DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 9

I. Utilisation de gprof sur le projet

A. Compilation

Nous avons commencé par ajouter une règle **gprof** au Makefile principal du projet. Lorsqu'elle est invoquée, celle-ci se charge de nettoyer le répertoire, puis de recompiler le projet avec les variables **GPROF=1** et **STATIC=1.** La première ajoute aux flags de compilation et d'édition de liens l'option d'instrumentation **-pg** dédiée à **gprof**, tandis que la seconde assure la génération de librairies statiques plutôt que partagées, étant donné que **gprof** ne fonctionne pas avec des librairies partagées. Ensuite, la règle exécute le programme **dico** sur un texte de grande taille (en l'occurrence, 16 Mo) inclus dans le répertoire **exemples** : **longtext.txt**. Enfin, elle exécute **gprof** avec en argument l'exécutable afin d'analyser le fichier **gmon.out** généré par la commande précédente.

Il est possible d'ajouter à l'appel de cette règle la déclaration de variable **GRAPH=1** afin de générer le graphe des appels au format **.png**. Pour cela, nous avons récupéré le code Python permettant de convertir les données de **gprof** au format image : il s'agit du fichier **gprof2dot.py** situé dans un nouveau répertoire du projet, **profiler**. C'est dans ce même répertoire que sera généré le fichier **.png** correspondant au graphe d'appels. Ainsi, on peut effectuer une analyse de profiling sur notre projet de deux manière différentes :

- **make gprof** pour obtenir les résultats au format textuel, dans le terminal ;
- **make gprof GRAPH=1** pour obtenir les résultats au format textuel ainsi qu'au format image.

B. Analyse des résultats

Lorsque l'on lance gprof pour obtenir une analyse textuelle des relevés faits pendant l'exécution de notre programme, on obtient sur la sortie standard un résumé en deux parties. D'abord l'analyse *flat* représentée sous forme d'un tableau à deux entrées :

- une ligne par fonction exécutée pendant les mesures, triées selon la première colonne, le pourcentage de temps total d'exécution ;
- une colonne par type de relevé, notamment le temps pris, le nombre d'appels et le temps par appel.

Parmi ces colonnes deux catégories retiennent notre attention. ***% time*** et ***Self seconds*** nous permettent de trouver la fonction dans laquelle notre programme passe le plus de temps, qui est donc le goulot d'étranglement de nos performances. La deuxième est ***calls***, qui nous permet de trouver la fonction la plus appelée de notre programme, sur laquelle la moindre optimisation donnera de grands résultats.

C'est ainsi qu'on porte notre attention sur deux fonctions qui sortent du lot :

- **displayWord** : fonction ayant le plus de temps cumulé d'exécution. C'est un résultat attendu car cette fonction s'occupe de l'affichage sur la sortie standard, qui est notoirement lente.
- **compareWord** : fonction la plus appelée du programme. Elle se charge de comparer les lettres de chaque mot une par une pour ordonner alphabétiquement les mots et tester leur égalité.

displayWord ne présente pas beaucoup d'opportunités d'optimisation, à part éventuellement de bufferiser les écritures pour éviter les appels répétés au système.

