DM - Correcteur orthographique

Dans ce DM, on se propose d'implémenter un mini correcteur orthographique.

1 Conditions de rendus

Ce devoir maison est à réaliser en binôme au sein d'un même groupe de TP.

• La première étape sera à rendre sur le e-learning pour le dimanche 20 mars 2022, 23h59. Un fichier `.zip` y sera déposé. Suivant les préceptes du cours de `Perfectionnement C`, le rendu devra comporter les modules `Listes`, `ATR`, ainsi que le programme `correcteur_0` (.c) (sans oublier le `makefile` associé).

• La seconde étape sera à rendre sur le e-learning pour le dimanche 03 avril 2022, 23h59. Un fichier `.zip` y sera déposé. Il contiendra cette fois les modules `Listes`, `ATR`, `Levenshtein`, ainsi que les programmes `correcteur_0/1` (.c) (sans oublier le `makefile` associé).

• Enfin, la troisième étape sera à rendre sur le e-learning pour le dimanche 24 avril 2022, 23h59. Un fichier `.zip` y sera déposé. Il contiendra cette fois les modules `Listes`, `ATR`, `Levenshtein` et `ArbreBK`, ainsi que les programmes `correcteur_0/1/2` (.c), ainsi que les programmes `correcteur_0/1` (.c) (sans oublier le `makefile` associé).

Un rapport expliquant les choix de développements et le travail en groupe accompagnera aussi chaque rendu. Pour le troisième rendu, il contiendra une comparaison en temps des deux méthodes pour proposer de la correction orthographique.

2 Descriptif du devoir

On se propose d'écrire une mini application permettant à l'utilisateur :

— de charger en mémoire un dictionnaire servant de référence ;
— de lister les mots mal orthographié (relativement au dictionnaire chargé) ;
— pour chaque mot mal orthographié, de proposer des corrections orthographiques raisonnables ou d'ajouter le mot dans le dictionnaire.

Le dictionnaire sera chargé en mémoire dans un arbre. La correction orthographique se fera alors par un parcours de l'arbre.

Le premier rendu consiste en la détermination des mots mal orthographié en utilisation un arbre préfixe. Il s'agit d'une étape préliminaire pour le second rendu (*étape 2*) qui propose une méthode par force brute en parcourant l'intégralité de l'arbre pour faire de la correction orthographique.. Le troisième rendu (*étape 3*) propose une autre méthode abordée en cours : les arbres BK. Il s'agit, avec cette structure de données de faire à la fois la détection des mots mal orthographiés et les propositions de corrections.

Dans tout ce devoir, nous n'utiliserons que les lettres minuscules `a`, ·, `z`. Un texte ne possèdera pas non plus de ponctuation.

3 Etape 1 : recherche des mots mal orthographié

Dans ce sujet, on appelle *d*ictionnaire un ensemble de mots qui, par convention, sont correctement orthographiés.

En machine, un dictionnaire sera toujours sauvegardé sous forme d'un fichier texte dont l'extension est `.dico`. Il contiendra toujours un et un seul mot par ligne. Les mots n'y sont pas nécessairement classés par ordre alphabétique.

En mémoire, l'ensemble des mots du dictionnaire sera contenu dans un *arbre lexicographique* implémenté sous forme d'`arbre ternaire de recherche` (ATR, en abrégé).

La recherche des mots mal orthographié dans un texte est un algorithme relativement simple :

Algorithm 1 Recherche de mots mal orthographiés dans un texte.

Require : un dictionnaire `dico`, ie un fichier texte contenant des mots (un par ligne) correctement orthographié
Require : un fichier texte `a_corriger`

```
1 : Créer une liste chainée erreurs vide
2 : for chaque mot m présent dans a_corriger do
3 :   if m n'est pas dans dico then
4 :     insérer m en tête de la liste chainée erreurs
5 :   end if
6 : end for
7 : return erreurs
```

Travail à faire :

- Dans le module `Listes.c`, implémenter les fonctions suivantes permettant de manipuler des listes chainées de chaines de caractères :
  - `Cellule * allouer_Cellule(char * mot);`
  - `int inserer_en_tete(Liste * L, char * mot);`
  - `void liberer_Liste(Liste * L);`
  - `void afficher_Liste(Liste L);`

