Rapport projet d’algorithmie

L2 Informatique – Université de Rouen

Année : 2022-2023

Fait par : Clerc Leo et Belhocine Juba

Sommaire

I-Introduction

1) Présentation du sujet

Le sujet nous demande de développer un programme en langage C, ce programme doit être capable de lire des noms de fichiers passés en ligne de commande et d'effectuer des traitements sur ceux-ci.

Il y a deux cas à distinguer :

-Si un seul nom de fichier est passé en paramètre le programme recherche les lignes de texte qui apparaissent au moins deux fois dans le fichier et affiche les numéros de lignes correspondants ainsi que le contenu commun associé à ces lignes. Par exemple si la phrase « toto mange » est à la ligne 15 et 43 alors l'affichage sera :

15, 43 --) toto mange

-Si au moins deux noms de fichiers figurent sur la ligne de commande, il faut afficher pour chaque ligne de texte non vide apparaissant au moins une fois dans tous les fichiers leurs occurrences dans chacun de ces fichiers suivis du contenu de cette ligne. Par exemple si la phrase « toto boit » apparaît une fois dans le premier fichier et cinq fois dans le deuxième, l'affichage serait :

1 --) 5 --) toto boit

De plus l'utilisateur peut entrer différentes options dans la saisie par ligne de commande, ces options permettent de faire des modifications du programme (sur l'affichage, la sélection des lignes etc…) mais nous reviendrons plus en détail sur cela dans la partie II.

2)Problématiques

Tout d'abord la question de quelle structure de données choisir pour stocker chaque ligne du premier fichier s'est posée. Deux choix étaient possibles, prendre la structure des tables de hachage ou celle des arbres binaires. Ensuite la problématique de la lecture et du stockage des lignes de caractères d'un fichier s'est posée. Enfin la saisie devait être minutieusement traité du fait que plusieurs options pouvaient être ajoutés avant le nom des fichiers.

II-Présentation de notre projet

1)Structures de données

Plusieurs structures de données ont été utilisées dans notre programme :

1-Le fourretout :

```
typedef struct choldall choldall;

struct choldall {
  void *ref;
  choldall *next;
};

struct holdall {
  choldall *head;
#if defined HOLDALL_PUT_TAIL && HOLDALL_PUT_TAIL != 0
  choldall **tailptr;
#endif
  size_t count;
};
```

Ces structures ont été prises du module fourretout qui a été développé plus tôt cette année en séance de travaux pratiques. Le fourretout est

essentiellement une liste dynamique simplement chaîné, « struct *holdall* » gère la tête de cette liste et permet de stocker le nombre d'éléments de cette liste. « struct *choldall* » permet de gérer les cellules de cette liste qui contiennent une référence. Le fourretout va nous permettre de stocker des données qui n'ont pas besoin d'être triées, il est parfois plus judicieux de choisir un fourretout plutôt qu'un tableau, cela sera expliqué pour la prochaine structure.

2-L'arbre binaire de recherche équilibré :

```
typedef struct cbst cbst;

struct bst {
  int (*compar)(const void *, const void *);
  cbst *root;
};

struct cbst {
  cbst *next[2];
  const void *ref;
  int height;
  union {
    holdall *line_table;
    int *occurence_table;
  };
};
```

Ces structures ont été prises du module « *bst* » qui a été développé plus tôt cette année en séance de travaux pratiques.

L'arbre binaire de recherche équilibré est un arbre binaire permettant de faire des recherches avec une complexité *O(ln n)*, « struct *bst* » gère la tête de cet arbre binaire et sauvegarde la fonction de comparaison qui va être utilisé pour comparer les références des cellules. « struct *cbst* » permet de gérer les cellules de cet arbre binaire, « *ref* » contiendra les lignes de texte.

L'union va permettre de créer une unique structure pour les deux cas différents de saisie (au lieu de deux structures).

Le fourretout « *line_table* » sera utilisé lorsqu'un seul fichier sera saisi par l'utilisateur. Ce fourretout sauvegardera les lignes où apparait « *ref* ».

