Compte rendu
Projet Algorithmique 2

Présentation des différents modules

L'archive est composée de 5dossiers différents, 3modules, le main et une extension du main nommée wordstruct.
Le premier, hashtable, nous permet de gérer efficacement des tables de hachage en lui donnant une bonne fonction de hachage.
Le deuxième, holdall, nous permet d'avoir un fourre tout afin, à la fin de notre programme, de pouvoir afficher ainsi que libérer notre résultat.
Ce module utilise lui même le module bunch afin de s'assurer que l'utilisateur lui donne bien des adresses valides, qui correspondent à des objets holdall défini via holdall_empty, pour pouvoir éviter une éventuelle erreur de segmentation. En effet, ce module récupère les différentes adresses que holdall peut créer, puis ensuite vérifie que celle que l'utilisateur donne en fait partie.
Le main, quant à lui, permet de récupérer sur des fichiers ou l'entrée standard une série de mot qui sont placés dans notre table de hachage, mais également dans notre holdall, sous forme d'une structure, puis affiche le résultat obtenu, dépendamment des options utilisées.
Pour cela, nous utilisons directement notre dossier ./wordstruct qui nous permet de gérer tout ce qui est lié à cette structure. Ainsi, nous pouvons facilement utiliser les différents modules ainsi que notre structure.
De plus, le module getopt est également utilisé afin de gérer les options efficacement.

Implantation

Pour réaliser ce programme, j'ai d'abord utilisé le module hashtable et holdall. Pour cela, j'ai d'abord créé les versions « empty » de chacun des différents « modules ». Ensuite, nous devons alloué différentes variables int, qui serviront plus tard pour avoir des variables int qui seront les mêmes dans tous les différents éléments du type de ma structure. max_word, par exemple, correspond au nombre de mot restant à afficher. Les 2autres seront expliqués plus tard.

Après, cela correspond à l'implantation des options. Pour ce faire, de nombreuses variables booléennes sont crées afin de savoir si l'option est activée ou non. Puis grâce au module getopt, nous récupérons les différentes options activés ou non, qu'elles soient longues ou non. Pour cela, nous utilisons un tableau dynamique qui

nous dit toutes les options longues existantes, avec si besoin un argument ou non. De plus, quand on appelle getopt_long, on lui donne une chaîne de caractère qui correspond à toutes les options courtes existantes, avec si besoin le caractère : pour dire qu'il faut un argument. En fonction de l'option, il faut faire plus ou moins d'action. Certains ne nécessite que de changer un true en false ou inversement, mais d'autre nécessite de changer plus de chose. Par exemple, si -i vaut 0, alors il ne faut plus couper les mots donc notre variable is_word_cut passe à false, cependant si il est différent de 0, il faut changer word_length_max, qui correspond à la limite maximale d'un mot, en changeant de la valeur de défaut à la valeur donnée par l'utilisateur. Si -i vaut 0, on ne change pas cette valeur par défaut car de toute façon celle ci sera augmentée si besoin plus tard.

Puis nous allouons pour une chaîne de caractère w, qui nous servira à stocker tous nos mots dedans, le nouveau mot écrasant l'ancien. Ensuite, nous passons à l'ouverture des différents fichiers. Pour cela, on fait une boucle sur le nombre de fichiers totaux, c'est à dire argc, sans pour autant commencer à 1, car il ne faut pas prendre en compte les options déjà lu. Pour cela, getopt nous donne l'indice du premier élément qui n'est pas une option, le module plaçant toutes les options au début si besoin. Ainsi, on récupère le nom du fichier mais on ne l'ouvre pas directement. En effet, il faut d'abord vérifier qu'il ne correspond pas à la chaîne « - » car sinon, il faut lire l'entrée standard ! Si c'est le cas, on utilise également la fonction rewind afin de réinitialiser une entrée potentiellement déjà utilisé. Sinon, on ouvre simplement le fichier.

Après cela, nous commençons par lire un mot. Pour cela, nous utilisons la fonction read_word, qui doit prendre l'adresse de notre chaîne allouée précédemment, mais également l'adresse de la taille maximale d'un mot, le flux courant, ainsi que des informations sur l'activation ou non des différentes options.

