LAURENT Maëva
MENLET Enzo

Algorithmique des arbres
S4 2022
L2 Informatique

RAPPORT DE PROJET

Correcteur orthographique partie 3 : Arbres BK

Présentation du sujet :

On se propose d'écrire une mini application permettant à l'utilisateur de charger en mémoire un dictionnaire servant de référence et de lister les mots mal orthographiés (relativement au dictionnaire chargé).
Le dictionnaire sera chargé en mémoire dans un arbre. La correction orthographique se fera alors par un parcours de l'arbre.
Dans tout ce devoir, nous n'utiliserons que les lettres minuscules (a,·,z).
Un texte ne possédera pas non plus de ponctuation.
Dans cette partie, nous proposons une méthode pour faire une correction orthographique d'un texte donné. Nous allons utiliser une structure de données adaptée à la recherche approximative de chaînes de caractères identiques : un arbre BK.

Choix de développement :

Comme demandé dans le sujet, nous avons créé deux modules :
Listes.c et **ATR.c**, **Levenshtein.c** et **ArbreBK.c**.

Pour voir les détails des modules **Listes.c**, **ATR.c** et **Levenshtein.c**, se référer au rapport de la partie 1 et 2.

1) Le module **ArbreBK.c** contient les fonctions :

- **NoeudBK * allouer_noeud(char * mot, int distance);**

- **int inserer_dans_ArbreBK(ArbreBK * A, char * mot);**

- **void rechercher_dans_ArbreBK_aux(ArbreBK A, char * mot, Liste * corrections, int * seuil);**

- **Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot);**

- **int est_dans_Arbre_BK(ArbreBK dico, char * mot);**

- **ArbreBK creer_ArbreBK(FILE * dico);**

- **void liberer_ArbreBK(ArbreBK * A);**

- **void afficher_ArbreBK(ArbreBK A, int decalage);**

Elle contient également la structure **NoeudBK** (dans un en-tête **ArbreBK.h**), composée d'un mot, d'une valeur (distance de Levenshtein au père), d'un pointeur vers un fils-gauche et vers un frère droit :

```
typedef struct noeudBK {
    char * mot;
    int valeur;
    struct noeudBK * filsG;
    struct noeudBK * frereD;
} NoeudBK, * ArbreBK;
```

La fonction **allouer_noeud** alloue la place mémoire nécessaire à la création d'un nœudBK et le renvoie.

La fonction **inserer_dans_ArbreBK** permet d'insérer un mot dans notre Arbre en fonction de sa distance de Levenshtein par rapport aux mots déjà insérés dans l'Arbre.

La fonction **rechercher_dans_ArbreBK** utilise la fonction **rechercher_dans_ArbreBK_aux.** Elle initialise un seuil et une Liste vide, et parcours l'arbre pour trouver les mots les plus proches de celui passé en argument, en suivant l'algorithme de recherche dans un arbre BK.

La fonction **est_dans_Arbre_BK** parcours l'Arbre BK, et pour vérifier la présence du mot passé en argument dedans. Si il est dans l'arbre, la fonction renvoie 1, 0 sinon.

creer_ArbreBK lit un fichier et rempli l'arbre avec tous les mots de se fichiers à l'aide de la fonction **inserer_dans_ArbreBK**.

liberer_ArbreBK libère récursivement un par un tous les nœuds de l'arbre BK donné en argument.

Enfin, la fonction **afficher_ArbreBK** permet de visualiser la construction de l'arbre BK avec un affichage décalé en fonction des fils et frères pour chaque nœud.

2) **correcteur_2.c** est notre main.

On crée une liste chaînée **erreurs** vide, et un dictionnaire **dico** composé des mots présent dans le fichier passé en deuxième argument de notre programme, grâce à la fonction **inserer_dans_ArbreBK** et un **fscanf** qui parcours le fichier.
Ensuite on ouvre le fichier passé en premier argument, contenant le texte à corriger. On vérifie ensuite si ce mot est valide ou non grâce à

est_dans_ArbreBK, et si non, on l'ajoute à la liste **erreurs** grâce à **inserer_en_tete**.

Finalement, pour chaque mot présent dans la liste des erreurs, grâce à la fonction **rechercher_dans_ArbreBK**, on parcourt l'arbre du dictionnaire, et on rentre dans la liste **correction** les mots les plus proches du mot erroné.
On affiche ensuite cette liste, puis on la libère, pour le mot erroné suivant.

Enfin on libère l'arbre avec **liberer_ArbreBK**.

On effectue également un mesure du temps d'exécution du programme à l'aide de la bibliothèque **time** et de la fonction **clock()**.

On effectue également cette mesure dans le correcteur 1 pour les comparer.

Au final, on observe que le premier est plus rapide que le second (sur plusieurs essais) :

Correcteur 2 :

```
Temps d'éxécution : 78315.000000
```

Correcteur 1 :

```
Temps d'éxécution : 47142.000000
```

Compilation :

Le projet se compile grâce à un makefile, donc à l'aide de la commande **make.**
Il est également possible de supprimer tous les fichiers liés à la compilation à l'aide de la commande **make clean**.
On peut choisir le correcteur que l'on veut utiliser en rentrant **make correcteur_0**, **make correcteur_1**, ou **make correcteur_2**. La commande de base make est équivalente à **make correcteur_2**.

Documentation utilisateur :

Le rendu est constitué d'un dossier src, contenant toutes les sources du programme, et d'un rapport.
Après compilation, le programme se lance avec une commande de la forme :

./correcteur_2 texte_a_corriger.txt dictionnaire.dico

texte_a_corriger.txt étant le chemin vers le fichier texte à corriger, et **dictionnaire.dico** le chemin vers le fichier texte correspondant au dictionnaire.