SALINGUE Valériane
VOLLERIN Thibault

IMA3

Rapport projet de Programmation Avancée
Réalisation d'un détecteur de fautes

année 2018-2019

Introduction

Le projet de Programmation Avancée consiste à construire un dictionnaire à partir d'un fichier de référence et de pouvoir ensuite utiliser ce dictionnaire pour corriger d'autres fichiers textes. Nous avons à notre disposition un dictionnaire anglais de 99 000 mots et un deuxième dictionnaire anglais qui lui ne contient pas d'accents. Pour permettre un temps de recherche bas et utiliser le moins d'espace mémoire, nous utiliserons la structure "Arbre".
Le but de ce projet est de concrétiser et de réunir les différentes connaissances acquises lors de ce semestre pendant les séances de cours, de TD ou de TP.
Nous avons réalisé ce projet en binôme, ce qui nous a permis d'utiliser le dépot Git afin d'optimiser le travail collaboratif en réunissant les différents fichiers dans un projet accessible aux deux personnes du binôme.

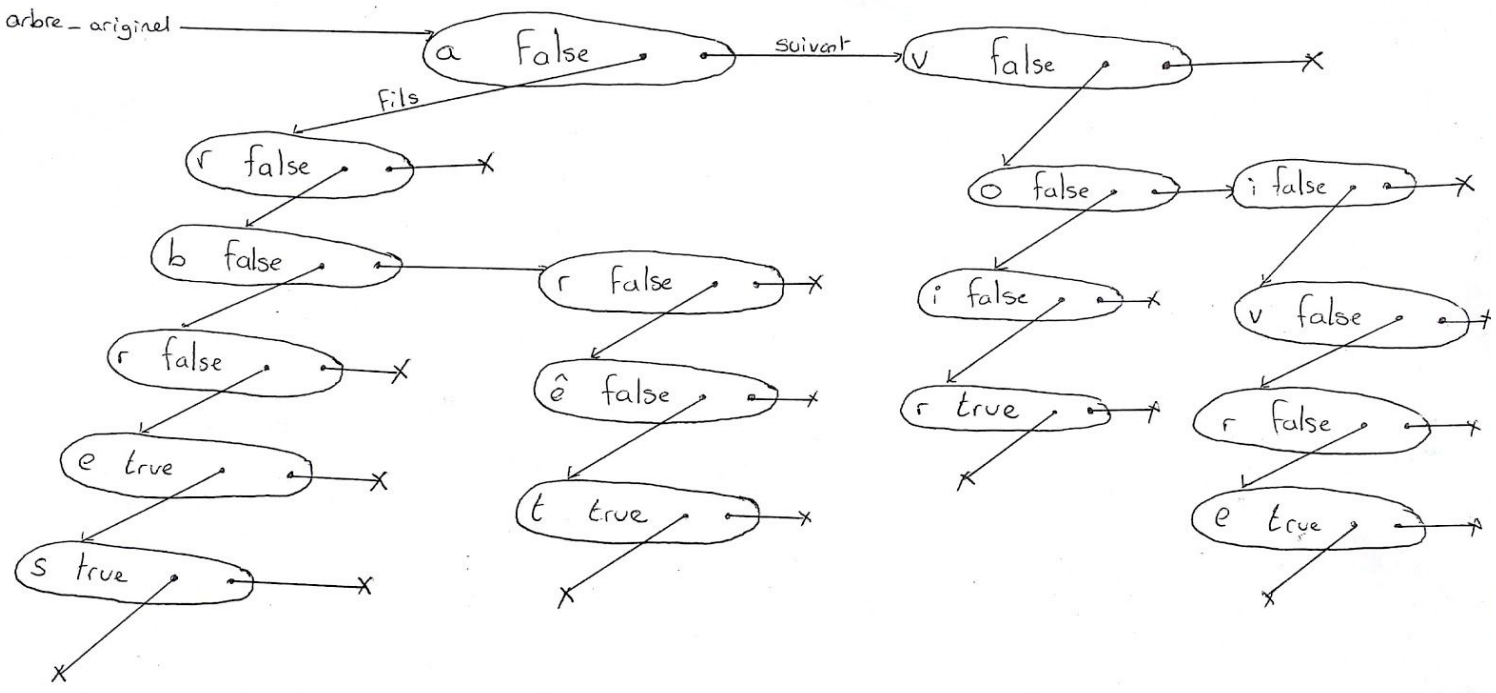
Sommaire:

1. Présentation de notre structure

Pour implémenter le dictionnaire, nous avons choisi la structure définie par :

```
typedef struct cell* ptarbre;
typedef struct cell* ptcellule;
typedef struct cell {
  char lettre;
  ptarbre fils;
  ptcellule suivant;
  bool fin_mot;
} cell;
```

Une cellule contient une lettre, un pointeur de cellule “fils” qui pointe sur la cellule contenant la lettre suivante dans le mot, et un pointeur de cellule “suivant” qui pointe sur la cellule contenant une autre lettre de même indice que la lettre de la cellule. Enfin, nous avons rajouté un booléen “fin_mot” initialement à false qui facilite la comparaison. Lors de la construction du dictionnaire, “fin_mot” de la dernière lettre d’un mot est passé à true. La toute première lettre est à l’adresse arbre_originel. Ensuite, lors de la comparaison, si on arrive au bout du mot, on peut en déduire que le mot existe dans le dictionnaire et qu’il n’y a pas d’erreurs sur ce mot. Si nous avons dans le fichier de référence les mots suivants : “arbre”, “arbres”,”arrêt”, “voir”, “vivre”, nous aurons un dictionnaire sous la forme suivante :



2. Présentation de l’algorithme de construction du dictionnaire

Pour construire le dictionnaire, nous ajoutons une à une les lettres des mots du fichier de référence pour l'ajouter à la structure jusqu’à la fin du fichier.
Pour ajouter, nous avons défini 3 cas. Le premier cas apparaît lorsque le dictionnaire est vide. On commence à compléter le dictionnaire avec le premier mot du fichier de référence. On utilise alors la fonction init_dico car c'est la première lettre du premier mot du dictionnaire. Elle permet d’initialiser la première cellule à partir de l’adresse de l’arbre originel. Le deuxième cas apparaît lorsqu’aucune lettre n’est présente dans la liste. Par exemple, pour le premier mot ‘voir’, la deuxième lettre ‘o’ est en réalité la première lettre à l’indice 2 suivant ‘v’. Si ensuite on ajoute le mot ‘arbre’, ‘a’ n’est pas la première lettre à l’indice 1 mais ‘r’ est la première lettre à l’indice 2 suivant ‘a’. On utilise alors la fonction ajout_dico_tete. Dans le dernier cas, on ajoute une lettre qui suit une autre lettre en utilisant la fonction ajout_dico. En résumé, pour le premier mot 'voir', on utilise init_dico pour le ‘v’ et pour les 3 autres lettres on utilise ajout_dico_tete. Ensuite, pour voile, on utilise ajout_dico pour 'l' et ajout_dico_tete pour 'e' ('v', 'o', 'i' déjà présents dans l'arbre, ‘l’ n’est pas la première lettre de la liste à avoir ‘voi’ comme début de mot). Pour savoir si la lettre est déjà présente dans la liste, on utilise la fonction rech. Cette fonction parcourt la liste de lettres à un indice donné. Elle renvoie une adresse. Soit il s’agit de l’adresse de la cellule où la lettre est présente, donc pas besoin de l’ajouter. Soit il s’agit de l’adresse de la dernière cellule de la liste. Dans ce cas, soit il s’agit d’une lettre à ajouter à la liste (avec ajout_dico), soit il s’agit de la première lettre de la liste (utilisation de ajout_dico_tete).

Toutes ces fonctions s'exécutent tant que le fichier n’est pas fini en prenant chaque mot à part. On considère que la fin d’un mot est le caractère ‘retour à la ligne’.