Read_word :
Cette fonction permet de lire un mot caractère par caractère, grâce à la fonction fgetc. De plus, on utilise également un compteur k afin de savoir où nous sommes actuellement dans le mot. Puis nous faisons une boucle while, tant que nous ne sommes pas à la fin du fichier, où qu'il n'y a pas d'espace on continue. Également, on rajoute le fait que si l'option -p est activée, on doit considérer la ponctuation comme un espace donc si on trouve un caractère de ce type, on sort de la boucle. Ensuite, nous plaçons le caractère lu au bon endroit dans la chaîne grâce à notre compteur k.
Ensuite, on vérifie si on a dépassé la limite. Si c'est le cas, si on ne doit pas couper les mots, on réalloue notre mot pour doubler sa taille, et on double aussi notre variable de limite de taille. Sinon, on sort de la boucle.
Une fois sortie de la boucle, on rajoute notre caractère fin de chaîne. On vérifie ensuite si jamais on est à la fin du fichier, si c'est le cas on renvoie 0. Cela nous permet de savoir dans le main quand quitter notre boucle pour lire tous les mots. Si on a lu une chaîne vide, on renvoie 2. C'est notamment le cas si l'option -p est

activée, et cela permet grâce à une boucle de « supprimer » tous les mots lus qui sont des chaînes vides et donc qui ne doivent pas être pris en compte . Ensuite, si l'option -u est activée, on mets toute notre chaîne en majuscule. C'est plutôt simple, il suffit d'enlever 32 à tous les caractères compris entre a et z.
Finalement, il nous faut éliminer toute la partie après le mot si celui ci à été coupé, qui ne forme donc pas un mot. Pour cela, on recrée exactement notre boucle au dessus en simplifié, simplement on lit les caractères sans rien en faire, ceux ci sont supprimés. Finalement, on renvoie 1, pour dire que notre mot a bien été lu, et qu'il n'est pas à la fin du fichier.

De retour dans le main, on vérifie la valeur de retour de notre fonction. Si jamais elle vaut -1, il y a eu une erreur de capacité donc on doit sortir. Si elle vaut 2, on relit des mots en boucle jusqu'à avoir une valeur différente. Sinon, on continue dans notre boucle pour lire tous les mots.

On commence par vérifier si notre mot est dans notre table de hachage. Si ce n'est pas le cas, on alloue une chaîne de caractère qui aura exactement la même valeur que w. Ensuite, nous appelons la fonction hold_empty, afin de créer un nouvel élément de notre structure, en lui donnant tout ce dont il a besoin, puis on l'ajoute à notre holdall et à notre hashtable en valeur, avec comme clé notre chaîne toute simple. Sinon, on mets simplement à jour notre mot grâce à la fonction hold_maj.

Une fois ça fait, on refait notre boucle, en lisant à nouveau un mot, et en faisant tout nos tests.

Explication de ./wordstruct :

Ce dossier nous permet de directement gérer la structure que l'on passe à notre table de hachage et à notre holdall. Cette structure est composée de nombreux champs :
-un champ s correspondant à une chaîne de caractère qui est le mot lu
-un champ file qui est une chaîne de caractère également qui est le motif, décrivant la présence dans les fichiers
-un champ oldFileNumber, qui sert à nous dire dans quel fichier le mot a été lu en dernier
-un champ occurrence, pour nous dire le nombre ce de mot lu dans tous les fichiers/entrée standard
-un champ ninfile, pour nous dire dans combien de fichiers différents/entrée standard le mot a été lu.
-un champ maxw, commun à tous les mots étant alloué au début du main, pour dire combien de mot il reste à afficher (si une infinité, il est <0)
-un champ nmaxfile pour dire le nombre de fichier max
-un champ occlast pour connaître le nombre d'occurrence du dernier mot affiché (par défaut vaut 0)

-un champ niflast pour connaître le nombre ninfile du dernier mot affiché (par défaut vaut 0)
-un champ issamenumber pour savoir si l'option -s est activée

Hold_empty :
Cette fonction permet de créer un élément de la structure hold, en lui donnant tous les paramètres nécessaires en entrée. Pour cela, on alloue notre contrôleur de structure, puis on alloue notre chaîne file. On l'initialise avec la chaîne vide, afin d'éviter un potentiel problème de mémoire, on ne sait pas ce qu'il y avait avant à l'endroit alloué. Ensuite, en utilisant le paramètre k, on peut savoir le numéro actuel du fichier où on se trouve, et donc on sait que notre chaîne de 1 à k-1 doit être composé de tiret. L'élément à l'indice k doit être lui un x, car il est bien dans ce fichier. Ensuite, pas besoin d'allouer s, car celui ci est déjà alloué précédemment dans le main. Nous avons juste à compléter les différents champs en fonctions des paramètres. Les champs occurrences et ninfile eux sont mis par défaut à 1. Plus qu'à renvoyer h ensuite !