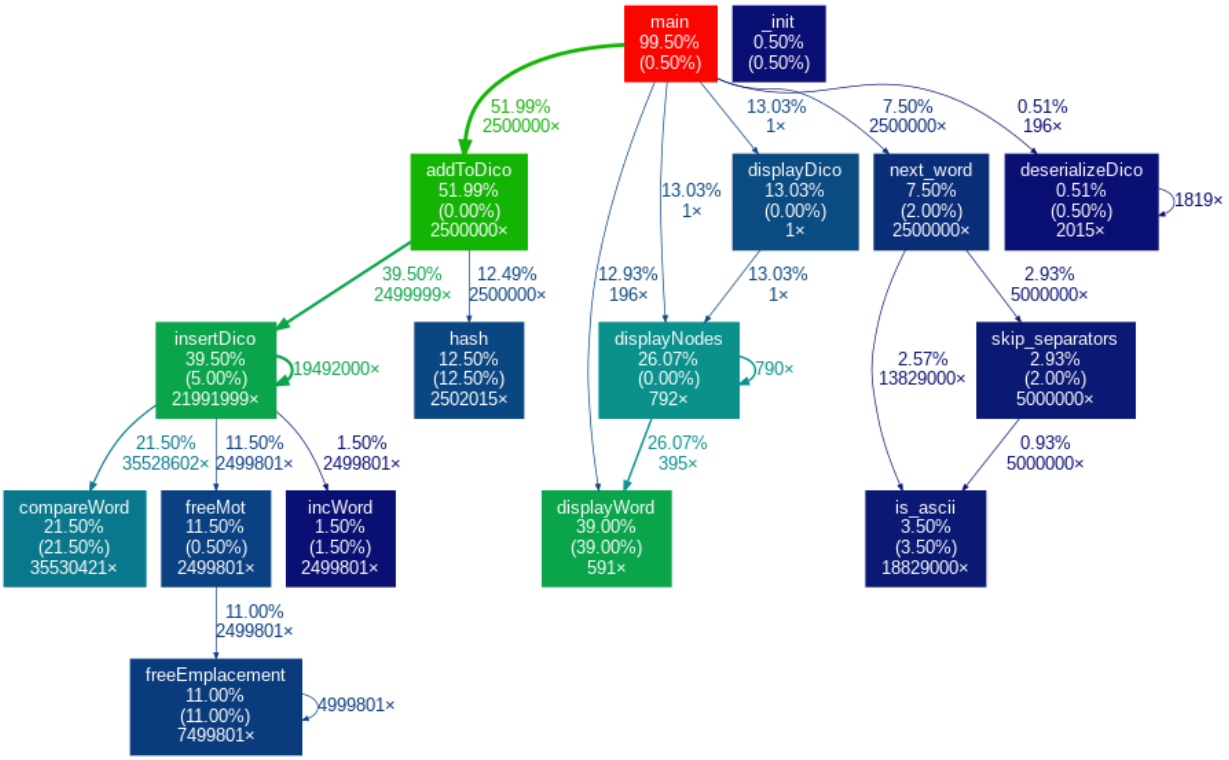
compareWord peut être légèrement améliorée en évitant de la redondance lors du calcul de la taille des mots. Il y a 8 appels à la fonction **strlen**, qui sont pourtant toujours sur les mêmes mots. En assignant les premiers résultats à des variables, on tombera à 2 appels à strlen. On peut également réduire le nombre de ses appels de la même manière dans **insertDico** qui l'appelle deux fois de suite avec les mêmes arguments. Sur notre exemple, cela nous permet de réduire de **40**% son nombre d'appels.

Nous avons plus tôt dans le projet également optimisé la fonction **incWord**, qui parcourait à chaque appel toute la liste chaînée des mots pour

atteindre la fin, alors que la queue de la listé est connue ; nous avons supprimé ce parcours inutile qui sur notre fichier de test nous évite **2499801** parcours. En supprimant l'optimisation, le temps d'exécution est quasiment dédoublé, et la fonction la plus chronophage devient **incWord**.

La deuxième partie est le ***Call graph***. Elle nous permet de savoir, pour chaque fonction, quelle est son appelante et quelles fonctions elle appelle. Cela permet de dresser une rapide architecture du programme. Toutefois l'affichage textuel peut être assez long dans le cas de grands projets. La lecture peut en être facilitée grâce à un script python pour le transformer en fichier **dot** puis en image.

L'utilisation du script **gprof2dot** sur notre exemple nous permet d'obtenir le graphe suivant :



