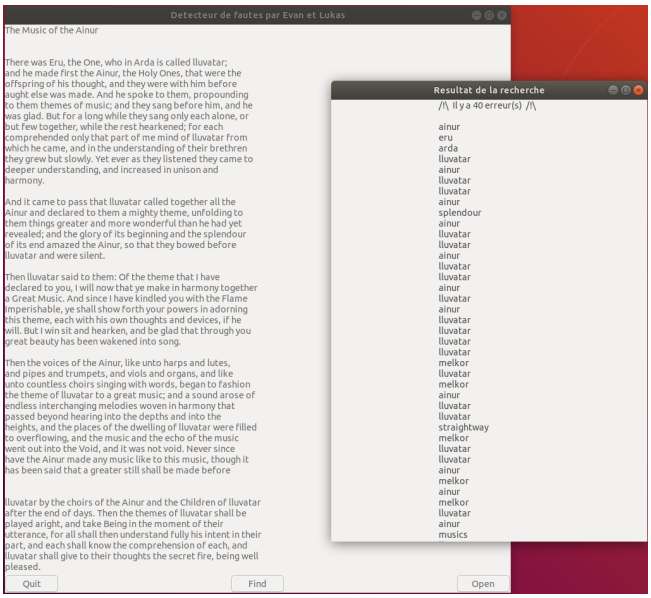


PROJET DE PROGRAMMATION AVANCEE

Evan GURY & Lukas FAUCHOIS

SUJET

Réaliser un programme effectuant une recherche d'erreurs dans un texte à partir d'un dictionnaire de référence

Table des matières

Introduction

Dans le cadre du projet de Programmation Avancée, notre objectif est de réaliser un programme permettant de détecter dans un texte quelconque tous les mots mal orthographiés en les comparant à un dictionnaire chargé au préalable par notre programme.

Pour pouvoir le réaliser, nous avons dû structurer nos données en passant par une arborescence et réaliser les algorithmes nécessaires à l'atteinte de l'objectif principal.

Au travers de ce compte-rendu, nous allons vous exposer les choix que nous avons fait pour mener à bien le projet et vous présenter les différents obstacles que nous avons pu rencontrer.

Dans un premier temps, nous détaillerons le cahier des charges imposé, nous analyserons ensuite le projet en expliquant nos choix de structuration. Nous expliquerons également nos algorithmes principaux et les résultats en(qui en) découlant. Pour finir, nous commenterons les limites de notre programme.

1 – Cahier des charges

Pour pouvoir commencer la réalisation de notre projet, nous prenons d'abord connaissance du cahier des charges imposé.

Il faut tout d'abord définir et implémenter une structure de données permettant de stocker et de manipuler un dictionnaire quelconque. Pour structurer ces données il est nécessaire de construire un arbre préfixe. Le dictionnaire doit être chargé à partir d'un fichier texte, nous choisirons un dictionnaire anglais sans accent. En effet traiter les accents étant proposé comme un bonus, nous avons préféré prendre ce temps pour réaliser une interface graphique par envie de découverte n'ayant tous les deux jamais réalisé une interface graphique.

La dernière consigne à respecter est d'analyser une phrase ou un texte en indiquant à l'utilisateur le nombre de mots qui ne sont pas présents dans le dictionnaire implémenté, de plus nous avons décidé d'afficher ces erreurs.

Après avoir pris connaissance de ce cahier des charges nous pouvons désormais analyser le projet et définir nos choix de structures.

