rappelle que nous avons la formule suivante :

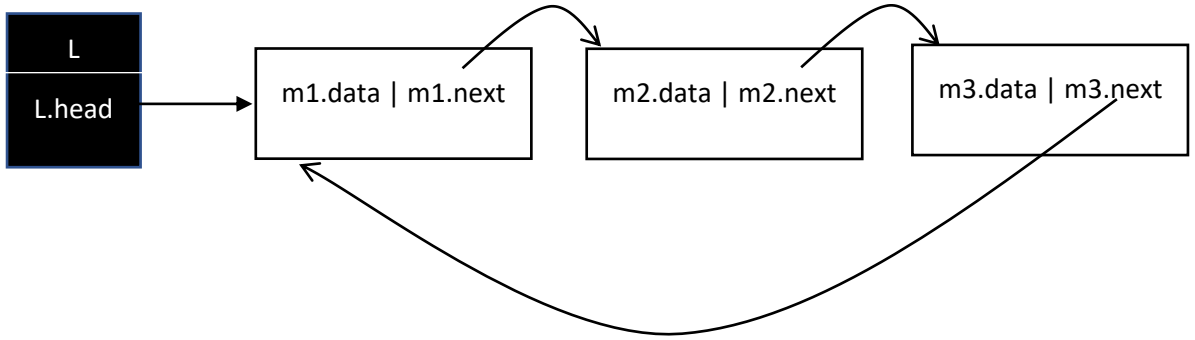
$$f_{i,j} = \frac{1}{5} \sum_{k=-2}^{2} c_{i+k,j+k}$$

Et $f_{i,j} = 1$ si la somme précédente donne une valeur $\geq 0{,}7$.

Nous avons constaté que pour les $f_{i,j}$ tels que $i \notin [\![3, nombre\ de\ lignes - 3]\!]$ ou $j \notin [\![3, nombre\ de\ colonnes - 3]\!]$, le calcul implique l'utilisation de coefficients non-définis car en dehors de l'espace mémoire considéré pour la matrice. Pour combler ce problème, au lieu de multiplier par $\frac{1}{5}$, nous avons introduit un compteur qui contient le nombre de $c_{i+k,j+k}$ « utilisables » pour k variant de -2 à 2. Ainsi, cela a permis d'avoir des bords d'images pgm homogènes par rapport au reste de l'image. Cela était problématique lors de tests sur de petits fichiers notamment au niveau de la distance finale. Sur des données plus conséquentes, ce changement est plutôt négligeable.

Choix des listes chaînées

On a préféré travailler avec des listes chaînées circulaires simples. Cela signifie que si on note L notre liste et m son dernier maillon, alors $m.next = L.head$.



Cela nous a permis une libération de la mémoire plus simple à implémenter. En effet, pour libérer la mémoire allouée pour chaque maillon, il a suffi de parcourir la liste en entière jusqu'à ce que $m.next = L.head$.

Choix du type numérique utilisé

Nous avons décidé de travailler avec des $double$ plutôt que des $float$ afin d'avoir une plus grande précision dans les calculs intermédiaires. Cela a pour principal objectif de contrer autant que l'on peut la propagation d'erreurs d'arrondis.

Concernant la division des étapes en fonctions

Nous avons divisé un maximum le travail que l'on avait en fonctions afin d'avoir une seule fonction principale par fichier source qui ne calcule rien en tant que tel mais qui fait appel à une pluralité de « sous-fonctions ». C'était fortement utile pour le $debug$ et la lisibilité du code. Comme principal inconvénient on peut noter le nombre conséquent de ces sous-fonctions. Dans la majorité des cas où nous avions besoin d'effectuer une tâche particulière, nous avons essayé d'implémenter les fonctions au lieu de chercher si elles existaient dans une quelconque bibliothèque.

Choix pour les expériences

Certains des fichiers utilisés pour cette phase ne nous appartiennent pas. Ils ont soit été généré par $ChatGPT$ d'OpenAI soit viennent du site cours-gratuit.com. Un des fichiers vient également du père d'Oscar : il s'agit d'un projet de C. En plus de ces fichiers, d'autres ont été codés par nous. La validité de tous les algorithmes n'est pas assurée. En effet, il s'agissait surtout de tester notre programme sur des fichiers contenant des lignes de code écrites en C. La logique de leur emplacement ne nous importait que peu (sauf dans une des expériences).