  Les ajouts se feront en tête de liste. Un fichier `Listes.h` sera disponible.
- Dans le module `ATR.c`, implémenter la structure de données ATR, ainsi que les fonctionnalités suivantes :
  - `ATR creer_ATR_vide()`
  - `void liberer_ATR(ATR * A)`
  - `int inserer_dans_ATR(ATR * A, char * mot)`
  - `void supprimer_dans_ATR(ATR * A, char * mot)`
  - `void afficher_ATR(ATR A)`

  Seules des minuscules seront stockées dans l'ATR. L'affichage permet d'afficher tous les mots contenu dans l'ATR, un par ligne, par ordre alphabétique.

  Un fichier `ATR.h` sera disponible.
- Créer un exécutable `correcteur_0` issue de la compilation du fichier `correcteur_0.c` qui implémente l'algorithme 1 de détection de mots mal orthographiés.

  Celui-ci prendra deux arguments, le chemin du fichier texte à corriger, suivi du dictionnaire à prendre en compte :

  ./correcteur_0 texte.txt dictionnaire.dico

4 Etape 2 : Correction orthographique par force brute

On appelle *distance de Levenshtein* entre deux chaînes de caractères `s` et `t`, et on note $\mathcal{L}(\texttt{s},\texttt{t})$, le nombre minimal d'étapes pour transformer `s` en `t` en effectuant les opérations suivantes :

- substitution de deux caractères ;
- insertion d'un caractère ;
- suppression d'un caractère.

Etant donnée une chaine de caractères `s`, on note `l(s)` sa longueur et `s[i]` son `i`-ième caractère. `s'` est aussi la chaine de caractères privée de son premier élément. Avec ses notations, la distance de Levenshtein se calcule récursivement comme suit :

$$\mathcal{L}(\texttt{s},\texttt{t}) = \begin{cases} \max\big(l(\texttt{s}), l(\texttt{t})\big) & \text{, si } \min\big(l(\texttt{s}), l(\texttt{t})\big) = 0 \\ \mathcal{L}(\texttt{s'},\texttt{t'}) & \text{, si } \texttt{s}[0] = \texttt{t}[0] \\ 1 + \min\big(\mathcal{L}(\texttt{s'},\texttt{t}), \mathcal{L}(\texttt{s},\texttt{t'}), \mathcal{L}(\texttt{s'},\texttt{t'})\big) & \text{, sinon} \end{cases} \tag{1}$$

Par exemple, $\mathcal{L}(\texttt{"dure"},\texttt{"gare"}) = 2$ en modifiant `'d'` en `'g'`, puis `'u'` en `'a'` et $\mathcal{L}(\texttt{"avait"},\texttt{"saint"}) \geq 3$ grâce à la suite de transformation de mots : `"avait"`$\longrightarrow$`"avaint"`$\longrightarrow$`"vaint"`$\longrightarrow$`"saint"` .

La distance de Levenshtein est une *vraie* distance, au sens mathématique du terme. En particulier, une inégalité triangulaire y est associée : si `u`, `v` et `w` sont trois mots, alors : $\mathcal{L}(\texttt{u},\texttt{w}) \leq \mathcal{L}(\texttt{u},\texttt{v}) + \mathcal{L}(\texttt{v},\texttt{w})$.

Remarque : La version récursive naïve n'est pas efficace, car elle fait calculer plusieurs fois la même distance avant d'obtenir le résultat souhaité. Pour une implémentation raisonnable, il est possible d'utiliser l'algorithme itératif donné sur la page wikipédia Distance de Levenshtein.
Il s'agir d'utiliser un tableau à deux dimensions que l'on allouera sur le tas et qu'on libérera une fois le calcul fini.

Pour chaque mot `m` mal orthographié d'un texte, on souhaite désormais faire des suggestions de correction à l'utilisateur. L'idée est de rechercher l'ensemble des mots `m'` du dictionnaire `dico.dico` qui sont à distance de Levenshtein minimale du mot `m`.