Hold_maj :
Cette fonction permet de mettre à jour un élément de hold. Pour cela, on augmente son nombre d'occurrence de 1, puis si le numéro du nouveau fichier passé en paramètre est différent de oldFileNumber, on mets à jour notre chaîne file, on augmente ninfile de 1 et on change oldFileNumber pour mettre notre paramètre à la place. Pour mettre à jour notre chaîne file, il suffit de rajouter, de notre oldFileNumber + 1 (avant qu'il soit mis à jour) à notre k-1 des tirets (dans le code le +1 est enlevé et un -2 apparaît de l'autre coté, cela revient au même), puis on ajoute à l'indice k notre x.

Holdcmp :
Cette fonction compare deux éléments du type hold, d'abord sur leur nombre de fichiers, puis si nécessaire sur leur nombre d'occurrence et enfin si nécessaire sur l' ordre lexicographique.

scptr_display :
Cette fonction affiche l'élément hold qu'elle reçoit en paramètre. Cependant, elle ne l'affiche que si ninfile est supérieur strictement à 1, c'est à dire si le mot est présent dans plus d'un fichier. Ensuite, si jamais le nombre de mot restant à afficher est différent de 0, alors on diminue de 1 cette valeur qui pour rappel est commune à tous les mots, puis on mets à jour une dernière fois notre champ file. Pour cela, de oldFileNumber + 1 à nmaxfile, on ajoute des tirets. C'est à dire que du fichier suivant après la dernière fois que l'on a lu notre mot jusqu'au nombre de fichier maximal, on ajoute des tirets, le mot n'ayant pas été trouvé dans ces fichiers. Ensuite, on mets à jour nos champs occlast et niflast pour mettre les valeurs du mot actuel, celui ci étant le dernier affiché actuellement, puis on affiche le mot. Si jamais le nombre de mot restant à afficher vaut 0, si l'option -s n'est pas activé on n'affiche plus rien, on renvoie 0. Cependant si elle est activée, si jamais notre mot actuel à autant

d’occurrence et est présent dans autant de fichier que le dernier mot lu, alors on l’affiche en mettant à jour son champ file comme précédemment.

FreeVal :
Cette fonction libère le champ s de l’élément hold en paramètre, puis son champ file, puis lui même. Ainsi, tout ce qui est associé à cet élément est libéré.

De retour dans le main, une fois sortie de la boucle pour lire tous les mots, on vérifie si notre fichier vaut stdin. Si ce n’est pas le cas, on peut ainsi fermer notre fichier, et vérifier que l’on a pas d’erreur, par exemple si on est pas arrivé à la fin du fichier. On fait ce test car fermer l’entrée standard veut dire ne plus pouvoir la rouvrir dans la suite du code.

Puis, sorti de toutes les boucles, nous réalisons le tri de notre holdall. Pour cela, nous utilisons la fonction holdall_sort, à qui nous donnons une fonction de comparaison spécial défini dans wordstruct.c. Ainsi, nous sommes capable de comparer sur les différents champs de hold sans problème.

Puis, en utilisant holdall_apply, nous pouvons afficher chaque élément de notre fourre tout en utilisant notre fonction fonction d’affichage vu précédemment.

Ensuite, il s’agit de la partie avec toutes les erreurs. Si il n’y en a pas eu, on passe direct à la zone dispose, sinon un message d’erreur est affiché puis après on passe à dispose.

Pour libérer toute notre mémoire, il suffit d’utiliser encore une fois holdall apply avec cette fois une fonction pour libérer notre structure, freeval. Ensuite, on libère nos différents champs qui étaient communs à tous les mots, on libère notre mot w puis notre table de hachage et enfin notre holdall. Ainsi, tous les éléments de notre programme sont bien libérés. En effet, on libère bien notre mot, et sa structure associée. Il n’y a pas besoin de libérer les valeurs des mots de hashtable, ceux ci étant exactement les mêmes que ce que l’on donne à holdall, donc libérer holdall libère aussi ceux la.