III. Problèmes rencontrés et leur résolution dans…

a. … l'implémentation

Au départ, malgré le fait qu'on ne devait faire aucune hypothèse sur la taille de fichier et de ligne maximale, nous avions implémenté temporairement nos différentes matrices sous la forme de tableaux à deux dimensions statiques. L'avantage était que nous n'avions pas à gérer la libération de la mémoire sur le moment. Leur taille était définie par une taille N en tête de chaque fichier. Ce choix d'implémentation a finalement posé plus de problèmes qu'autre chose lors des phases de tests sur des fichiers de taille « normale » i.e. de plus de 100 lignes. Les erreurs renvoyées sont encore assez abstraites pour nous mais il s'agissait essentiellement d'erreurs de segmentation et d'erreurs concernant le $heap$. Pour contrer ce problème, nous avons alloué dynamiquement des espaces mémoire pour chacune de nos matrices et changé les types d'entrée de nos fonctions ($double\ matrice[N][N]$ est devenu $double\ **\ matrice$).

Une autre erreur de notre part a été de nous dire que le fait de stocker les lignes sous forme de listes chaînées sera traité après que tout soit fonctionnel. Cela signifie que nous avons initialement mis ces lignes dans un tableau alloué dynamiquement et que nous avons travaillé dessus pendant la majorité du temps imparti pour le projet. Le changement a cependant été réalisé et nous a demandé d'être très vigilant lors de la modification des fonctions nécessitant ces listes. Veuillez-trouver ci-dessous le code effectué initialement :

```
void segment(char *File1,char *File2){
  int i;
//splitting File1 in seg1[][]
  printf("\n\tSplitting first file into segments..\n");
  FILE* in = fopen(File1, "r");
  int LinesFile1 = 1;

  char **seg1 = (char**)malloc(LinesFile1*sizeof(char*));

  char line[1000];
  while ((fgets(line, 1000, in)) != NULL){

   seg1[LinesFile1-1] = (char*)malloc((strlen(line)+1) * sizeof(char));
   for (i = 0; i < strlen(line); i++){
     seg1[LinesFile1-1][i] = line[i];}
   seg1[LinesFile1-1][strlen(line)] = '\0';

    LinesFile1 = LinesFile1+1;
    seg1 = realloc(seg1, LinesFile1 * sizeof(char*));}

//splitting File2 in seg2[][]
printf("\tSplitting second file into segments..\n");

  FILE* in2 = fopen(File2, "r");
  int LinesFile2 = 1;

  char **seg2 = (char**)malloc(LinesFile2*sizeof(char*));

  while ((fgets(line, 1000, in2)) != NULL){
   seg2[LinesFile2-1] = (char*)malloc((strlen(line)+1) * sizeof(char));
   for (i = 0; i < strlen(line); i++){
     seg2[LinesFile2-1][i] = line[i];}
   seg2[LinesFile2-1][strlen(line)] = '\0';

    LinesFile2 = LinesFile2+1;
    seg2 = realloc(seg2, LinesFile2 * sizeof(char*));
   }
```

Retour sur les parties de pré-traitement et de transformation en segments. Nous avons rencontré des difficultés à localiser les mots qui devaient être remplacés par $'w'$ et en particulier les caractères. Effectivement, on pensait au début que l'initialisation d'un caractère tel que : $char\ c = 'p'$ était traduit par $ww = 'w'$. Mais $'p'$ est traduit comme une chaîne de caractères. On doit donc supprimer le contenu entre les guillemets. Et cela semble logique car dans notre cas, le fait que le caractère c soit égal à $'p'$ ou à n'importe quel autre caractère, importe peu dans le contexte de détection de plagiat.

De plus un des désavantages des fonctions $letterTOw$ et $removeW$ est leur non capacité à identifier les mots-clés du langage C ($printf$, $scanf$...). Et c'est pourquoi on a essayé de traiter le bonus qui permet de mettre en évidence certains mots-clés. Cependant nous n'avons pas eu le temps de l'intégrer au programme (trop courte distance avec la date butoir de rendu). Pour plus de détail, voir l'annexe.

Autre problème, si dans un fichier on a $'x.next'$, nous nous sommes demandé s'il fallait traduire par $'w'$ ou $'w.w'$. Tout d'abord, d'après la définition sur le site de Larousse, les caractères alphanumériques peuvent être soit alphabétiques (A à Z), soit numériques (0 à 9), soit codés par un autre signe conventionnel (., §, &, ...). Donc nous pensions devoir le traduire par $'w'$. Or sur la page Wikipédia partagé par un des professeurs, on apprend que les caractères de ponctuation ne sont pas des caractères alphanumériques donc le point $'.'$ n'en n'est pas un. Et c'est ainsi pour cela qu'on le traduit par $'w.w'$. Néanmoins, la fonction $isalnum()$ qu'on a utilisé pour cela nous a bien indiqué que le point n'était pas un caractère alphanumérique.
De plus l'avantage de cette fonction est que tous les caractères alphanumériques sont directement repérés grâce à cette fonction, nous n'avons pas besoin de faire des boucles et des conditions pour savoir s'il y a une lettre de l'alphabet (majuscule ou minuscule), s'il y a un chiffre,...