Dans le cas où plus d'un fichier est saisi ce sera le tableau d'entiers « *occurrence_table* » qui sera utilisé, ce tableau sera de taille le nombre de

fichiers saisis par l'utilisateur, chaque case du tableau correspondra au nombre d'occurrence de « *ref* » dans un fichier, la première case du tableau correspond au premier fichier, la deuxième case au deuxième fichier etc.

Pourquoi avoir choisi un fourretout et un tableau d'entiers ?

Tout simplement car le nombre de lignes où « ref » apparaît dans le fichier ne peut pas être connu à l'avance, ainsi l'espace mémoire sera dynamique et un fourretout convient à cet usage.

En opposition le tableau d'entiers est de taille le nombre de fichiers entrés par l'utilisateur, ce nombre est donc connu avant l'initialisation de la structure, ainsi un tableau convient pour un espace mémoire statique.

3- La saisie des fichiers et des options :

```
struct options{
  bool filter;
  int (*is_class)(int);
  bool sort;
  int (*compar)(const void *, const void *);
  bool uppercase;
  holdall *file_names;
};
```

Cette structure a été créée pour ce projet, elle permet de gérer les options que l'utilisateur peut saisir, elle permet aussi de sauvegarder dans un fourretout les noms de fichiers que l'utilisateur saisit. Ainsi, lorsque l'utilisateur saisit des options et des noms de fichiers, la structure sera modifiée en conséquence. Par la suite cette structure permettra de gérer les différentes options lors de la lecture des fichiers et l'affichage.

Le booléen « *filter* » est à « *true* » lorsque l'option filtre a été saisie « *false* » sinon, « *is_class* » contient la fonction qui a été saisie avec cette option.

Le booléen « *sort* » est à « *true* » lorsque l'option sort a été saisi « *false* » sinon, « *compar* » contient la fonction qui a été saisie (implicitement) avec cette option.

Le booléen « *uppercase* » est à « *true* » lorsque l'option « *uppercase* » a été saisie.

Le fourretout « *file_names* » contient tous les noms de fichiers saisis par l'utilisateur.

2) La saisie de l'utilisateur

Pour la saisie de l'utilisateur, le module options contient toutes les fonctions et macro constantes nécessaires pour gérer cette saisie, voici les fonctions qui vont permettre de vérifier cette saisie.

```
//------------Getter-------------//
// opt_filter : renvoie le booléan filter
extern bool opt_filter(options *opt);

// opt_sort : renvoie le booléan sort
extern bool opt_sort(options *opt);

// opt_compar_fun : renvoie un pointeur vers la fonction compar
extern int (*opt_compar_fun(options *opt))(const void *, const void *);

// opt_uppercase : renvoie le booléan uppercase
extern bool opt_uppercase(options *opt);

// opt_file_names : renvoie un pointeur vers la fonction file_names
extern holdall *opt_file_names(options *opt);

// test_short_filter : vérifie si s est la même chaîne de caractères que la concaténation de OPT__SHORT et FILTER_SHORT,
//      si c'est le cas, la fonction renvoie true, sinon elle renvoie false
extern bool test_short_filter(char *s);

// test_short_sort : vérifie si s est la même chaîne de caractères que la concaténation de OPT__SHORT et SORT_SHORT,
//      si c'est le cas, la fonction renvoie true, sinon elle renvoie false
extern bool test_short_sort(char *s);

// help : Affiche le fonctionnement du programme et son utilisation sur la sortie standard
extern void help();
//-------------------------------//
```

