GUILLOMON Rémi Mai 2019

Rapport
Projet informatique Semestre 6

I. Objectif

L'objectif du programme est de charger un dictionnaire construit à partir d'un texte ou d'un ensemble de mots de référence puis de détecter dans un texte tous les mots mal orthographiés.

Le cahier des charges est le suivant :

* Définir et implémenter une structure de données permettant de stocker et de manipuler un dictionnaire sous forme d'arbre préfixe / trie.

* Charger un dictionnaire à partir d'un fichier texte de données. Ce fichier texte pouvant être un texte court, un roman ou une liste de mots.

* Analyser l'orthographe d'une phrase ou d'un texte en indiquant les nombres de mots qui ne sont pas reconnus par le dictionnaire.

II. Architecture

Le programme est divisé selon plusieurs fichiers.
- Le fichier **tree.c** contient toutes les fonctions utiles à son bon fonctionnement.
- Le fichier **main.c** constitue le programme en lui-même (chargement du fichier dictionnaire, lecture du fichier à analyser, compte des erreurs et libération du dictionnaire à la fin).
- Enfin, le fichier **annexe.c** contient une fonction annexe qui n'est pas utile pour répondre au cahier des charges mais qui a son intérêt.

Voici une liste exhaustive des fonctions contenues dans chaque fichier.
Un détail de ces fonctions est fourni à la fin de ce rapport.

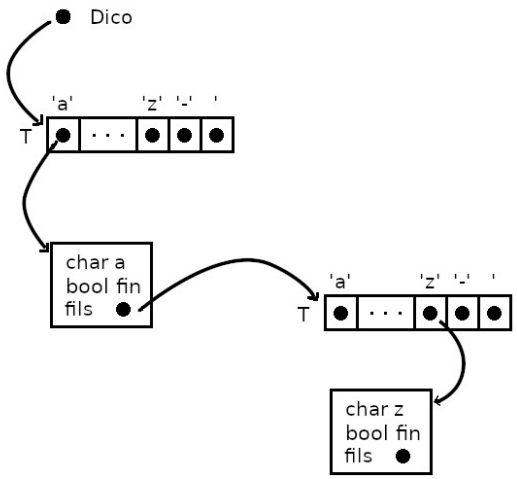
–tree.c :
----**new_node** : crée un nœud
----**init_node** : initialise un nœud selon les paramètres souhaités
----**init_tableau_pt** : initialise un tableau de pointeurs de nœuds passé en paramètres
----**free_tree** : libère un nœud donné et son pointeur devient « NULL »
----**free_tableau_pt** : libère un tableau_pt donné, son pointeur vaut ensuite « NULL »
Remarque : ces deux dernières fonction permettent la libération récursive d'un arbre
----**indice_tab** : à un caractère associe son indice dans tableau_pt
----**casse** : passe un mot en lettres minuscules
----**verif_taille_mot** : vérifie si le mot n'est pas trop grand
----**verif_caractere :** vérifie que le mot ne contient pas de caractère inconnu
----**recherche** : renvoie un booléen associé à la présence d'un mot dans un dictionnaire donné
----**ajout_dico** : ajoute un mot à un dictionnaire (double pointeur de tableau_pt)
----**charge_dico** : permet le chargement d'un dictionnaire à partir d'un fichier
----**charge_texte** : compte le nombre de mots inconnus dans un fichier donné et les affiche

–annexe.c :
----**affiche_tab** : affiche le contenu d'un dictionnaire par récursivité

III. Structure du dictionnaire

La structure du dictionnaire choisie est celle d'un tableau de pointeurs de nœud, du type **struct tableau_pt**.
Ce tableau est composé de 28 cases. 26 sont réservées à l'alphabet, les deux autres au trait d'union et à l'apostrophe.
Chaque nœud (du type **struct node**) est constitué du caractère à stocker, d'un booléen « fin » désignant si ce caractère est la fin d'un mot ou non et d'un pointeur « fils » de tableau_pt.