Concernant les bonus, l'algorithme hongrois a nécessité une longue réflexion. En effet, l'explication de Wikipédia n'est pas très claire. Nous avons donc suivi les étapes décrites dans le document à l'adresse suivante :

http://optimisons.free.fr/Cours%20M%C3%A9thode%20Hongroise.pdf

Cependant, cet algorithme est seulement décrit dans des problèmes d'affectation « un à un ». Cela implique que tous les exemples rencontrés concernaient des matrices carrées. Malheureusement, dans notre cas, il n'est pas rare de comparer deux fichiers de tailles différentes. Pour régler ce problème, nous avons essayé de rendre carrée « de force » la matrice avec laquelle on travaillait. Pour cela, nous la complétons avec une valeur strictement plus grande que 1.0 afin que ces coefficients ne soient pris en compte en tant que coefficients de poids minimal qu'à la fin du programme. Malheureusement cette tentative de résolution n'a pas fonctionné et il arrive que l'algorithme boucle indéfiniment dans certaines configurations. D'où l'ajout d'un compteur d'itérations et d'une valeur maximale d'itérations (2000) afin de l'arrêter dans ce cas.

b. …la libération de la mémoire

Du fait de l'utilisation massive de $malloc$ et de $calloc$, il a fallu prêter attention à la libération des espaces mémoire alloués. Ainsi, nous avons utilisé $valgrind$ pour détecter les fuites. Finalement, ces fuites s'élèvent à environ 1 octet sur tout le programme. Il manque au total deux $free$ dans tout le programme (introuvables…).

c. Compréhension

Nous avons eu quelques difficultés de compréhension du sujet par rapport aux digrammes et particulièrement sur le nombre de digrammes communs entre deux chaînes. Au tout début, on pensait qu'un digramme commun aux deux chaînes se répétant était compté qu'une seule fois et non autant de fois qu'il était présent dans les deux chaînes. Nous avions commencé par calculer en prenant en compte qu'il fallait se débarrasser des doublons. Cependant, cette compréhension a conduit à des absurdités dans les distances. Nous avons donc décidé de nous séparer de notre première implémentation au profit de celle actuellement utilisée, qui compte bel et bien les doublons. Car nous avons compris finalement que si le digramme était présent x fois dans les deux chaînes alors on a x digrammes communs et si on avait x≠y alors on a min(x,y) digrammes en commun. Veuillez-trouver ci-dessous le code effectué initialement :

```
// Calculate number of digrams in each string
int num_digrams1 = len1 - 1;
int num_digrams2 = len2 - 1;

// Calculate number of shared digrams
int shared_digrams = 0;
for (int i = 0; i < len1 - 1; i++) {
for (int j = 0; j < len2 - 1; j++) {
if (str1[i] == str2[j] && str1[i + 1] == str2[j + 1]) {
shared_digrams++;
break;
}
}
}
```

d. Problèmes connus mais non résolus

Dans le cas où nous avons deux fichiers de taille différente en entrée, l'écriture dans les fichiers . pgm est codée de telle sorte qu'en sortie, il faut lire les segments de $fichier1$ en ordonné et ceux de $fichier2$ en abscisse. Ce souci peut être réglé par modification de la fonction $generate\_pgm\_files()$. Ou bien par une fonction de transposition de matrice avant l'appel à cette fonction.

On peut mentionner à nouveau l'octet de mémoire non libéré en fin de programme.

Au niveau du bonus d'affichage de lignes originales plagiées, nous n'avons pas respecté la consigne. Nous avons décidé d'afficher les segments plagiés plutôt. En effet, notre implémentation ne permet pas d'avoir un lien direct entre segment et ligne originale. Pour contrer cet inconvénient, il faudrait modifier à l'avenir la structure $Line$, ajouter un attribut entier num indiquant le numéro de ligne associé

et passer les listes chaînées en argument à la fonction $following\_steps()$ (correspondant aux parties 4, 5 et 6 du sujet).

