Rapport de projet

Algorithmique 2 – Année 2021-2022

Table des matières

Présentation du projet

L'objectif de ce projet est la réalisation d'un exécutable *ws* (pour *words shared*) qui affiche sur la sortie standard une liste de longueur arbitraire de mots en commun avec au moins deux des sources de fichiers fournies par l'utilisateur via la ligne de commande, selon des options personnalisées modifiant le comportement du programme à diverses étapes (lecture des mots, affichage de ceux-ci).

Le code source de l'exécutable est intégralement réalisé en langage C et ne fait usage que des librairies standard offertes *à une exception près*.

L'exécutable peut être produit en exécutant le *makefile* situé dans le sous-dossier *main* dans l'arborescence du code source.

Pour des soucis de qualité du code, l'ensemble de ces options sont mises en œuvre lors du processus de compilation de chaque fichier source : -std=c18, -Wall, -Wconversion, -Werror, -Wextra, -Wpedantic, -Wwrite-strings et -O2.

Arborescence du code source et modules

Description de l'arborescence

Le code source est subdivisé en plusieurs sous-dossiers : chaque sous-dossier correspond à un module. Un module est une paire (au minimum) composée d'un fichier d'en-tête décrivant l'interface du module ainsi que d'une partie implantation décrivant le comportement du module. Le nom des fichiers source et d'en-tête correspondent exactement au nom du dossier dans lequel ils se trouvent.

Le sous-dossier *main* n'est pas un module en tant que tel : il contient le point d'entrée et la boucle principale du programme. C'est aussi dans ce dernier que se trouve le fichier *makefile* responsable de la production de l'exécutable final.

Tout module, à l'exception du module *options*, est entièrement isolé et ne fait appel à aucun fichier source ou d'en-tête situé à l'extérieur de son périmètre.

Modules impliqués

Différents modules sont impliqués dans le projet, chacun réalisant une tâche bien particulière complémentaire au bon déroulement du programme. Voici une liste ainsi qu'une courte description de ces derniers :

- *hashtable* : module introduit lors de la fiche de TP n°8. Permet la gestion d'une table de hachage de taille variable. L'avantage d'une table de hachage est la possibilité de récupérer et d'ajouter des entrées dans cette dernière en **temps constant** (*O(1)*).

- *holdall* : module également introduit lors de la fiche de TP n°8. Fourretout permettant de garder une trace de tout objet dynamiquement alloué au cours de la vie du

programme. Permet également une itération sur chaque élément qui le compose et donc est d'une grande utilité lorsqu'il s'agit d'afficher les mots lus. Le module est identique à sa version fournie dans les sources du TP n°8 à l'exception de l'implémentation supplémentaire de la fonction *holdall_sort* utilisant un tri fusion pour parvenir à ses fins.

- *options* : module réalisé pour l'occasion permettant la lecture d'un nombre arbitraire d'options. Les options reconnaissables sont définies par l'implémentation. L'interface fournit un alias de type d'une structure capable de contenir l'ensemble des informations pouvant être lues, une fonction réinitialisant ladite structures aux valeurs par défaut, ainsi qu'une fonction lisant dans le tableau d'arguments et affectant les valeurs adéquates à la structure d'options passées en paramètre.

- *reader* : module réalisé pour l'occasion permettant la lecture mot-à-mot dans un flot de fichier donné. La longueur maximale des mots peut être contrôlée, tout comme le comportement face aux caractères de ponctuation ainsi que la possibilité de lire tous les mots en majuscule.

- *shword* : module réalisé pour l'occasion facilitant la gestion de mots partagés. Le module offre via son interface un alias de type contenant l'ensemble des informations sur un mot partagé (mot, nombre d'occurrences et motif d'occurrence dans les fichiers), ainsi qu'un ensemble de fonctions permettant la gestion de la structure. Le fourretout utilisé par le programme ne comporte que des instances de la structure renvoyée par *shword_create*.

Choix des structures de données

La table de hachage

La table de hachage est une structure de données associative : chaque valeur est associée à une clé servant à l'identifier. Le principal avantage de la table de hachage est sa complexité pour les opérations d'insertion et de récupération : en effet, ces opérations se déroulent en **temps constant** lorsque la table est bien implémentée (tableau suffisamment large et bonne diffusion par la fonction de hachage).