Cela peut être représenté ainsi :



Structure du dictionnaire

Par ce choix, l'accès à chaque caractère du tableau se fait en temps constant, ce qui permet un algorithme efficace du point de vue de la recherche dans le dictionnaire.
De plus, le tableau n'est pas directement implémenté dans le **struct node** afin de prendre moins d'espace lorsqu'il est la fin d'un mot (8 octets au lieu de 28*8octets) bien que 8 octets supplémentaires soient utilisés pour chaque caractère. Ceci apparaît comme valant la peine.

Il aurait pu être envisagé une allocation dynamique du tableau par laquelle on n'ajouterait que les caractères réellement utilisés et on ré-allouerait de l'espace pour chaque nouveau caractère.
Mais ceci porte l'inconvénient de devoir tester chaque caractère du tableau et on perd l'accès direct lors de la recherche.

IV. Choix des fichiers

a) Le dictionnaire

J'ai choisi comme dictionnaire le fichier **eng_list.txt** trouvé sur internet auquel j'ai soustrait les abréviations qui contenaient des points, des caractères spéciaux comme l'esperluette et parfois des chiffres.
Je préférais celui-ci car le fichier « words » fourni ne semblait contenir que des noms propres.

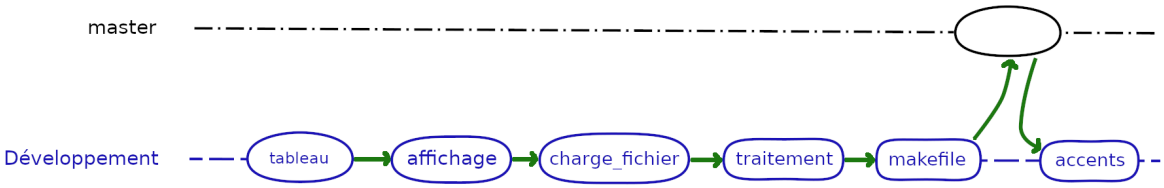
b) Fichier composé d'erreurs

J'ai choisi comme fichier **text.txt** qui est un synopsis d'un bouquin dans lequel j'ai introduit plusieurs types d'erreurs pour tester mon programme.
- Des mots n'existant pas comme « virtortues » ou « samuway »
- Idem avec des apostrophes et des tirets : « manio », le mot composé « king's » n'apparaît pas non plus dans le dictionnaire
- des caractères inconnus : 'é' dans « thé » et des chiffres (1, « mag1cal »)
- un mot qui n'existe pas mais qui est une partie d'un mot plus long :
  « manio » n'existe pas mais « manioc » est dans le dictionnaire

Remarque : un mot de plus d'au moins TAILLE_MOT caractères (nombre défini à 50 dans **tree.h**) peut être introduit dans le fichier **text.txt**. Dans ce cas, la fonction charge_texte renverra un code d'erreur et le programme sera annulé.

V. Gestion de versions sous Github

Au cours de mon projet, j'ai pu utiliser la gestion de version de Github et ai utilisé plusieurs branches dont voici le schéma suivi.



- Tableau contient les fonctions de bases pour implémenter le dictionnaire : création des arbres, libération et main pour le tester.
- Dans affichage a été ajouté la fonction affichant le contenu d'un arbre
- charge_fichier apporte le chargement du dictionnaire par un fichier et l'affichage prend maintenant un tableau_pt en paramètres.
- Traitement est l'implémentation des fonctions pour la lecture du fichier à analyser.
- Makefile correspond à l'adaptation du programme pour le makefile.
- De cette branche est issue la version opérationnelle master.
- Accents est une tentative d'implémentation des accents.

VI Les limites du programme

Le programme ainsi constitué présente plusieurs limites.
L'inconvénient majeur est l'important espace mémoire sacrifié au profit de la rapidité de recherche. D'après Valgrind, 244 000 000 octets sont alloués pour un dictionnaire de 4,9Mo. Ce ratio de 50 est énorme mais doit diminuer pour un dictionnaire initial plus gros.
La rapidité est cependant un net avantage pour la recherche de fautes dans un texte volumineux comme un roman.