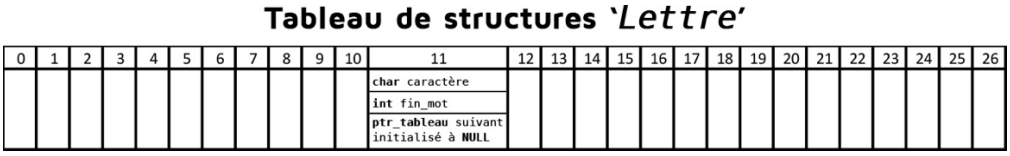
2 – Analyse du projet

2.1 – Structure de données

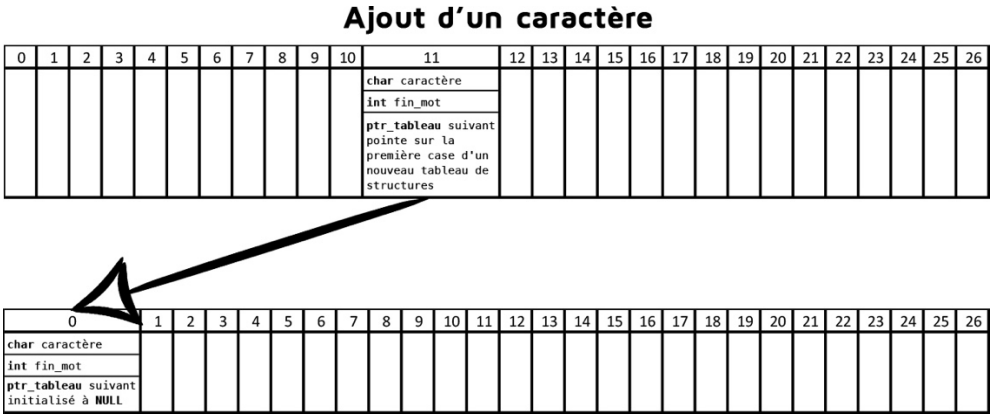
Pour structurer convenablement nos données, nous décidons premièrement de définir un pointeur de tableau `ptr_tableau` qui pointe sur une structure `lettre`. La structure `lettre` étant paramétrée par un caractère, un entier qui indiquera si l'on est à la fin d'un mot et un pointeur de tableau.

```
typedef struct lettre* ptr_tableau;
typedef struct lettre{char caractere; ptr_tableau
suivant; int fin_mot}Lettre;
```

Chaque caractère est stockée dans un tableau de structures à la case correspondant à son code $ASCII - 97$ (par exemple, si l'on souhaite ajouter le caractère [k], la structure lettre associé à [k] sera stockée dans la case 10 du tableau de structures). Nous convertissons chaque caractère en minuscule pour éviter d'avoir à ajouter les caractères en majuscule dans le tableau. Un tableau de structure est de taille 27 étant donné qu'il contient tout l'alphabet et le caractère apostrophe. Voici un schéma permettant de vous illustrer ce tableau de structures :



A l'ajout d'un caractère grâce à la fonction `ajout_caractere`, nous enregistrons le nouveau caractère et faisons pointer le pointeur associé sur la première case d'un nouveau tableau de structures en initialisant les pointeurs de tableau de chaque structures à **NULL** comme le montre le schéma suivant :



Pour gérer le problème du choix de structure, nous nous sommes mis d'accord que, dans ce type de programme, la principal problématique est la rapidité d'exécution. En effet au vu de la quantité de mot dans un dictionnaire et de la possible quantité de mot à rechercher et corriger, nous avons décidé de sacrifier de la mémoire pour gagner en efficacité. Chaque caractère d'un mot se trouvant dans une case mémoire d'un tableau que nous connaissons à l'avance, il est bien plus rapide d'enregistrer et de trouver ce dernier que s'il se trouvait dans une liste chainée.

2.2 – Structure du programme

Concernant la structure de notre programme, nous nous sommes lancé dans un programme simple qui stocke tout d'abord le dictionnaire, qui corrige le texte et qui, finalement, libère la mémoire utilisée comme le montre la figure 1 de l'**Annexe 1**.