La méthode par force brute consiste parcourir l'intégralité du dictionnaire, *i.e.* de l'arbre codant le dictionnaire :

Algorithm 2 Correction orthographique de mots mal orthographiés dans un texte par force brute

Require : un dictionnaire `dico`, ie un fichier texte contenant des mots (un par ligne) correctement orthographié
Require : une chaine de caractères `mot`

```
1 : Créer une liste chainée correction vide
2 : d_min = +∞
3 : for chaque mot m présent dans dico do
4 :     d = L(m, mot)
5 :     if d ≤ d_min then
6 :         if d < d_min then
7 :             d_min = d
8 :             vider la liste corrections
9 :         end if
10 :        insérer m en tête de la liste chainée corrections
11 :    end if
12 : end for
13 : return corrections
```

Travail à faire :

- Dans le module `Levenshtein.c`, implémenter la fonction `int Levenshtein(char * un, char * deux)` qui calcule la distance de Levenshtein entre les mots `un` et `deux`.

  Un fichier `Levenshtein.h` sera disponible.
- Créer un exécutable `correcteur_1` issue de la compilation du fichier `correcteur_1.c` qui :
  - implémente l'algorithme 1 de détection de mots mal orthographiés ;
  - propose des corrections obtenues par force brute (algorithme 2) pour chaque mot mal orthographié.

  Celui-ci prendra deux arguments, le chemin du fichier texte à corriger, suivi du dictionnaire à prendre en compte :

  ./correcteur_1 texte.txt dictionnaire.dico

5 Etape 3 : Arbres BK

Pour chaque mot `m` mal orthographié d'un texte, on souhaite toujours faire des suggestions de correction à l'utilisateur. Cette fois, nous allons utiliser une structure de données adaptée à la recherche approximative de chaines de caractères identiques : un *arbre BK*, (d'après leurs inventeurs Burkhard and Keller).

5.1 Rappels et exemples

Etant donnée un dictionnaire $\mathcal{D}$ de mots, un arbre n-aire A ayant des nœuds étiquettés par des éléments de $\mathcal{D}$ est appelé un *arbre BK* lorsque :

1. A est l'arbre vide ou A est un arbre de racine r ayant des sous-arbres $A_1, \cdots, A_n$ qui sont eux-même des arbres BK.
2. Si A est non vide, les arêtes de A à un sous-arbre A_k sont valuée par la distance de Levenshtein entre le mot contenu à la racine et les mots contenant dans le sous-arbre A_k.

```
Lexique = {une, fois, foie, qui, ville,
           foix, elle, foi, est, que}
```