Deux pistes peuvent être suivies pour réduire cet espace utilisé :

i) Le caractère dans un **struct node** n'est utilisé que dans la fonction affiche_tab (dans annexe.c) Cette fonction n'est utile qu'à l'affichage éventuel du dictionnaire.
Une gestion totalement implicite de ce caractère est tout à fait envisageable.
Cela diminuerait l'espace alloué d'autant d'octets que de nœuds créés.

ii) Les caractères comme l'apostrophe et le trait d'union sont d'usage très minoritaires.
Nous pouvons imaginer maintenir un accès direct sur les 26 lettres de l'alphabet et ne ré-allouer de l'espace pour agrandir le tableau qu'en cas d'utilisation de ces caractères spéciaux.
Il faudrait cependant adapter les fonctions et considérer le tableau comme une liste contigu et non comme un vecteur de taille déterminée. Ceci rajouterait un octet pour une variable décrivant la fin du tableau, et ce pour chaque nœud.
Le gain ne serait pas énorme ici mais pourrait valoir la peine en cas d'implémentation des accents.

Ensuite, le chargement du dictionnaire mot par mot ne permet pas vraiment sa création à partir d'un texte. C'est pourquoi le fichier destiné à la création du dictionnaire doit être sous forme de liste de mots avec des renvois à la ligne.

Une autre limite est que les accents ne sont pas implémentés et seuls les mots exemptés de caractères spéciaux (différents de l'apostrophe et du trait d'union) ne seront pris en compte dans le dictionnaire.
Ainsi les chiffres isolés dans une phrase ne sont pas reconnus.
Il en est de même pour les traits-d'union en tant que structure de phrase - comme ici.

Enfin, il est dommage que la fonction affiche_tab ne soit pas tout à fait au point, bien que sa fonction soit satisfaite (voir la remarque associée dans la partie suivante).

VII. Détail des fonctions

new_node :
Cette fonction alloue de l'espace pour un nœud si nécessaire. Elle initialise par défaut le *car* caractère à '0', le booléen *fin* à false et le pointeur de **tableau_pt** *fils* à « NULL ».

init_tableau_pt :
Alloue l'espace d'un *tableau_pt* si nécessaire.
Initialise chacune des composantes du tableau (pointeurs de nœuds) à « NULL ».
L'espace pour un nœud fils n'est alloué que lors de la création de celui-ci.

free_tree :
Si un nœud possède un fils (pointeur de **tableau_pt**), elle le libère par la fonction **free_tableau_pt**.
Le nœud est ensuite lui-même libéré et son pointeur est mit à « NULL ».

free_tableau_pt :
Libère chaque composante du tableau_pt par la fonction **free_tree**.
Le tableau est ensuite lui-même libéré et son pointeur est mit à « NULL ».

indice_tab :
Elle renvoie l'indice dans la structure tableau_pt correspondant au caractère passé en paramètre.
Elle distingue donc les lettres de l'alphabet en minuscule (indices de 0 à 25) du tiret (indice 26) et de l'apostrophe (indice 27).
Si le caractère ne correspond à aucun de ces cas, la fonction renvoie -1.

casse :

Passe en minuscule chaque lettre du mot qui est en majuscules. Si le caractère n'est pas une lettre, il n'est pas modifié.

verif_taille_mot :

Si le mot ne contient pas le caractère '\0', autrement dit si la longueur du mot est supérieure à celle maximale définie (TAILLE_MOT - 1), alors la fonction renvoie 0. Elle renvoie 1 sinon.

verif_caractere :

Vérifie pour l'existence de chaque caractère du mot dans le **tableau_pt**.

recherche :

Fonction itérative qui recherche l'existence d'un mot non vide dans un dictionnaire ***tableau_pt**.
Plus précisément, le mot n'est pas présent si :
- le dictionnaire est un pointeur NULL
- un des caractère n'est pas prévu dans **tableau_pt** (indice_tab(mot[i]) vaut -1)
- le pointeur de nœud pt->T[j] (du caractère mot[i]) est NULL
    exemple : on recherche le mot « vers » mais seul « ver » est présent
    Au niveau du nœud associé au 'r', j vaut 18 pour la lettre 's' et pt→T[j] == NULL car aucun 's' n'est répertorié à la suite
- le mot est présent mais le booléen *fin* du dernier nœud vaut « false ».