Le stockage du dictionnaire choisi se fait à partir d'un fichier texte, que la fonction `ajout_dico` ouvre et parcourt mot par mot, une fois qu'il lit un mot il le convertit en minuscule avec `conversion_mot` et l'ajoute à notre structure principal par la fonction `ajout_mot` que nous décrirons dans la suite du rapport (on notera que si le mot possède un préfixe déjà présent dans notre structure alors il complétera en ajoutant seulement le suffixe à la suite, exemple : salut et salutation). A la fin du fichier, celui-ci est évidemment fermé. (Voir la figure 2 de l'**Annexe 1**)

Après avoir construit le dictionnaire, le programme va corriger le texte souhaité stocké lui aussi dans un fichier texte. La fonction `correction_texte` ouvre et parcourt le fichier mot par mot. A chaque lecture d'un mot, elle le convertit également en minuscule puis le compare au dictionnaire stocké en passant par la fonction `presence_mot`. Si le mot n'est pas présent alors elle le stocke et incrémente un compteur d'erreur. A la fin du fichier, celui-ci est fermé, le nombre d'erreurs ainsi que ces erreurs seront alors affiché à l'utilisateur. (Voir la figure 3 de l'**Annexe 1**)

Voici d'un point de vue général comment nous avons décidé de structurer notre programme. Nous allons pouvoir maintenant vous donner plus de détails sur les fonctions et les algorithmes utilisés dans ces fonctions principales.

3 – Description des algorithmes principaux

Pour pouvoir expliquer clairement notre programme, nous allons suivre sa structure et expliquer comment fonctionne les deux fonctions principales `ajout_dico` et `correction_texte`, ce qui comprend la conversion des mots en minuscule ainsi que la libération de la mémoire.

3.1 – Fonctions nécessaires

3.1.1 – Conversion d'un mot en minuscule

Pour pouvoir faciliter les choses, nous décidons de convertir tous les mots en minuscules. Nous créons alors une fonction `conversion_mot` qui fonctionne de façon simple. Un mot est une chaîne de caractères passée en paramètre, la fonction parcourt cette chaîne caractère par caractère grâce à une boucle tant que la fin du mot n'est pas atteinte.

A chaque caractère, on vérifie que celui-ci possède un code ASCII compris entre 65 et 90 (lettres majuscules), si c'est le cas alors on le

remplace par ce le caractère correspondant en minuscule en ajoutant 32 à ce code ASCII :

```
if(mot[i]>64 && mot[i]<91) mot[i]+=32;
```

La fonction étant entièrement rédigée dans le code joint **projetPA/projetBase/fonction.c**.

3.1.2 – Libération de la mémoire

La fonction `lib_memoire` libère quant à elle la mémoire utilisée par chaque pointeur de tableau qu'elle prend en paramètre étant donné que ces pointeurs sont des `malloc()` comme on le verra dans la fonction `ajout_caractere`. Elle parcourt grâce à une boucle le tableau pointé et libère chacune de ses cases par récursivité (voir **projetPA/projetBase/fonction.c**)

3.2 – Ajout du dictionnaire

3.2.1 – Ajout d'un caractère

Pour construire notre dictionnaire, il faut d'abord comprendre comment se fait l'ajout d'un caractère dans la fonction `ajout_caractere`.

Comme expliqué dans la partie structure de données, une structure `Lettre` est une caractérisé par un caractère, un entier indiquant si celui-ci clôture un mot et un pointeur de tableau.

La fonction `ajout_caractere` prend alors en paramètre un caractère et pointeur de structure. On associe alors le caractère en paramètre et on fait pointer cette `Lettre` vers la première case d'tableau de structures en allouant un espace mémoire de la taille d'un tel tableau au pointeur de tableau `suivant` soit de la taille `(27*sizeof(Lettre))` sachant que ce pointeur était initialement NULL.

On initialise alors tous les ptr_tableau du tableau de structures pointé à NULL.

3.2.2 – Ajout d'un mot

L'ajout d'un mot se fait grâce à la fonction `ajout_mot` qui prend en paramètre un pointeur de structure, une chaîne de caractères déclarée comme pointeur sur `char` ainsi qu'un entier indiquant la position où nous sommes dans la chaîne de caractères.

On convertit dans un premier temps le code ASCII du caractère sur lequel on pointe dans le mot en un indice pour connaître sa position dans le tableau de structures (expliqué en 2.1) en lui soustrayant 97.