Ici toutes les fonctions permettent d'obtenir des informations sur la structure options, hormis la fonction « *help* » qui affiche le fonctionnement du programme et son utilisation. Ces fonctions nous seront utiles dans le programme principal main.

```
// opt_compar : renvoie le résultat de la fonction de comparaison compar qui a pris
//      en argument a et b, cela renvoie 1 si a > b, 0 si a == b et -1 si a < b
extern int opt_compar(options *opt, const void *a, const void *b);

// opt_class : renvoie le résultat de la fonction is_class qui a pris c en argument,
//      filter doit être vrai pour utiliser cette fonction, le comportement est sinon
//      indéterminé.
extern int opt_class(options *opt, int c);

// test_short_uppercase : vérifie si s est la même chaîne de caractères que la concaténation de OPT__SHORT et UPPERCASE_SHORT,
//      si c'est le cas, uppercase passe à vrai et la fonction renvoie true, sinon elle renvoie false
extern bool test_short_uppercase(char *s, options *opt);

// test_short_opt_filter : compare la chaîne de caractères s à toutes les fonctions de <ctype.h> (isalpha, isdigit ...), si un cas
//      match, alors filter passe à true et is_class prend cette fonction en paramètre, puis la fonction renvoie 1. Renvoie 0 si
//      aucun cas ne match.
extern int test_short_opt_filter(char *s, options *opt);

// test_short_opt_sort : compare la chaîne de caractères s aux deux options "locale" et "standard"
//      s'il y a un match avec standard sort passe à true et la fonction renvoie 1, s'il y a un match avec
//      locale sort passe à true, la fonction setlocate() est appelé et compar pointe vers strcoll puis renvoie 1.
//      Renvoie 0 si aucun cas ne match.
extern int test_short_opt_sort(char *s, options *opt);
// test_long_opt : Test si la chaîne de caractères pointé par s fait partie de la liste des filtres long
// renvoie 1 si c'est le cas, -1 en cas d'erreur système, 0 si s n'est pas un filtre long
```

Les fonctions « *opt_compar* » et « *opt_class* » permettent d'utiliser les fonctions « *compar* » et « *is_class* » de la structure options. La fonction « *opt_class* » n'est pas toujours définie, « *opt_filter* » doit renvoyer vrai pour pouvoir utiliser la fonction « *opt_class* ».

Les fonctions « *test_short_uppercase* », « *test_short_opt_filter* » et « *test_short_opt_sort* » permettent de vérifier la bonne saisie d'une option courte, elles mettent aussi à jour la structure dans le cas où la saisie est correcte.

```
// test_long_opt : compare la chaîne de caractères s à la concaténation de OPT__LONG, FILTER_LONG et :
//      -Chaque fonction de <ctype.h> (isalpha, isdigit ...) ou
//      -Aux chaînes "locale" ou "standard" ou
//      -UPPERCASE_LONG
//      S'il y a un match avec les fonctions ctype alors filter passe à true, is_class pointe vers la fonction
//      ctype et la fonction renvoie 1.
//      S'il y a un match avec "standard" sort passe à true et la fonction renvoie 1, s'il y a un match avec "locale"
//      sort passe à true, la fonction setlocate() est appelé et compar pointe vers strcoll puis renvoie 1.
//      S'il y a un match avec UPPERCASE_LONG uppercase passe à true et renvoie 1.
//      Renvoie -1 si une erreur a lieu, 0 si aucun match n'est fait.
extern int test_long_opt(char *s, options *opt);

// opt_initialize : Alloue et initialise une structure de type options, alloue aussi le fourretout associé, filter, sort et uppercase sont
//      mis à false, compar est initialisé à strcmp.
extern options *opt_initialize();

// dispose_opt : Libère l'espace alloué au fourretout et la structure.
extern void dispose_opt(options *opt);
```

La fonction *« test_long_opt »* regroupe tous les tests pour les options longues. Elle permet pareillement de tester puis mettre à jour si la saisie de l'option longue est correcte.

Les fonctions *« opt_initialise »* et *« dispose_opt »* permettent respectivement d'initialiser la structure options et de la désallouer.

Les fonctions tests doivent être utilisées dans le fichier « main » selon certaines conditions, l'image ci-dessous le montre :

```
int c = 0;
for(size_t i = 1; i < (size_t)argc; i++){
  if(strstr(argv[i], OPT__LONG) != NULL){
    if( (c = test_long_opt(argv[i], opt)) == 1){
    }else if(c == -1){
      goto error_sprintf;
    }else if(c == 0){
      dispose_opt(opt);
      goto error_use;
    }
  }
  else if(strstr(argv[i], OPT__SHORT) != NULL){
    if(test_short_uppercase(argv[i], opt)){
    }
    else if(i + 1 < (size_t)argc){
      if(test_short_filter(argv[i])){
        if( (c = test_short_opt_filter(argv[i + 1], opt)) == 1){}
          else if(c == 0){
            dispose_opt(opt);
            goto error_use;
          }
      }else if(test_short_sort(argv[i])){
          if( (c = test_short_opt_sort(argv[i + 1], opt)) == 1){}
          else if(c == 0){
            dispose_opt(opt);
            goto error_use;
          }
      }else{
        dispose_opt(opt);
        goto error_use;
      }
    }
  }else if(strstr(argv[i], IS_FILE) != NULL){
      for(size_t j = i; j < (size_t)argc; j++){
        if(holdall_add(opt_file_names(opt), argv[j]) != 0){
          goto error_alloc_aux;
        }
      }
      break;
  }
}
```