Dans le cadre du projet, la table de hachage nous est très utile pour lier une chaine de caractères dénotant un mot à sa structure *shword* associée. Ainsi, on peut récupérer la structure (ou à défaut la créer) en temps constant lorsqu'on lit un mot. La fonction de hachage utilisée est identique à celle de la fiche de TP n°8 :

```c
size_t str_hashfun(const char *s) {
  size_t h = 0;
  for (const unsigned char *p = (const unsigned char *) s; *p != '\0'; ++p) {
    h = 37 * h + *p;
  }
  return h;
}
```

Figure 1 : Fonction de hachage utilisée par la table.

Le fourretout

Le fourretout est quant à lui utilisé pour stocker les structures *shword* initialisées lors de la lecture des mots. L'implantation du fourretout est réalisée sous forme de liste chainée, ce qui en fait un très bon candidat pour itérer dessus. Ainsi, l'affichage des mots peut se faire facilement en itérant sur le fourretout correctement trié à l'aide de la fonction *holdall_sort* dont la fonction de comparaison passée en paramètre est *shword_compare*.

Voici un aperçu de la fonction de comparaison de la structure *shword* :

```c
int shword_compare(const shword *shw1, const shword *shw2) {
  size_t filecount = shword_filecount( shw: shw2) - shword_filecount( shw: shw1);
  if (filecount) {
    return (int) filecount;
  }
  SHW_OCCURRENCES_TYPE occurrences = shword_occurrences( shw: shw2)
      - shword_occurrences( shw: shw1);
  if (occurrences) {
    return (int) occurrences;
  }
  return strcmp(shword_word( shw: shw1), shword_word( shw: shw2));
}
```

Figure 2 : Fonction de comparaison utilisée par le tri du fourretout.

Comme explicité dans la spécification de la fonction, trois clés sont utilisées pour comparer deux objets issus de la structure. Premièrement, on vérifie le nombre de fichiers dans lequel les deux mots partagés apparaissent. Deuxièmement, on vérifie le nombre d'occurrences global des deux mots partagés. Enfin, dernièrement, on compare les chaines de caractères à l'aide de la fonction standard *strcmp*. Cette fonction nous permet d'être en accord avec les contraintes fournies dans le sujet et, par conséquent, permet un affichage cohérent des mots.

Principe de fonctionnement

Boucle principale

Le programme peut être résumé en un algorithme décrivant les opérations effectuées dans les grandes lignes :

```
options <- récupérer les options et les fichiers de la ligne de commande

pour chaque fichier spécifié dans les options faire :
        pour chaque mot de fichier :
                word <- recherche du mot dans la table de hachage
                si word vaut NULL faire:
                        struct <- nouvelle struct shword pour mot
                        insérer struct dans le fourretout
                        lier le mot à struct dans la table de hachage
                fin
                incrémenter le nb. d'occurrences de word
        fin pour
fin pour

trier le fourretout selon la fonc. de comparaison de la structure shword

afficher les mots partagés du fourretout selon la fonc. d'affichage de la structure
shword
```

Gestion d'un mot partagé

La structure *shword* comporte quatre champs :

- Une chaine de caractères dénotant le **mot concerné**.
- Un compteur du **nombre d'occurrences du mot**.
- Un entier comportant le **motif d'occurrences du mot**.
- Un compteur du **nombre de fichiers** dans lequel le mot est **marqué présent**.

Lors de l'opération d'incrémentation,

- Si le mot n'apparait pas dans le *k*-ième fichier (son bit vaut zéro) :
  - Le compteur du nombre de fichiers est incrémenté d'une unité.
  - Le *k*-ième bit se voit affecter la valeur un.
- Si la limite d'occurrences n'a pas été atteinte :
  - Le nombre d'occurrences du mot est incrémenté d'une unité.

La chaine de caractères n'est nullement modifiée au cours de la vie du programme.

Algorithme de tri : le tri fusion

Pour garantir un affichage cohérent des mots partagés, il est nécessaire de trier le fourretout contenant ces derniers selon une fonction de comparaison donnée. Cette fonction est fournie par le module *shword* et se nomme *shword_compare* (prenant deux objets *shword* en paramètre).