IV. Amélioration et pistes d'ouverture

A posteriori, le projet pourra être amélioré notamment avec l'implémentation de tous les bonus proposés. Notamment celui de la distance de Levenshtein qui pourra compléter l'algorithme hongrois afin d'avoir des distances aussi fines que possible.
Également, il pourra toujours servir dans la suite de notre scolarité si jamais apparaissait le besoin de vérifier le plagiat.

V. Conclusion

Concernant notre ressenti, le projet a été agréable à réaliser. Nous sommes fiers de notre code et nos principaux objectifs ont été atteints. Nous avons fait preuve de patience, de curiosité et d'entre-aide qui ont été des atouts dans l'élaboration du projet.

Nous avons confirmé le fait que la communication est primordiale dans les projets de groupe. Cela nous a notamment permis de mettre en pratique et de consolider toutes nos connaissances du cours de programmation C et même d'en découvrir plus.

VI. Annexes

Veuillez trouver ci-dessous les références des sites web et documentations utilisés pour l'élaboration du projet:

[1] Slide de cours "Programmation C" de Emmanuel Lazard – Dauphine 2022-2023
[2] Polycopié de C (H. Garreta)
[3] Fonction isalnum, koor (https://koor.fr/C/cctype/isalnum.wp)
[4] Utilisation de fflush(stdin) en C, StackLima (https://stacklima.com/utilisation-de-fflush-stdin-en-c/)

Openclassrooms:
[5]Manipulez des fichiers à l'aide de fonctions (https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16421-manipulez-des-fichiers-a-laide-de-fonctions)
[6] Stockez les données avec les listes chaînées https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19733-stockez-les-donnees-avec-les-listes-chainees

[7] OpenAi (https://openai.com)

Wikipédia:
[8] Caractère alphanumérique (https://fr.wikipedia.org/wiki/Caract%C3%A8re_alphanum%C3%A9rique)
[9] Mesure de similarité (https://fr.wikipedia.org/wiki/Mesure_de_similarit%C3%A9)
[10] Indice de Sørensen-Dice (https://fr.wikipedia.org/wiki/Indice_de_S%C3%B8rensen-Dice)
[11] N-gramme (https://fr.wikipedia.org/wiki/N-gramme)
[12] Problème d'affectation (https://fr.wikipedia.org/wiki/Probl%C3%A8me_d%27affectation)
[13] Algorithme hongrois (https://fr.wikipedia.org/wiki/Algorithme_hongrois)

[14] PROBLEMES D'AFFECTATION (http://optimisons.free.fr/Cours%20M%C3%A9thode%20Hongroise.pdf)

[15] De la mesure de similarité de codes sources vers la détection de plagiat: le ”Pomp-O-Mètre” de Romain Brixtel, Boris Lesner, Guillaume Bagan, Cyril Bazin (https://hal.science/hal-01066127)

[16] projet en C pour calculateur de points de championnat en tennis (https://www.cours-gratuit.com/applications-langage-c/mini-projet-en-langage-c-pour-

calculateur-de-points-de-championnat-en-tennis)
[17] Projet de C réalisé en 1997 par le père d'Oscar quand il était à l'université

Voici l'extrait de programme pour le bonus non-traité :

```
// List of keywords to replace with 'm'
#define KEYWORD_CHAR ‘m’
if (strcmp(keyword, "break") == 0 ||
strcmp(keyword, "case") == 0 ||
strcmp(keyword, "char") == 0 ||
strcmp(keyword, "const") == 0 ||
strcmp(keyword, "continue") == 0 ||
strcmp(keyword, "default") == 0 ||
strcmp(keyword, "do") == 0 ||
strcmp(keyword, "double") == 0 ||
strcmp(keyword, "else") == 0 ||
strcmp(keyword, "enum") == 0 ||
strcmp(keyword, "extern") == 0 ||
strcmp(keyword, "float") == 0 ||
strcmp(keyword, "for") == 0 ||
strcmp(keyword, "if") == 0 ||
strcmp(keyword, "int") == 0 ||
strcmp(keyword, "long") == 0 ||
strcmp(keyword, "return") == 0 ||
strcmp(keyword, "short") == 0 ||
strcmp(keyword, "sizeof") == 0 ||
strcmp(keyword, "static") == 0 ||
strcmp(keyword, "struct") == 0 ||
strcmp(keyword, "switch") == 0 ||
strcmp(keyword, "typedef") == 0 ||
strcmp(keyword, "union") == 0 ||
strcmp(keyword, "unsigned") == 0 ||
strcmp(keyword, "void") == 0 ||
strcmp(keyword, "while") == 0) {
memset(&line[i], KEYWORD_CHAR, keyword_len);
}
free(keyword);
i = j;
continue;
}
```