Algorithme général

initialiser (holdall)
initialiser(hashtable)
initialiser(options)
pour chaque fichier texte fichier faire

pour chaque mot w selon les options dans fichier faire
si w est un nouveau mot pour hashtable alors
initialiser(nouvel élément de la structure hold)
ajouter cet élément dans holdall
ajouter w et cet élément en valeur dans hashtable
sinon
mettre à jour l'élément de la structure associé au mot déjà existant
trier(holdall)
afficher(holdall)
libérer (holdall)
libérer (hashtable)

Exemple

Premier exemple : on teste d'abord sur énormément de fichiers textes, avec toutes les options possibles sauf t, sans qu'un mot puisse être coupé avec valgrind et time. On obtient le test suivant.

```
==5987== Memcheck, a memory error detector
==5987== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5987== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5987== Command: ./ws ../../textes/lesmiserables.txt ../../textes/lavare.txt ..
/../textes/lebourgeoisgentilhomme.txt ../../textes/surleboutdesdoigts.txt ../../
textes/toto1.txt ../../textes/toujoursetjamais.txt ../../textes/lecid.txt ../../
textes/moilhiverjepense.txt ../../textes/phedre.txt ../../textes/toto0.txt ../..
/textes/tartuffe.txt ../../textes/toto2.txt ../../textes/toto3.txt -i 0 -s -p -u
```

Cet appel donne comme résultat :

```
xxx-xxxxxxxxx   18995   LA
xxxxx-xxxxxxx   4725    JE
xxxx-xxxxxxx-   8980    LES
xxx-xxx-xxxx-   15331   A
xxx-xxx-xxxx-   14237   LE
xxx-x-x-xxxxx   141     DEMANDE
xxx-x-x-xxxxx   45      MAITRESSE
xxx-x-x-xxxx-   24913   DE
xxx-xxx-x-xx-   16371   ET
xxx--xxxx-xx-   12591   L
==5987==
==5987== HEAP SUMMARY:
==5987==     in use at exit: 0 bytes in 0 blocks
==5987==   total heap usage: 150,248 allocs, 150,248 frees, 4,669,151 bytes allo
cated
==5987==
==5987== All heap blocks were freed -- no leaks are possible
==5987==
==5987== For lists of detected and suppressed errors, rerun with: -s
==5987== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

On voit bien que tous ce qu'y a été alloué a été libéré, et qu'effectivement comme prévu aucun mot n'a été coupé (pas de message d'avertissement), tout est bien en

majuscule, les caractères de ponctuation sont bien considérés comme des espaces (d'où le L seul, qui correspond en réalité à l'). On voit également que les mots sont bien triés comme prévu.
De plus, ce résultat est obtenu très rapidement :

Deuxième exemple, on teste sur les fichiers toto :

```
==6962== Memcheck, a memory error detector
==6962== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6962== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6962== Command: ./ws -t 13 -s -i 7 -p -u ../../textes/toto0.txt ../../textes/t
oto1.txt ../../textes/toto2.txt ../../textes/toto3.txt
==6962==
```

On obtient le résultat suivant :

```
*** Warning: Word 'INTERRO...' sliced.
*** Warning: Word 'ACCELER...' sliced.
*** Warning: Word 'MAITRES...' sliced.
*** Warning: Word 'POURRAI...' sliced.
*** Warning: Word 'MIGRATE...' sliced.
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
```

On voit que les mots sont effectivement bien coupés comme prévu. De plus, on voit que l'option -s marche bien car il y a 15mots affichés au lieu de 13normalement, et les deux supplémentaires sont bien présents dans autant de fichier avec le même nombre d'occurrences que le dernier affiché « normalement ».

Limitations

Le nombre de malloc de mon programme est extrêmement élevé et pourrait être largement optimisé.
De plus, on pourrait imaginer une implantation différente de ma chaîne file avec une chaîne qui vaut par défaut que des tirets et à qui on mettrait des x aux endroits nécessaires.

Finalement, certaines parties du code pourrait être mieux écrite afin de faciliter la compréhension et d’éviter les répétitions.