Ici sont appelés les fonctions tests sous des conditions précises, trois blocs principaux sont à distinguer, celui qui vérifie si l'option est longue, (on teste le *« -- »)*, celui qui vérifie si l'option est courte *(« - »)* et celui qui vérifie si l'utilisateur à entrer un fichier. Si l'option est courte on effectue une série de test, si aucune option courte n'est trouvée alors que l'utilisateur a mis le « - » alors le programme s'arrête en précisant que l'option n'est pas reconnue, la même idée est appliquée dans le deuxième bloc avec l'option longue.

Ensuite pour le troisième bloc, lorsque l'utilisateur a saisi un fichier, alors il doit nécessairement contenir un « . », et la suite de la saisie doit nécessairement être des fichiers. Si l'utilisateur ne respecte pas cette convention, alors le programme s'arrêtera lors de l'ouverture de ces noms de fichiers et indiquera qu'un fichier ou une option n'a pas été reconnu.

3)Le traitement des fichiers

```
bool filtered;
while ((c = fgetc(src)) != '\n' && c != EOF) {
  filtered = true;
  if(opt_filter(opt) && !opt_class(opt, c)){
    filtered = false;
  }
  if(count + 1 >= TAB_SIZE && filtered) {
    TAB_SIZE *= 2;
    char *s = realloc(t, TAB_SIZE * (sizeof *s));
    if (s == NULL) {
      r = EXIT_FAILURE;
      if(i >= 1){
        goto dispose_multiple_file;
      }
      goto dispose_one_file;

    }
    t = s;
  }

  if(filtered){
    if(opt_uppercase(opt)){
      t[count] = (char)toupper(c);
    }else{
      t[count] = (char) c;
    }
    ++count;
  }
}
t[count] = '\0';
```

Premièrement nous lisons caractère par caractère le fichier qui a été ouvert, on applique directement le filtre dans le cas où un filtre a été saisie par l'utilisateur.

On sauvegarde ces caractères dans un tableau « *t* » qui va ensuite être ajouté à l'arbre binaire de recherche équilibré grâce à nos deux fonctions qui ont été développés ci-dessous :

```
// bst_add_endofpath_one_file : renvoie -1 si ref vaut NULL. Recherche sinon dans
//    l'arbre binaire de recherche associé à t la référence d'un objet égal à
//    celle de référence ref au sens de la fonction de comparaison. Si la
//    recherche est positive, on ajoute line dans le fourretout line_table associé à la cellule de ref
//    et on renvoie 0. Tente sinon d'ajouter la référence selon la méthode de l'ajout en bout de chemin si le mot n'est pas blanc;
//    renvoie -1 en cas de dépassement de capacité 0 si ref est blanc, 1 sinon
extern int bst_add_endofpath_one_file(bst *t, const void *ref, size_t *line);

// bst_add_endofpath_multiple_file : renvoie -1 si ref vaut NULL. Recherche sinon dans
//    l'arbre binaire de recherche associé à t la référence d'un objet égal à
//    celle de référence ref au sens de la fonction de comparaison. Si la
//    recherche est positive, on incrémente occurence_table à l'indice index et on renvoie 0.
//    Tente sinon d'ajouter la référence selon la méthode de l'ajout en bout de chemin si le mot n'est pas blanc et index != 0;
//    renvoie -1 en cas de dépassement de capacité 0 si ref est blanc ou que indice == 0, 1 sinon.
extern int bst_add_endofpath_multiple_file(bst *t, const void *ref,
    size_t index,
    size_t number_of_files);
```

La fonction « *Bst_add_endofpath_one_file* » est utilisée lorsqu'un seul fichier a été rentré par l'utilisateur, « line » est systématiquement ajouté au fourretout de la cellule contenant « *ref* », si « *ref* » n'existe pas dans l'arbre binaire associé à « t », la cellule sera créée.