Ensuite, on contrôle si le caractère lu n'est pas la fin du mot '/0', si c'est le cas on quitte alors la fonction puisqu'on a fini d'ajouter le mot.

Dans le cas où le mot n'est pas fini, alors on vérifie que le caractère que l'on souhaite ajouter n'est pas encore enregistré dans la structure principal, comme expliqué dans le 2.2 si le mot « salut » est déjà ajouté alors si nous voulons ajouter le mot « salutations » nous ajoutons seulement le suffixe « ations ». Si le caractère est ajouté, on parcourt le mot par récursivité jusqu'à trouver le caractère qui pointera sur un tableau de structures vide. A ce moment-là, nous appellerons la fonction `ajout_caractere` et passerons à l'indice suivant de la chaîne de caractère pour finalement rappeler la fonction `ajout_mot` avec en paramètre le caractère suivant. Notons que dans le cas où le caractère est un apostrophe nous exécuterons les mêmes commandes mais à l'indice 26 du tableau de structures.

Il faut ajouter que quel que soit le chemin parcouru, si le caractère suivant est le caractère de fin de chaîne ' \0 ' alors nous passons l'entier fin_mot à 1 pour indiquer que la structure `Lettre` ajoutée clôture un mot.

3.2.3 – Construction du dictionnaire

Pour construire le dictionnaire, nous suivons les indications données dans la partie 2.2, la fonction `ajout_dico` ouvre un fichier et le parcourt mot par mot. Chaque mot est stocké comme une chaîne de caractères, converti en minuscule par `conversion_mot` et ajouté grâce

à `ajout_mot`. Une fois la fin du fichier atteinte, le fichier est fermé et l'exécution de la fonction est terminée.

3.3 – Correction du texte

Après avoir ajouté un dictionnaire de référence, il faut corriger le texte voulu grâce à la fonction `correction_texte`.

3.3.1 – Vérification de la présence d'un mot

La fonction la plus importante de cette correction est la fonction `presence_mot` qui va vérifier si oui ou non un mot est dans le dictionnaire de référence.

Cette fonction prend un pointeur de structure, une chaîne de caractères déclarée comme pointeur sur `char`, un entier qui indique l'indice du caractère comparé et un entier qui indique la fin du mot. Elle retourne 1 dans le cas où le mot existe et 0 sinon.

Tout d'abord, au début d'un mot, il faut convertir le code ASCII du caractère en un indice i de tableau de structures compris entre 0 et 26. Si le caractère lu est compris entre 97 et 122 alors l'indice dans le tableau est égale à son code $ASCII - 97$. S'il est égal à 39 (apostrophe) alors l'indice est 26 (dernière case du tableau de structures). Xxxxx

Si à l'indice i du tableau de structure associé on trouve le caractère recherché alors on relance la recherche en passant au caractère suivant et on met l'entier de fin de mot à 1 si l'indice de `fin_mot` de la structure `Lettre` est à 1 sinon on retourne 0.

Si le caractère recherché est '\0' alors on retourne 1 si l'entier de fin de mot est égal à 1, sinon on retourne 0 ce qui signifie que le mot n'existe pas.

De plus si la `Lettre` ne pointe sur aucune `Lettre` alors on retourne 0 aussi.

3.3.2 – Comptage et stockage des erreurs

Pour réaliser le comptage des erreurs et le stockage des erreurs dans la fonction `correction_texte` qui prend en paramètre un fichier et un pointeur de structure.

Il faut tout d'abord initialiser un compteur à 0 ainsi qu'un tableau permettant le stockage des mots erronés en utilisant un `malloc()`.

Le fichier texte comprenant le texte à corriger est alors ouvert et parcouru mot par mot. Chaque mot est converti en minuscule et est contrôlé par la fonction `presence_mot` précédemment expliquée. Si la fonction retourne 1 alors on passe au mot suivant sinon le compteur d'erreurs est incrémenté et le mot est stocké dans la variable stockage_erreur et on agrandit le stockage grâce à un realloc (voir **`projetPA/projetBase/fonction.c`**).