L'implantation de l'algorithme de tri du fourretout reste à faire : le fourretout est fondamentalement une liste simplement chainée. Pour cette structure de données, le **tri fusion** semble être le plus adapté. Reprenant le principe de « diviser pour mieux régner », il permet de trier une liste simplement chainée en un temps logarithmique *n*log n*. De plus, elle permet de s'affranchir de toute allocation mémoire puisqu'elle agit directement sur la liste à trier.

L'algorithme peut être résumé en quatre grandes étapes :

1. Si la liste ne contient qu'un élément, elle est considérée triée.
2. Sinon, diviser la liste en deux sous-listes de taille au mieux égales, au pire à une unité près.
3. Exécuter l'étape 1 pour chacune des deux sous-listes.
4. Fusionner les deux sous-listes en insérant en queue de la liste résultante soit la tête de la 1e liste, soit la tête de la 2e liste en fonction de laquelle des deux sous-listes contient la plus petite valeur en tête.

À la fin de l'exécution de l'algorithme, la liste chainée reconstituée est triée selon les exigences de la fonction de comparaison fournie. On utilise un pointeur de pointeur vers un *choldall* dans la fonction *choldall__merge* afin d'éviter une récursivité non terminale lors de la reconstruction de la liste finale. Ainsi, on n'explose pas la pile d'exécution lorsque la chaine à reconstituer dépasse une certaine taille.

Limites du programme

Nombre d'occurrences

Le nombre d'occurrences d'un mot dans un fichier est borné par la valeur de la macro-constante **SHW_OCCURRENCES_MAX** qui, par défaut vaut *ULONG_MAX* (la valeur maximale que peut contenir un entier du type *unsigned long*), mais qui peut être arbitrairement choisie. Pour un bon fonctionnement, il est demandé que la valeur de cette macro-constante puisse être contenue par le type contenu par la macro-constante **SHW_OCCURRENCES_TYPE**. Lors de l'affichage, si le nombre d'occurrences est égal à la limite, alors la chaine contenue par la macro-constante **SHW_OCCURRENCES_MANY** est affichée à la place.

Nombre de fichiers

Le nombre de fichiers pouvant être pris en charge simultanément est borné par la capacité en bits du type contenu par la macro-constante **SHW_PATTERN_TYPE**, cette valeur peut être obtenue à l'aide de la macro-constante **SHW_PATTERN_MAX**. Si l'utilisateur fournit plus de fichiers que **SHW_PATTERN_MAX**, alors le programme s'arrêtera.

Nombre de caractères initiaux

Le nombre de caractères initiaux dans un mot **ne peut pas** être de longueur illimitée. L'option *initial* se comportera comme un compteur de mots si la valeur zéro lui est affectée via la ligne de commande. Le nombre de caractères initiaux maximal accepté est la taille en bytes que *malloc* acceptera d'allouer + un.

Notice d'utilisation

La notice d'utilisation est intégrée au programme à l'aide de l'option longue *--help* (ou courte *-?*). La syntaxe attendue y est décrite ainsi que l'ensemble des options acceptées par le programme, si elles demandent un argument, ainsi qu'une courte description de leurs conséquences sur le comportement du programme.

Au cas où la ligne de commande ne pourrait pas être traitée, une erreur explicite est affichée sur la sortie d'erreur pour faciliter la compréhension du problème.