3. Présentation de l’algorithme de comparaison pour détecter les fautes

Pour cette partie, nous avons créé deux fonctions différentes dans le fichier compare.c. La première fonction (lecture_mot) va récupérer mot par mot le texte à corriger. Chaque mot va être stocké dans un tableau et ensuite comparé aux mots du dictionnaire. Afin de retirer les différents caractères de ponctuation qui peuvent être stocké à la fin du mot, nous avons fait une boucle permettant de comparer chaque caractère du tableau et les remplacer par le marqueur de fin de chaîne ‘\0’ si une ponctuation est trouvée.

Ensuite, nous avons fait une fonction de comparaison (compare) qui va analyser le mot lettre par lettre et comparer ces lettres aux différentes listes de l’arbre. Pour cela, nous avons commencé par demander au programme d’arrêter la fonction si l’arbre est vide. Car en effet, si aucun mot n’a été rentré dans le dictionnaire et par conséquent si l’arbre créer pour représenter ce dictionnaire est vide, nous ne pouvons pas faire de comparaison. Puis pour commencer la comparaison nous faisons une boucle while demandant de continuer les comparaisons de lettres tant que le caractère à analyser n’est pas ‘\0’. Dans cette boucle, nous demandons au programme de faire une scrutation de l’étage correspondant à l’indice du tableau “mot[ ]”. Tant que le programme ne trouve pas le caractère correspondant au caractère du mot analysé, il pointe vers le caractère suivant dans la liste jusqu’à arriver à la fin de la liste représenté par la valeur NULL. Si on arrive à la valeur NULL, cela veut dire que le caractère analysé n’a pas de correspondance dans le dictionnaire et donc qu’il y a une faute. Et dans le cas où le programme trouve le caractère correspondant, le programme va pointer sur la première lettre de l’étage inférieur de l’arbre et incrémenter l’indice du tableau “mot[]” de 1 et recommencer l’analyse. De plus, nous avons ajouté une condition permettant de voir si le mot analysé est faux dans le cas où le mot est constitué des lettres d’un mot présent dans le dictionnaire mais se termine avant (exemple : chi au lieu de chien). Pour cela nous faison une comparaison avec une variable booléenne présente dans la structure de l’arbre qui se met à “true” lorsqu’un mot du dictionnaire est terminé.

Enfin, pour permettre de compter le nombre de fautes dans le texte, nous avons simplement fait un compteur qui s’incrémente lorsqu’une erreur est détectée.

4. Les spécificités et limites de notre programme

La structure que nous utilisons et le mode de recherche permet la plupart du temps de gérer les accents. En effet, nous recherche compare un caractère avec celui de l'arbre, mais aussi sa majuscule ou sa minuscule. Par exemple, si notre dictionnaire est composé des mots 'avoir' et 'étude'. Le mot' Avoir' et 'avoir' sont considérés comme bien orthographié alors que seul avoir se trouve réellement dans le dictionnaire. Pour cela, nous utilisons la différence entre majuscule et minuscule qui est un intervalle régulier dans la table ASCII. Cependant, les accents n'ont pas le même intervalle entre minuscule et majuscule. C'est pourquoi 'étude' sera considéré comme bon car présent dans le dictionnaire mais 'Étude' ne le sera pas.
Aussi, pour construire, le fichier de référence doit avoir un format précis. Nous n'avons pas développé la possibilité de prendre un texte comme fichier de référence mais uniquement des fichiers de types dictionnaire qui sont composés d'une suite de mots séparés par des retours à la ligne.

Conclusion

Ce projet nous a permis de bien maîtriser les différentes structures vues dans le semestre ainsi que la notion de pointeurs. Il nous a permis aussi de nous familiariser avec git. Cependant, nous avons eu quelques aléas en l'utilisant. Nous aurions dû créer chacun une branche pour pouvoir travailler et réunir les deux à la fin mais nous avons tous les deux travaillé en parallèle sur la branche master. Nous avons donc eu un conflit au moment de notre dernier commit. C'est pourquoi nous avons dans notre projet une branche master1 qui correspond à une copie de l'ancienne branche master avec tous les anciens commit et la branche master qui contient la dernière version de tous les fichiers ajoutés lors du dernier commit. Aussi, certains commit manque car nous mettions à jour notre branche de notre répertoire personnel à celle en ligne sans ajouter les fichiers que nous avions modifiés, donc nous ajoutions dans le vide.