A la fin du fichier, le fichier est fermé et on retourne par l'intermédiaire d'un `printf()` le nombre d'erreurs ainsi que les mots correspondants à ces erreurs.

3.4 – Une interface graphique

Tous les algorithmes vu précédemment ont pour finalité de réaliser un projet avec un rendu graphique.

Le solution sélectionnée a été l'utilisation de la bibliothèque `gtk`. Après avoir lu de nombreux tutoriels nous nous sommes lancés dans la réalisation de notre propre interface graphique.

`gtk` marche sur un principe d'interruption, c'est-à-dire que chaque action sur notre fenêtre principal génère des signaux, signaux qui peuvent à leur tour être traité dans le fichier `callback.c` pour appeler nos fonctions. (voir **`projetPA/projetGraphique/callback.c`**)

Notre interface dispose de quatre boutons, appelé `Widget` dans la bibliothèque, "`Quit`", "`Find`", "`Open`" et "`X`". `Quit` et `X` permettent tous deux de fermer la fenêtre principale. `Open` permet quant à lui d'ouvrir un fichier texte et de l'importer dans la zone de saisie pour le corriger et enfin `Find` permet de

compter et détecter les erreurs grâce à la fonction `correction_texte` que nous avons modifié spécialement pour cette interface graphique (**`projetPA/projetGraphique/fonction.c`**)).

L'utilisateur a le choix entre taper son texte directement dans la zone de saisie et le corriger ou ouvrir un fichier et le corriger. Il peut également importer un fichier et le modifier directement dans la zone de saisie.

Comme cité plus haut, certaines spécificités de la bibliothèque `gtk` nous ont contraints à modifier la fonction de correction du texte. En effet, le texte à corriger n'est cette fois plus un fichier mais une simple chaîne de caractères, les conditions de vérification ont donc changé. Cependant cette méthode s'est avérée bien plus performante, notamment au niveau de la gestion des ponctuations qui nous a posé quelques souci dans le programme initial sans interface graphique. Nous n'avons pas modifier notre premier programme dans le but de garder le même principe de fonctionnement puisque nous considérons notre interface graphique comme la finalité du projet, alors que le premier n'en est que la base.

4 – Résultats de l'exécution

4.1 – Programme basique

Lors de l'exécution du programme de base dans le terminal, nous avons un affichage assez simple qui dit soit qu'il n'y a pas d'erreur soit qu'il y en a en les affichants ensuite comme le montre les deux images suivantes :

- ***Exécution sans fautes***

```
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$ time ./projet american-english-no-accents texte_vide.txt

Dans le texte il y a 0 erreur(s)

real    0m0,121s
user    0m0,077s
sys     0m0,044s
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$
```