Exemples

Validité des options

Comme attendu, le programme fonctionne convenablement et produit le même affichage que les jeux d'essais fournis dans le sujet du projet. Ici, une reproduction de l'exemple utilisant les cinq options du programme avec les trois textes *toto* :

```
[debian@vps-f06f73b4:~/main$ ./ws -t 13 -s -i 7 -p -u toto0.txt toto1.txt toto2.txt \ ]
[> toto3.txt                                                                        ]
ws: Word 'MAITRES...' was truncated in file 'toto0.txt'.
ws: Word 'CONJUGU...' was truncated in file 'toto0.txt'.
ws: Word 'MAITRES...' was truncated in file 'toto1.txt'.
ws: Word 'CONJUGU...' was truncated in file 'toto1.txt'.
ws: Word 'PREMIER...' was truncated in file 'toto1.txt'.
ws: Word 'PERSONN...' was truncated in file 'toto1.txt'.
ws: Word 'MANGERA...' was truncated in file 'toto1.txt'.
ws: Word 'MAITRES...' was truncated in file 'toto2.txt'.
ws: Word 'CONJUGU...' was truncated in file 'toto2.txt'.
ws: Word 'PERSONN...' was truncated in file 'toto2.txt'.
ws: Word 'MAITRES...' was truncated in file 'toto2.txt'.
ws: Word 'INTERRO...' was truncated in file 'toto2.txt'.
ws: Word 'ACCELER...' was truncated in file 'toto2.txt'.
ws: Word 'MAITRES...' was truncated in file 'toto3.txt'.
ws: Word 'POURRAI...' was truncated in file 'toto3.txt'.
ws: Word 'MIGRATE...' was truncated in file 'toto3.txt'.
xxxx    9       TOTO
xxxx    8       JE
xxxx    6       LA
xxxx    5       DEMANDE
xxxx    5       MAITRES
xxx-    5       A
xxx-    3       CONJUGU
xxx-    3       DE
xxx-    3       LE
xxx-    3       VERBE
x-x-    5       IL
x--x    4       QU
-xx-    3       ET
x-x-    3       LUI
--xx    3       TU
```

Figure 3 : Exemple d'utilisation des cinq options.

Tous les mots, transformés en majuscule grâce à *-u*, lus n'ayant pu être conservés entièrement (longueur supérieure à 7 caractères) dû à l'option *-i 7* sont signalés sur la sortie d'erreur avec leur fichier source. En dessous, nous retrouvons les 13 premiers mots demandés à l'aide de *-t 13* ainsi que les deux derniers ayant le même nombre d'occurrences, effet de l'option *-s*. Et enfin, aucun caractère de ponctuation ne figure sur la liste puisqu'ils ne sont plus considérés comme faisant partie d'un mot ou étant un mot à part entière puisque l'option *-p* est activée.

Performances générales

Avant d'effectuer des tests grandeur nature, nous allons d'abord évaluer la complexité du programme :

- La lecture, et l'enregistrement éventuel, des mots depuis les fichiers se fait en temps linéaire *n* *(la recherche dans une table de hachage se fait en temps constant lorsque bien réalisée, ou au pire linéaire)*.
- Le tri se fait en temps « linéarithmique » (linéaire * logarithmique) *n*log n*.
- L'affichage des mots se fait en temps linéaire *n*, voire plus précisément *t + x* où *t* est la valeur de l'option *top* et où *x* est potentiellement différent de zéro dans le cadre où l'option *same-numbers* est activée. Plus généralement, elle reste linéaire.

Globalement, ce programme est de complexité linéarithmique puisque c'est la complexité la plus « grave » qui soit présente. Les performances attendues devraient être correctes pour des fichiers de grosse taille. On peut ici réaliser un essai avec le texte *lesmiserables.txt* mis en opposition aux quatre textes *toto* déjà utilisés auparavant.

```
kevin@MacBook-Air-de-Kevin main % time ./ws lesmiserables.txt toto0.txt toto1.txt \
toto2.txt toto3.txt
ws: Word 'paroisse-meurs-de-faim-si-tu-as-du-feu-meurs-de-froid-si-tu-as-...' was tru
ncated in file 'lesmiserables.txt'.
ws: Word '--L'agent-de-police-Ja-vert-a-ete-trouve-noye-sous-un-bateau-du...' was tru
ncated in file 'lesmiserables.txt'.
xxxxx   1334    La
xxxxx   72      demande
xxxxx   18      maitresse
xxxx-   21437   de
xxxx-   11293   a
xxxx-   10926   le
xxx-x   1666    je
-xxxx   7       -
-xxxx   7       Toto
-xxxx   6       :
./ws lesmiserables.txt toto0.txt toto1.txt toto2.txt toto3.txt  0,13s user 0,01s syst
em 98% cpu 0,138 total
```

Figure 4 : Test de performance incluant Les Misérables.

Le texte des Misérables étant particulièrement volumineux (2.9Mo), cela donne une certaine idée des performances globales du programme. Le temps de traitement est donc de 138 millisecondes ce qui semble relativement convenable pour l'entrée fournie.