Si le mot est présent et le booléen du dernier nœud vaut « true » alors la fonction renvoie « true ».

ajout_dico :

Chaque caractère est ajouté par récursivité.
Pour chaque caractère :
- Si le pointeur de tableau_pt est NULL, on l'initialise.
- Puis si le caractère n'a pas déjà été ajouté (c'est-à-dire si (*pt)→fils>T[i] ==NULL), on crée et initialise le nœud.
- enfin, si c'est le dernier caractère du mot, on passe *fin* à « true ».
- Sinon, on ajoute mot[1:] à la suite.

charge_dico :

Ajoute un à un les mots d'un fichier ouvert dans un dictionnaire passé en paramètre.
Si un mot est trop long (**verif_taille_mot**) ou si un de ses caractères est inconnu (**verif_caractere**), la fonction passe au mot suivant sans l'ajouter.

charge_texte :

Compare tous les mots du texte avec ceux du dictionnaire. Compte ceux qui ne sont pas reconnus et les affiche et renvoie ce nombre.

L'algorithme fonctionne de la sorte.
Il initialise mot à une chaîne de caractère vide et charge le fichier caractère par caractère.
Pour toute la lecture du fichier, tant que le caractère lu ne désigne pas la fin d'un mot, c'est-à-dire qu'il est différent de : . , : ; ! ? ( ) " \n ou 'espace'
alors on l'ajoute à la suite du mot. Si le mot devient trop long (égal à TAILLE_MOT défini à 50 dans **tree.h**), la fonction s'arrête et retourne -1.

Une fois le mot terminé (ou la fin du fichier atteinte), on ajoute le caractère '\0' pour terminer la chaîne de caractère. On passe le mot en minuscules à l'aide de **casse** et on le **recherche** dans le dictionnaire passé en paramètre.

Si le mot n’est pas reconnu, on incrémente le compteur *erreur.*
Enfin, on initialise à nouveau mot à la chaîne vide et on continue la lecture du texte.

affiche_tab (dans annexe.c):
Prend en paramètres un dictionnaire ***tableau_pt** et le mot en cours d’affichage (entrer le mot "" comme premier appel).
Affichage compact par récursivité de tous les mots contenus dans le dictionnaire
Affiche un point-virgule à la suite d’un caractère si celui-ci représente la fin d’un mot.

Soit k le niveau de récursivité. Si le nœud de niveau k possède plusieurs fils, la branche ancêtre est ré-affichée autant de fois que de fils (affj passe à 1 une fois un fils affiché).
La fonction renvoie l’entier *new_aff* désignant si des petits-fils ont été affichés (1 si oui, 0 sinon).

Détaillons avec un exemple : un dictionnaire composé de « ver », « vers » et « verticaux ».
On se place à l’affichage de l’arbre correspondant à « ver; ».
Deux branches filles sont à afficher :
- s : On affiche ‘s;’ à la suite car c’est la fin d’un mot, *motSuiv* devient « ver;s; »
Pas de fils au caractère s, *new_aff* reste à 0.
affj passe à 1 car le fils ‘s’ a été affiché

- t : *motSuiv* redevient *mot* = « ver; »
affj = 1 donc on ré-affiche mot
auquel on affiche le caractère actuel ‘t’ et *motSuiv* = « vert »
la suite du mot « verticaux » est affichée par récursivité et *new_aff*=1 car t possède des fils
On retourne alors à la ligne
etc.

L’affichage sera ainsi :
ver;s;
ver;ticaux;

Remarque : l’affichage insert 4 caractères « aléatoires » entre le premier et le deuxième caractère de chaque mot. Je n’ai pas pris le temps de résoudre ce problème car cette fonction n’est pas utile au cahier des charges.