- ***Exécution avec fautes***

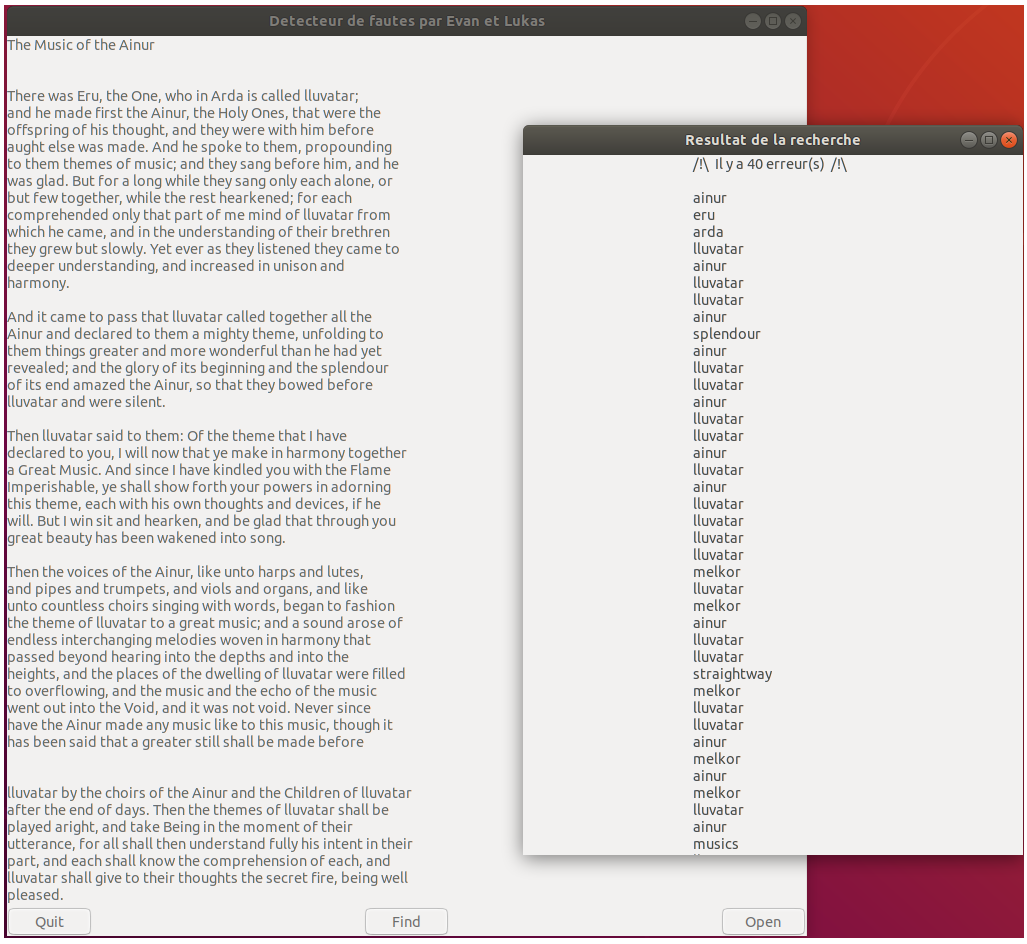
```
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$ time ./projet american-english-no-accents correction.txt

Dans le texte il y a 1 erreur(s)
Les erreurs sont les suivantes :
plummmage
real    0m0,114s
user    0m0,075s
sys     0m0,039s
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$
```

Nous pouvons constater que le temps d'exécution est très court, ce que nous voulions.

4.2 – Programme avec interface graphique

Concernant l'exécution avec notre interface, nous pouvons observer un résultat similaire mais plus esthétique, le temps d'exécution reste inchangé mais l'utilisation et l'affichage sont plus agréables pour l'utilisateur comme le montre l'illustration suivante :



5 – Limites du programme

5.1 – Gestion de la ponctuation

Le premier programme (sans `gtk`) trouve ses limites dans la gestion de certaines ponctuations. En effet si un mot commence par une ponctuation, il sera

considéré comme faux. Cependant ce problème a été corrigé dans la partie graphique où cette fois-ci, l'utilisateur peut entrer les ponctuations qu'il veut dans un ordre quelconque, ces dernières ne seront pas considérées comme des fautes. Nous avons fait le choix de ne pas corriger les fautes de syntaxe mais uniquement les fautes d'orthographe. De plus comme dit précédemment nous n'avons souhaité corriger les accents.

5.2 – Libération de la mémoire

Une autre limite de notre programme se trouve dans la libération de la mémoire, nous pouvons remarquer, grâce aux captures d'écran que dans le cadre du second dictionnaire beaucoup plus grand, une petite partie de l'espace alloué est perdu contrairement au premier dictionnaire et nous n'avons malheureusement pas réussi a trouver la source du problème.