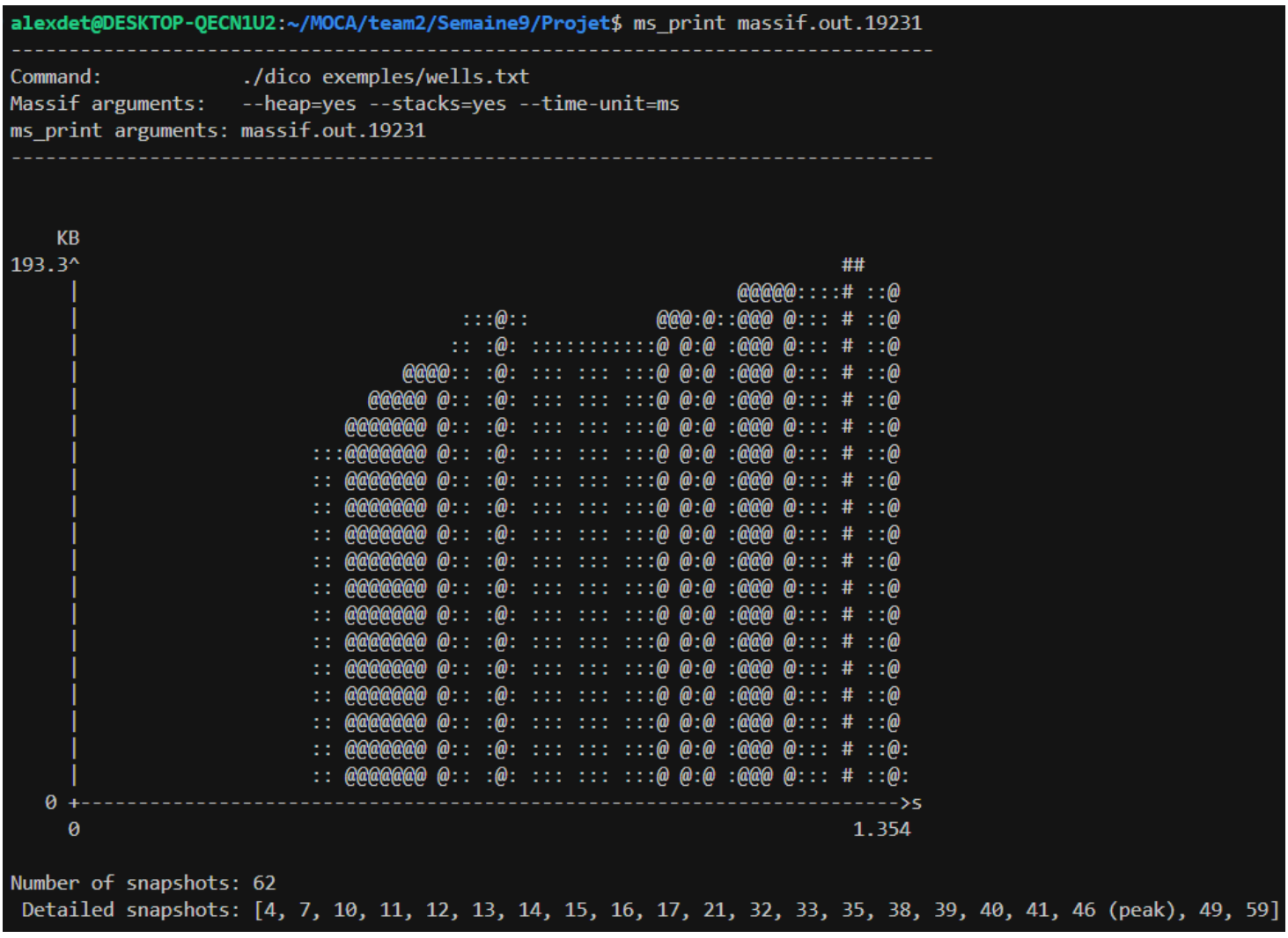
Il permet rapidement de comprendre la structure et les appels du programme, ainsi que d'évaluer le temps d'exécution rapidement grâce aux couleurs des cases. Plus la couleur est foncée et tend vers le rouge plus le temps d'exécution de la fonction + ses appels internes est grand. On y repère moins facilement la fonction la plus chronophage car pour ne pas prendre en compte les appels internes il faut regarder le pourcentage donné entre parenthèses (colonne *self*).

II. Utilisation de Valgrind Massif sur le projet

En compilant notre projet de manière ordinaire, puis en utilisant valgrind avec l'outil massif sur l'exécutable, avec pour entrée un fichier d'environ 2.5 Ko :

```
valgrind --tool=massif --heap=yes --stacks=yes --time-unit=ms
./dico exemples/wells.txt
```

Nous avons obtenu le graphique suivant :



L'outil nous indique avoir enregistré 62 captures du programme, à des moments différents de l'exécution. La durée totale d'exécution est de 1.3s, et l'on atteint un pic de 193 Ko de mémoire utilisée pendant l'exécution. Pour

chacune de ces 62 captures, un détail de l'utilisation mémoire de notre programme est fourni, avec les attributs suivants :

- **n** le numéro de la capture ;
- **time** le temps estimé d'exécution du programme au moment de la capture, en millisecondes ;
- **total** l'espace mémoire total utilisé (en octets), incluant à la fois la pile et le tas (puisqu'on a demandé un profiling des deux à la fois ici) ;
- **useful-heap** l'espace mémoire principal alloué au programme dans le tas (en octets) ;
- **extra-heap** l'espace mémoire additionnel alloué au programme dans le tas (en octets) ;
- **stacks** l'espace mémoire alloué au programme dans la pile (en octets).