La fonction « *Bst_add_endofpath_multiple_file* » est utilisée lorsque plusieurs fichiers ont été saisies par l'utilisateur, si « *ref* » est dans l'arbre binaire associé à « t » alors on incrémente de 1 l'occurrence à l'indice « index » dans le tableau d'entiers *« occurrence_table »*, si *« ref »* n'est pas dans l'arbre alors il est ajouté et l'incrémentation s'effectue aussi.

```
  int m;
  if (holdall_count(opt_file_names(opt)) == 1) {
    if((m = bst_add_endofpath_one_file(b, t, compteur)) == -1){
      free(t);
      fprintf(stderr, "*** Error: Not enough memory\n");
      goto dispose_one_file;
    }else if(m == 0){
      free(t);
    }
  } else {
    if((m = bst_add_endofpath_multiple_file(b, t, i, holdall_count(opt_file_names(opt)))) == -1){
      free(t);
      free(compteur);
      fprintf(stderr, "*** Error: Not enough memory\n");
      goto dispose_multiple_file;
    }else if(m == 0){
      free(t);
    }
    free(compteur);
  }
  ++k;
}
```

Il y a deux cas, celui où un seul fichier a été saisi et que donc le fourretout des noms de fichiers à un seul élément, et celui où plusieurs fichiers ont été saisis donc le cas du *« else ».*

Dans le premier cas on appellera « *bst_add_endofpath_one_file* », si « *ref* » est déjà dans l'arbre binaire alors on désalloue la ligne lue car elle ne peut pas être ajouté plus d'une fois.

Dans le deuxième cas on appellera «*bst_add_endofpath_multiple_file*», si « *ref* » est déjà dans l'arbre binaire, alors on désalloue la ligne lue pour les mêmes raisons.

```
//  bst_repr_graphic_one_file : parcours l'arbre binaire associé à t pour afficher
//      les lignes de line_table séparés par une virgule suivi d'une tabulation suivi de ref
//      pour chaque cellule de l'arbre binaire, cela uniquement dans le cas où line_table contient au moins deux lignes.
extern void bst_display_one_file(bst *t);

//  bst_repr_graphic_multiple_file : parcours l'arbre binaire associé à t pour afficher
//      les occurences du tableau occurence_table séparés par une tabulation puis suivi d'une tabulation suivi de ref
//      pour chaque cellule de l'arbre binaire, cela uniquement dans le cas où occurence_table ne contient pas d'occurences à 0.
extern void bst_display_multiple_file(bst *t, size_t number_of_files);
```

La fonction « *Bst_display_one_file* » est utilisée dans le cas où un seul fichier a été saisi (donc que le fourretout a été alloué dans la structure), elle affiche les lignes ainsi que la référence de chaque cellule de l'arbre binaire associé à *« t »* dans le cas où le fourretout contient au moins deux lignes. L'affichage se fait selon le format demandé dans le projet.