• ***Chargement du petit dictionnaire :***

23 983 allocs libérés sur un total de 23 983.

```
==6390==
==6390== HEAP SUMMARY:
==6390==     in use at exit: 0 bytes in 0 blocks
==6390==   total heap usage: 23,983 allocs, 23,983 frees, 15,547,446 bytes allocated
==6390==
==6390== All heap blocks were freed -- no leaks are possible
==6390==
==6390== For counts of detected and suppressed errors, rerun with: -v
==6390== Use --track-origins=yes to see where uninitialised values come from
==6390== ERROR SUMMARY: 24077 errors from 50 contexts (suppressed: 0 from 0)
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$
```

• ***Chargement du grand dictionnaire***

225 267 allocs libérés sur un total de 227 031.

```
==6446==
==6446== HEAP SUMMARY:
==6446==     in use at exit: 1,143,072 bytes in 1,764 blocks
==6446==   total heap usage: 227,031 allocs, 225,267 frees, 147,122,550 bytes allocated
==6446==
==6446== LEAK SUMMARY:
==6446==    definitely lost: 484,056 bytes in 747 blocks
==6446==    indirectly lost: 659,016 bytes in 1,017 blocks
==6446==      possibly lost: 0 bytes in 0 blocks
==6446==    still reachable: 0 bytes in 0 blocks
==6446==         suppressed: 0 bytes in 0 blocks
==6446== Rerun with --leak-check=full to see details of leaked memory
==6446==
==6446== For counts of detected and suppressed errors, rerun with: -v
==6446== Use --track-origins=yes to see where uninitialised values come from
==6446== ERROR SUMMARY: 226068 errors from 57 contexts (suppressed: 0 from 0)
evan@Evan:~/Documents/Programmation/ima3_projet_pa_2019$
```

Conclusion

Pour conclure, nous pouvons remarquer la proximité entre le résultat et les caractéristiques du cahier des charges. Nous avons pu réaliser un programme permettant, dans un premier temps, de définir et de manipuler un dictionnaire en le chargeant à partir d'un fichier texte. Et ayant pour finalité de corriger une phrase ou un texte, à partir d'un fichier texte également, en le comparant à ce dictionnaire.

Nous avons pu améliorer les attentes initiales en ajoutant l'affichage des erreurs repérées ainsi qu'un interface graphique facilitant l'utilisation de notre application.

Malgré quelques limites, le programme effectue les tâches que nous lui demandons sur un texte simple sans accents qui auraient nécessité un encodage en UTF-8. Nous avons préféré utiliser le temps que nous avions pour la réalisation de ce projet en créant l'interface `gtk`.

Si nous avions une suite à donner à ce projet, elle pourrait être de réaliser un correcteur proposant un remplacement des erreurs par leur orthographe correct dans le dictionnaire.