Voici un exemple des résultats obtenus pour les 10 premières captures (dans l'ordre temporel) :

```
--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0              0            0
  1             15              624                0              0          624
  2             31            1,216                0              0        1,216
  3             57            4,168                0              0        4,168
  4            103            4,928                0              0        4,928
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  5            106            4,976                0              0        4,976
  6            335            5,920                0              0        5,920
  7            362            1,376                0              0        1,376
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  8            391          136,312          135,648             96          568
  9            413          137,936          136,824            584          528
 10            437          143,888          140,620          2,212        1,056
97.73% (140,620B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->91.08% (131,048B) 0x109871: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|
->02.85% (4,096B) 0x49D1BA3: _IO_file_doallocate (filedoalloc.c:101)
| ->02.85% (4,096B) 0x49E0CDF: _IO_doallocbuf (genops.c:347)
|   ->02.85% (4,096B) 0x49DFCDB: _IO_file_underflow@@GLIBC_2.2.5 (fileops.c:485)
|     ->02.85% (4,096B) 0x49E0D95: _IO_default_uflow (genops.c:362)
|       ->02.85% (4,096B) 0x10A5CF: next_word (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|         ->02.85% (4,096B) 0x109940: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|
->01.94% (2,788B) in 8 places, all below massif's threshold (1.00%)
|
->01.87% (2,688B) 0x109F59: addToDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
  ->01.87% (2,688B) 0x10995C: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
```

Nous constatons que Massif initialise toutes les valeurs à 0 sur la première ligne (n=0), puis commence à traquer l'utilisation mémoire du programme. Pour les 7 premières captures (de n=1 à n=7), on observe que le tas n'est toujours pas exploité (colonnes **heap** à 0), et seule la pile l'est progressivement, par des appels de fonctions et/ou stockage de variables locales ainsi que de paramètres effectifs. En revanche, à partir de la 8ème capture, l'utilisation du tas augmente brusquement : cela correspond principalement aux allocations mémoire (**malloc**) effectuées dans le **main** (**dico.c**). En effet, Massif indique que sur les **140 620 octets** alloués dans le tas, **91%** proviennent du main.

On peut retrouver les **131 048** octets alloués par le main facilement. L'instruction suivante alloue un tableau d'adresses de taille 8 octets :

```
mot_data_t **serialized_dico = (mot_data_t **) calloc(MaxSizeArray,
sizeof(mot_data_t *));
```

Or ce tableau est de taille **MaxSizeArray**, une macro définie sur **16381** dans le code. On obtient donc bien les **16381 * 8 = 131 048** octets indiqués. Cette allocation est de loin la plus lourde du programme, en raison de la taille considérable du tableau.

Les autres allocations mémoire du programme, dispersées entre **dico_tools.c** (**insertDico, addToDico**), **word_tools.c** (**nextWord**, **incWord**) et **serialization.c** (**deserializeDico**) sont clairement négligeables en comparaison, puisqu'on y alloue des structures de quelques dizaines d'octets tout au plus (des entiers et quelques pointeurs). Néanmoins, ces dernières sont bien plus nombreuses, puisque répétées à chaque ajout d'un mot dans le dictionnaire ou presque. De cette manière, le monopole du **main** quant à l'utilisation globale du tas va peu à peu se répartir dans les captures suivantes.