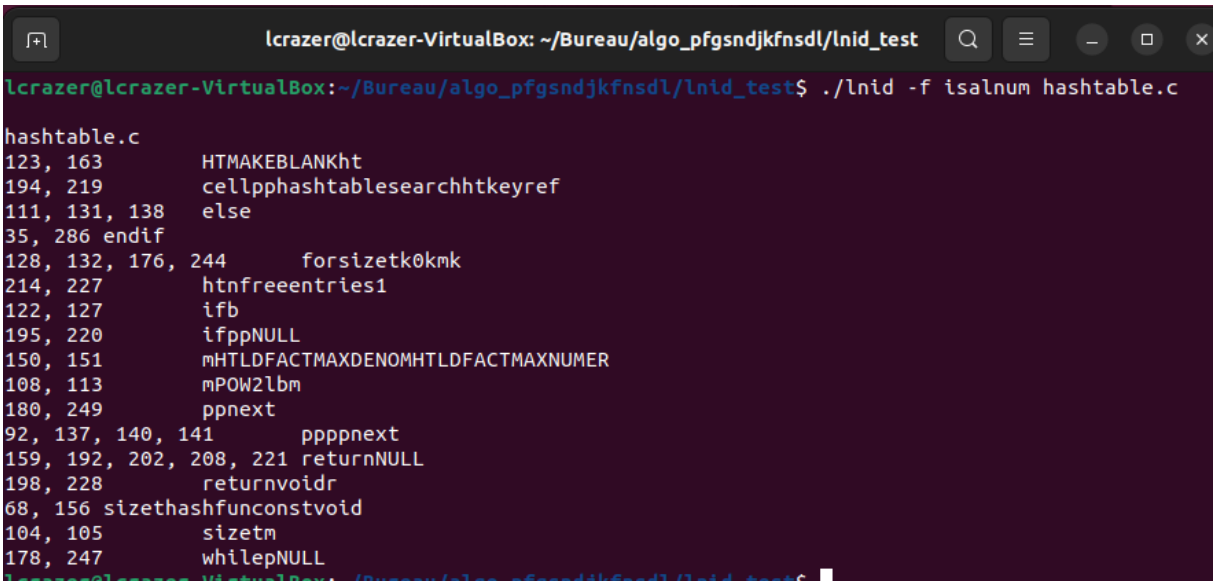
La fonction *« Bst_display_multiple_file »* est utilisée dans le cas où plusieurs fichiers ont été saisis (donc que le tableau d'entiers a été alloué dans la structure), elle affiche *« occurrence_table »* selon le format demandé dans le cas où toutes ses cases sont différentes de zéro, la référence est affichée à la suite.

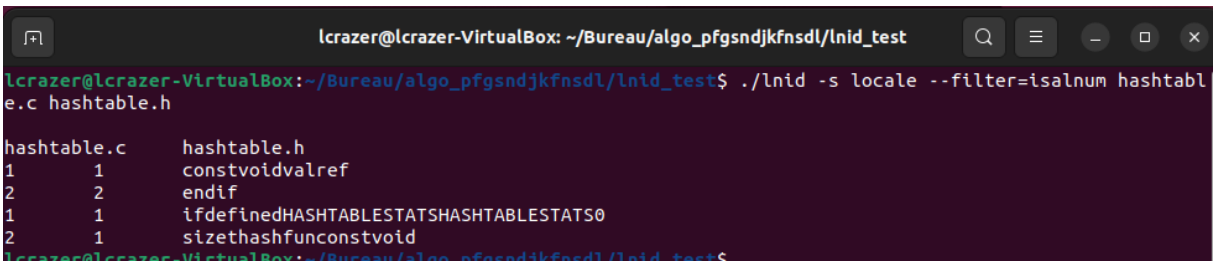
Le parcours de l'arbre est infixe pour ces deux fonctions.

III-Evaluation du programme

1)Test

Des tests ont été présentés dans le sujet, voici ces tests fait par notre projet :





Les résultats retournés par les tests sont satisfaisants car ils sont identiques à ceux dans le sujet.

2)Optimisations

Lors de la lecture d'au moins deux fichiers identiques parmi les fichiers lus sur la ligne de commande, le programme va parcourir ces fichiers identiques comme s'ils étaient différents or une seule lecture est suffisante.

En effet une seule lecture de ces fichiers identiques est nécessaire. Si un autre fichier est identique au premier fichier lu, le programme appellera la fonction *« bst_add_endofpath_same_file »* qui fera une simple copie des occurrences du premier fichier pour les références correspondantes.

