

# COMPTE RENDU DE PROJET : Détection automatique de langue.

SDA2

JOCHYMSKI Hugo  
WERNERT Thomas

Décembre 2020

---

## I. Introduction

Dans ce projet il était question d'implémenter une détection automatique de langue sur une phrase donnée par l'utilisateur. Deux approches ont été proposées, à savoir l'utilisation d'un **trie** (ou **arbre préfixé**) ainsi que d'un **DAWG** (**D**irected **A**cyclic **W**ord **G**raph). Nous verrons dans ce rapport les différents points clés de l'implémentation de ces deux structures. Nous examinerons ensuite les choix réalisés ainsi que les performances de notre implémentation.

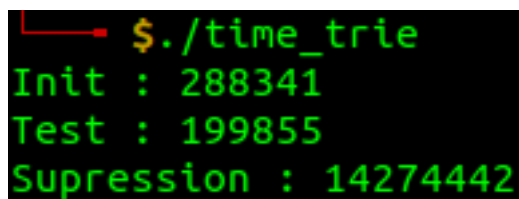
## II. Trie

### A. Problèmes rencontrés

Le **trie** n'a pas posé de soucis particulier. Il reprenait en effet les concepts vus en cours pendant le semestre, ce qui a permis une implémentation aisée et rapide.

### B. Performances

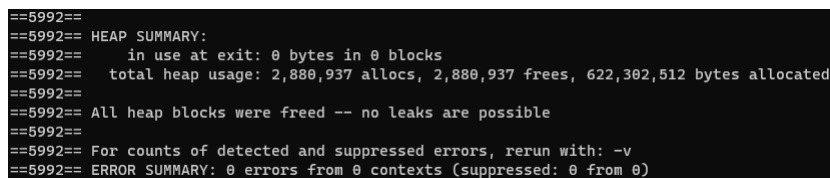
Voici un test de performance issu de la commande `make tempsTrie`. 100 itérations de chaque étape ont été réalisées pour obtenir cette moyenne.



```
➤ $./time_trie
Init : 288341
Test : 199855
Supression : 14274442
```

Figure 1: Moyenne en nombre de ticks d'horloge des étapes de détection par **trie**

On utilise `valgrind` pour estimer l'espace mémoire utilisé par la structure.



```
==5992==
==5992== HEAP SUMMARY:
==5992==   in use at exit: 0 bytes in 0 blocks
==5992==   total heap usage: 2,880,937 allocs, 2,880,937 frees, 622,302,512 bytes allocated
==5992==
==5992== All heap blocks were freed -- no leaks are possible
==5992==
==5992== For counts of detected and suppressed errors, rerun with: -v
==5992== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 2: Capture écran de `valgrind` en utilisant un **trie**

### III. DAWG

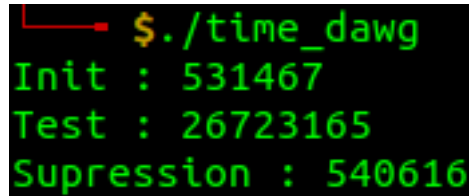
#### A. Problèmes rencontrés

Le dawg fut compliqué à implémenter. Déjà de par la présence de la hashmap. En effet, le temps de saisir son fonctionnement et son application dans le programme. Néanmoins la minimisation n'était pas si dure à mettre en oeuvre une fois que la hashmap est comprise. Cependant, l'insertion d'un mot a été plus dur pour nous !

Lors de l'implémentation du **DAWG**, nous avons subi de grosses fuites mémoire que nous avons finalement réussi à patcher. Pendant la phase de debug nous avons opté pour l'utilisation de `memset` lors de l'initialisation, permettant un contrôle plus précis de la structure que `calloc`. La solution était de minimiser jusqu'à la profondeur 0 dans `construct_dawg` pour optimiser l'espace et permettre une libération totale.

#### B. Performances

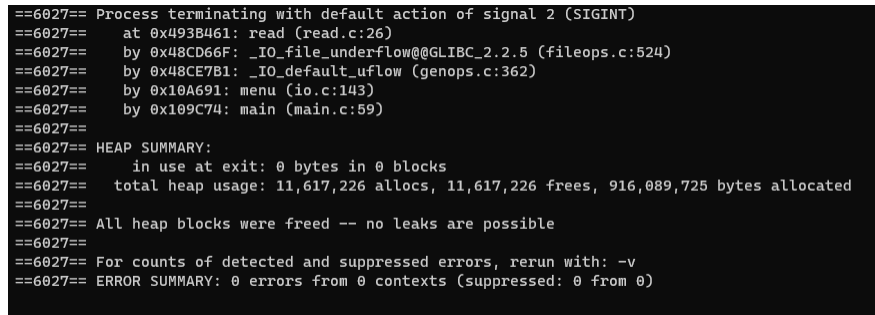
Voici un test de performance issu de la commande `make tempsDawg`. 100 itérations de chaque étape ont été réalisées pour obtenir cette moyenne.