Voici les résultats obtenus pour les captures associées à l'utilisation maximale de la mémoire (tas + pile) par le programme. Le **main** ne représente alors plus que **66%** de l'espace mémoire alloué dans le tas, et les autres allocations cumulées représentent environ le tiers restant de l'espace alloué.

```
--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 42         1,170          196,912          178,152       17,320        1,440
 43         1,194          195,888          177,856       17,272          760
 44         1,218          195,944          177,856       17,272          816
 45         1,242          197,544          177,856       17,272        2,416
 46         1,258          197,984          177,856       17,272        2,856
89.83% (177,856B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->66.19% (131,048B) 0x109871: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|
->06.05% (11,984B) 0x109F59: addToDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
| ->06.05% (11,984B) 0x10995C: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|
->05.97% (11,816B) 0x4857378: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| ->03.20% (6,328B) 0x485746C: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | ->01.75% (3,472B) 0x485746C: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | | ->01.75% (3,472B) in 3 places, all below massif's threshold (1.00%)
| | |
| | ->01.41% (2,800B) 0x4857450: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | | ->01.41% (2,800B) in 3 places, all below massif's threshold (1.00%)
| | |
| | ->00.03% (56B) in 1+ places, all below ms_print's threshold (01.00%)
| |
| ->02.74% (5,432B) 0x4857450: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | ->01.50% (2,968B) 0x485746C: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | | ->01.50% (2,968B) in 3 places, all below massif's threshold (1.00%)
| | |
| | ->01.22% (2,408B) 0x4857450: deserializeDico (in /home/alexdet/MOCA/team2/Semaine9/Projet/libs/libs/libserialization.so)
| | | ->01.22% (2,408B) in 3 places, all below massif's threshold (1.00%)
| | |
| | ->00.03% (56B) in 1+ places, all below ms_print's threshold (01.00%)
| |
| ->00.03% (56B) in 1+ places, all below ms_print's threshold (01.00%)
|
->02.59% (5,120B) 0x49D1BA3: _IO_file_doallocate (filedoalloc.c:101)
| ->02.59% (5,120B) 0x49E0CDF: _IO_doallocbuf (genops.c:347)
|   ->02.07% (4,096B) 0x49DFCDB: _IO_file_underflow@@GLIBC_2.2.5 (fileops.c:485)
|   | ->02.07% (4,096B) 0x49E0D95: _IO_default_uflow (genops.c:362)
|   |   ->02.07% (4,096B) 0x10A5CF: next_word (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|   |     ->02.07% (4,096B) 0x109940: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|   |
|   ->00.52% (1,024B) in 1+ places, all below ms_print's threshold (01.00%)
```

Enfin, ci-dessous se trouvent les résultats obtenus pour les dernières captures. Au fur et à mesure des libérations (**free**), l'utilisation du tas diminue, ainsi que celle de la pile puisque l'on cesse d'ajouter des mots au dictionnaire, donc d'appeler les fonctions de gestion récursive de l'arbre binaire de recherche. A partir de la capture n°60, on n'utilise plus aucun octet du tas : tous les blocs alloués ont été libérés, il n'y a pas de fuite mémoire.

Au global, quelques petites optimisations mémoire sont envisageables, comme une allocation dynamique de l'énorme tableau évoqué ci-dessus, ou le passage par pointeur de toutes les structures, y compris **mot_data_t**. Mais pour de gros fichiers d'entrée, le gain serait peu intéressant par rapport au travail de restructuration du code et des tests que cela demanderait.

```
--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 50         1,321          190,208          174,224       15,456          528
 51         1,324          186,632          171,872       14,280          480
 52         1,327           52,248           38,312       13,136          800
 53         1,330           44,912           33,056       11,040          816
 54         1,333           33,928           25,352        7,952          624
 55         1,336           22,880           17,376        4,768          736
 56         1,339           11,768            9,512        1,600          656
 57         1,342            1,520            1,024            8          488
 58         1,345            1,608            1,024            8          576
 59         1,348            1,536            1,024            8          504
66.67% (1,024B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->66.67% (1,024B) 0x49D1BA3: _IO_file_doallocate (filedoalloc.c:101)
| ->66.67% (1,024B) 0x49E0CDF: _IO_doallocbuf (genops.c:347)
|   ->66.67% (1,024B) 0x49DFF5F: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:744)
|     | ->66.67% (1,024B) 0x49DE6D4: _IO_new_file_xsputn (fileops.c:1243)
|     |   ->66.67% (1,024B) 0x49DE6D4: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1196)
|     |     ->66.67% (1,024B) 0x49D3F1B: puts (ioputs.c:40)
|     |     | ->66.67% (1,024B) 0x109987: main (in /home/alexdet/MOCA/team2/Semaine9/Projet/dico)
|     |     |
|     |     ->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
|     |
|   ->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 60         1,351              496                0            0          496
 61         1,354              576                0            0          576
```

III. Autres modifications sur le projet

Nous avons aussi remarqué que les résultats de notre dictionnaire n'étaient pas ceux attendus pour le projet. On avait le résultat de chaque mot de la forme (ligne où le mot apparaît, colonne où le mot apparaît indenté par le nombre de mots).

Donc par exemple sur le fichier **hugo1.txt** pour le mot **aller** où on attendait le tuple **(2, 6)**, on avait **(2, 2)** car c'était le 2ème mot de la ligne. Nous avons donc modifié cela en incrémentant les colonnes dans la boucle de création d'un mot où l'on lit caractère par caractère sans les séparateurs dans le fichier **next_word**.

Nous avons donc dû changer les tests sur cette fonction pour convenir à cette implémentation car on pensait initialement au fait que les colonnes pour les mots étaient incrémentées par les mots et non les caractères.