Table des annexes

ANNEXE 1 : ALGORIGRAMMES

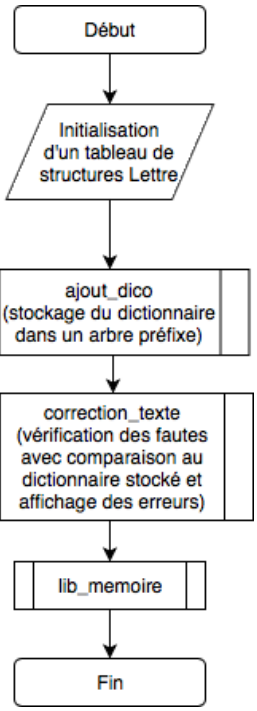


Figure 1 : Algorigramme programme principale

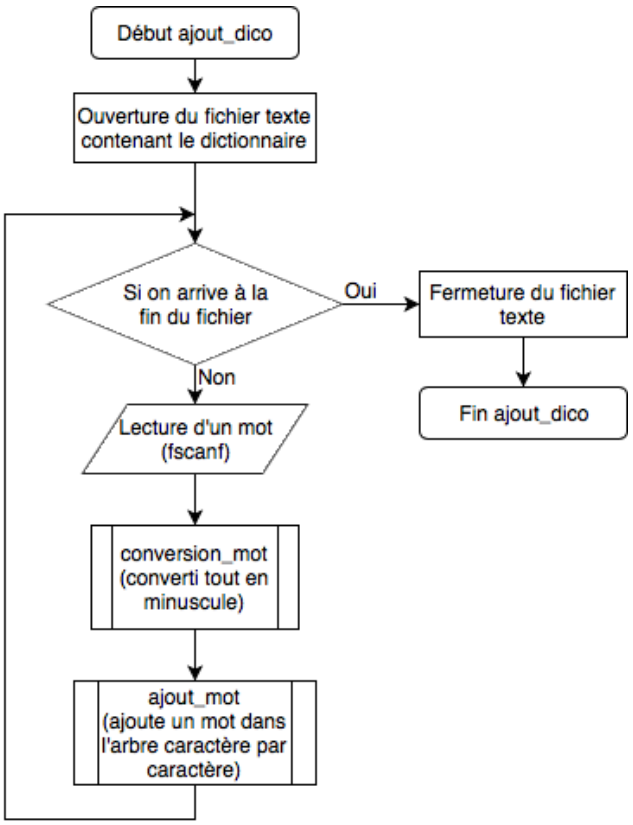


Figure 2 : Algorigramme fonction ajout_dico

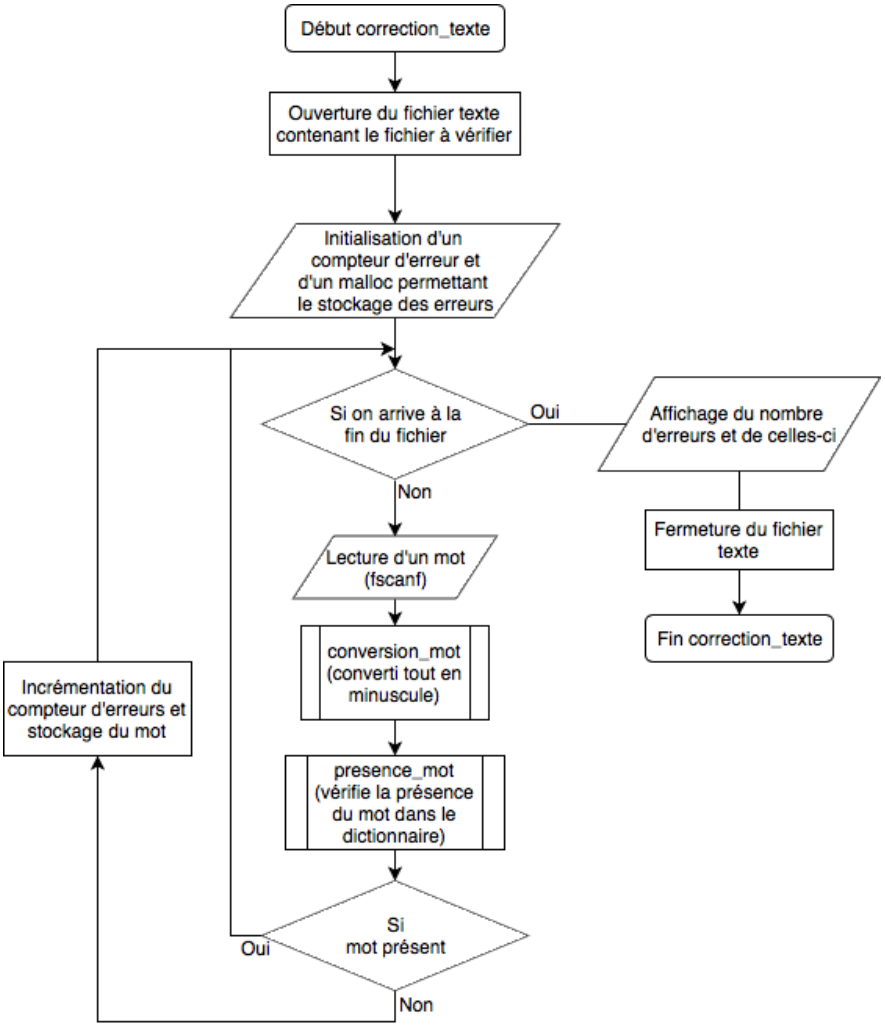


Figure 3 : Algorigramme fonction correction_texte