L’explication de cette fonction sera faite en premier lieu, puis l’explication de son utilisation dans le fichier main.

```
// bst_add_endofpath_same_file : parcours l'arbre binaire de recherche associé à t
//    en copiant l'occurence de occurence_table à l'indice index_similar dans occurence_table
//    à l'indice index pour chaque cellule de l'arbre binaire.
extern void bst_add_endofpath_same_file(bst *t, size_t index, size_t index_similar);
```

Cette fonction permet pour chaque cellule de l’arbre binaire associé à « t » de copier l’occurrence du tableau *« occurrence_table »* à l’indice *« index_similaire »* dans la case située à l’indice *« index »* du même tableau.

```
90      }
91      bool same_file = false;
92      for(size_t n = 0; n < i; n++){
93        if (strcmp(s ,(char *)holdall_search_index(opt_file_names(opt),n)) == 0){
94          bst_add_endofpath_same_file(b, i, n);
95          same_file = true;
96        }
97      }
98
99      if(!same_file){
100       FILE *src = fopen(s, "r");
```

Voici son utilisation dans le fichier main, à chaque fichier traité par la boucle principale, la boucle « for » à la ligne 92 permet de vérifier si ce fichier a déjà été lu, si c’est le cas alors la fonction *« bst_add_endofpath_same_file »* est appelée, et la variable *« same_file »* est mise à *« true »,* ce qui signifie que le fichier ne sera pas lu et que le programme passera au prochain fichier. On en conclu qu’il y aura un gain de temps grâce à cette vérification et un gain sur les allocations mémoires.

Voici une ligne de commande qui va permettre d’effectuer un test de ce gain :

```
lcrazer@lcrazer-VirtualBox:~/Bureau/clercleo_projet/lnid_test$ time valgrind ./l
nid lesmiserables.txt lesmiserables.txt lesmiserables.txt lesmiserables.txt
```

Voici deux captures d'écrans, la première sans la vérification et la deuxième avec l'ajout de cette vérification :

1ère capture :

```
==6003==
==6003== HEAP SUMMARY:
==6003==     in use at exit: 0 bytes in 0 blocks
==6003==   total heap usage: 1,727,050 allocs, 1,727,050 frees, 47,016,568 bytes
 allocated
==6003==
==6003== All heap blocks were freed -- no leaks are possible
==6003==
==6003== For lists of detected and suppressed errors, rerun with: -s
==6003== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

real    0m7,065s
user    0m5,795s
sys     0m1,080s
```

2ème capture :

```
==6011== HEAP SUMMARY:
==6011==     in use at exit: 0 bytes in 0 blocks
==6011==   total heap usage: 506,605 allocs, 506,605 frees, 13,850,440 bytes all
ocated
==6011==
==6011== All heap blocks were freed -- no leaks are possible
==6011==
==6011== For lists of detected and suppressed errors, rerun with: -s
==6011== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

real    0m3,502s
user    0m2,186s
sys     0m1,157s
```

Dans le 1er cas on effectue plus de 1,7 million allocations mémoires, alors que dans le 2ème cas seulement 500 000 allocations mémoires sont effectuées, on a donc 3 fois moins d'allocation dans le 2ème cas.

En effet dans le 1er cas les allocations s'additionnent à chaque fichier, car chaque fichier est lu une nouvelle fois ce qui implique de réallouer toutes les lignes du fichier. Ce qui n'est pas le cas dans notre deuxième capture, car une fois que le fichier *« lesmiserables.txt »* est lu une fois, il ne sera pas parcouru à nouveau.

Au niveau du temps on remarque que le temps d'exécution du programme est divisé par deux, cela s'explique par le fait qu'encore une fois, les autres fichiers ne sont pas parcourus, et donc que toutes les opérations liées au parcours de ces fichiers n'ont pas lieux.

IV-Conclusion

1)Synthèse

Le programme permet à l'utilisateur d'entrer toutes les options qui ont été demandés dans le sujet, suivi de fichiers (un seul ou plusieurs).

Le traitement des fichiers ainsi que les options produisent un affichage cohérent par rapport aux exemples donnés dans le sujet.

2) Difficultés rencontrées et améliorations possibles

La désallocation et le traitement des erreurs étaient une difficulté qu'on a rencontrée, en effet il ne fallait pas s'emmêler les pinceaux dans les « goto » et dans les désallocations étant donné que chaque erreur dans le programme amenait souvent à un traitement spécifique.

Le programme peut être amélioré sur différents aspects, premièrement le développement des fonctions qui testent la saisie, en effet cela est visible avec la fonction *« test_long_opt* » où une série de test avec des « if » pourraient être réduit à une boucle dans le cas où les noms de fonctions de la librairie *« ctype »* seraient sauvegardés.