```
➡ $./time_dawg
Init : 531467
Test : 26723165
Supression : 540616
```

Figure 3: Moyenne en nombre de ticks d'horloge des étapes de détection par **DAWG**

On utilise `valgrind` pour estimer l'espace mémoire utilisé par la structure.



```
==6027== Process terminating with default action of signal 2 (SIGINT)
==6027== at 0x493B461: read (read.c:26)
==6027== by 0x48CD66F: _IO_file_underflow@@GLIBC_2.2.5 (fileops.c:524)
==6027== by 0x48CE7B1: _IO_default_uflow (genops.c:362)
==6027== by 0x10A691: menu (io.c:143)
==6027== by 0x109C74: main (main.c:59)
==6027==
==6027== HEAP SUMMARY:
==6027== in use at exit: 0 bytes in 0 blocks
==6027== total heap usage: 11,617,226 allocs, 11,617,226 frees, 916,089,725 bytes allocated
==6027== All heap blocks were freed -- no leaks are possible
==6027==
==6027== For counts of detected and suppressed errors, rerun with: -v
==6027== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4: Capture écran de `valgrind` en utilisant un **DAWG**

### IV. Mise en relation

Les performances en terme de recherche semblent similaires entre les deux structures, quoique légèrement plus rapide pour le **DAWG**. La génération du **DAWG** est plus lente que son homologue, le chargement du dictionnaire dans ce dernier prend du temps de par l'emploi d'autres structures (**hashmap**, **pile**). Elle emploie aussi largement plus de `malloc`. Cependant, l'étape de la minimisation permet de limiter sa taille en mémoire. Le **DAWG** est donc plus compact en mémoire. Cela favorise son utilisation sur des machines au stockage limité comme les systèmes embarqués par exemple. Comme vu au dessus, le temps de recherche est, très légèrement, supérieur au trie.

A partir des résultats obtenus, nous pouvons en déduire trois points principaux :

- Les deux ont la même vitesse de recherche.

- La génération du **DAWG** est plus lente.
- Le **DAWG** prend moins de place en mémoire.

## V. Choix réalisés

### A. Interface utilisateur

Nous avons pris le parti d’implémenter une interface graphique pour l’utilisateur. Vous l’obtiendrez par défaut en lançant le projet suite à un appel à **make**. Cependant, vous pouvez décider de la passer en utilisant les options de lancement suivantes :

- **-T** ou **--trie** : Lancera la détection de la phrase avec un **trie**.
- **-D** ou **--dawg** : Lancera la détection de la phrase avec un **DAWG**.
- **-t=0** ou **-t=1** : Permet de choisir si vous souhaitez voir le temps d’exécution affiché à la fin de la détection (**=1**) ou non (**=0**)

Veuillez noter qu’après la première détection, le programme vous renverra au menu principal, dans lequel vous pourrez modifier vos préférences.

### B. Compteurs

Afin de déterminer la langue de la phrase de l’utilisateur, nous avons implémenté un système de compteurs en suivant une logique très simple : ajouter 1 à un compteur dédié par langue si le mot traité se trouve dans le dictionnaire de cette dernière. Cependant, un souci s’est posé : celui du cas où les mots se trouveraient dans plusieurs dictionnaires à la fois. Pour y pallier nous avons utilisé un marqueur, utilisé lors de la détection, permettant de connaître la langue du dernier mot traité. Ensuite, si le mot traité est dans la même langue que le précédent, alors le compteur sera augmenté deux fois plus que si il ne l’était pas.

On se base sur le fait qu’en partant du principe que si la phrase est en français, il est plus probable que deux mots présents, par exemple, également en anglais soient côte à côte plutôt que dispersés dans la phrase.

## VI. Conclusion

En définitif, nous avons trouvé ce projet moyennement difficile. En effet, la conceptualisation du dawg ainsi que son application nous grandement ralentit dans la réalisation du projet. Mais, nous avons trouvé intéressant de pouvoir recourir à deux implémentations différentes pour détecter la langue dans une phrase donnée. Les tests comparatifs nous ont permis d’apprécier les deux méthodes d’implémentation.