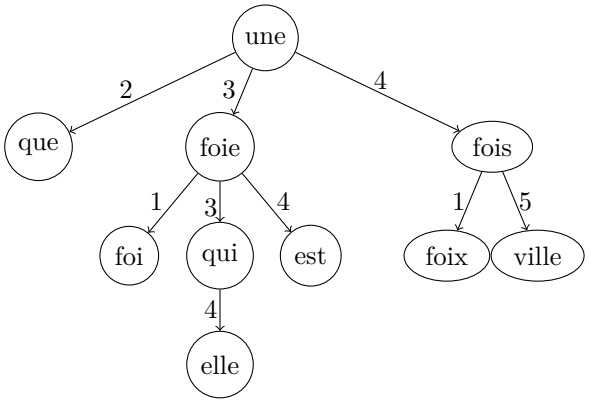

FIGURE 1 – Un lexique et un arbre BK associé

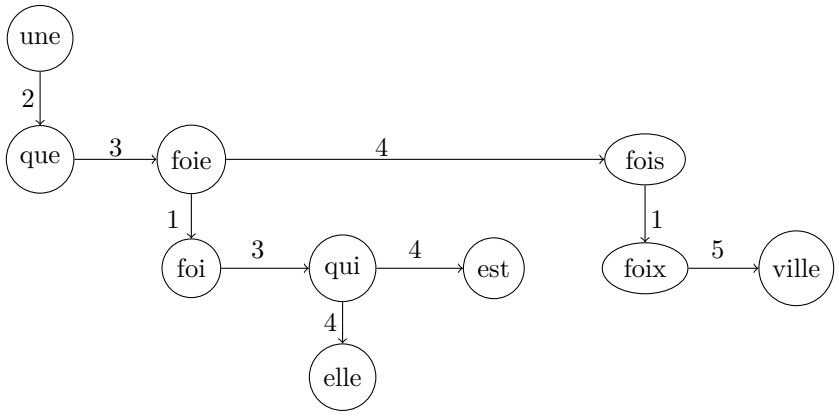

FIGURE 2 – L'arbre BK de la figure 1 sous forme d'arbre fils gauche / frères droits

5.2 Implémentation

En tant qu'arbre n-aire, un arbreBK sera représenté sous forme d'arbre binaire en utilisant la représentation fils gauches, frères droits. L'arbre BK de la Figure 1 sera donc représenté en machine par l'arbre représenté

Figure 2.

Ainsi, on utilisera la structure suivante :

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

L'attribut `valeur` représentera la valuation de l'arête liant le nœud courant à son parent. Pour le cas exceptionnel de la racine (qui n'a pas de parent), cette valeur sera mise à 0.

5.3 Travail à faire

- Dans le module `ArbreBK.c`, implémenter la structure de données `ArbreBK`, ainsi que les fonctionnalités suivantes :
  - `int inserer_dans_ArbreBK(ArbreBK * A, char * mot);`
  - `Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot);`
  - `ArbreBK creer_ArbreBK(File * dico);`
  - `void liberer_ArbreBK(ArbreBK * A);`
  - `void afficher_ArbreBK(ArbreBK A);`

  Un fichier `ArbreBK.h` sera disponible.
- Créer un exécutable `correcteur_2` issue de la compilation du fichier `correcteur_2.c` qui :
  - implémente l'algorithme 1 de détection de mots mal orthographiés en parcourant un arbre BK ;
  - propose des corrections pour chaque mot mal orthographié.

  Celui-ci prendra deux arguments, le chemin du fichier texte à corriger, suivi du dictionnaire à prendre en compte :

  ./correcteur_1 texte.txt dictionnaire.dico

6 Améliorations : laissez libre court à votre imagination

Des améliorations pourront être apportées une fois l'exécutable `correcteur_2` fonctionnel.
Celles-ci ne seront prises en compte dans la notation que si celui-ci a le comportement attendu.

7 Exemples

Pour corriger les exécutables `correcteur_0` et `correcteur_1`, un test similaire à celui produit dans les paragraphes suivant sera mis en place.

7.1 L'exécutable `correcteur_0`

En guise d'exemple, utilisons, par exemple, le dictionnaire suivant :

apres	classe	debarquee	nuit	ses	une
avait	de	freres	ou	sur	venue
avec	denise	gare	passee	train	wagon
banquette	deux	la	pied	troisieme	a
cherbourg	dure	lazare	saint	un	etait

FIGURE 3 – Le contenu du dictionnaire `dico_1.dico`

et cherchons à corriger le texte suivant :

```
deniz etait venu a pieds de la gare saint laz ou un train deux cherbourg l avait debarquee.
```

FIGURE 4 – Le contenu du fichier texte `texte_0.txt`

En utilisant le fichier texte donné à la figure 4 avec le dictionnaire de la figure 3, l'exécutable `correcteur_0` aura pour sortie la figure 5

```
bouillot $ ./correcteur_0 texte_0.txt dico_1.dico
Mot(s) mal orthographié(s) :
deniz
venu
pieds
laz
l
```

FIGURE 5 – Un exemple d'utilisation et de rendu de l'exécutable `correcteur_0`

7.2 L'exécutable `correcteur_2`

En guise de second exemple, utilisons, par exemple, le dictionnaire suivant :

dans	marchande	une	il	vends	je	vendait
du	premiere	fois	elle	foix	et	derniere
foi	de	foie	ma	se	que	
ville	la	qui	dit	est	etait	

FIGURE 6 – Le contenu du dictionnaire `dico_2.dico`

qui donne l'arbre BK de la figure 9, si l'on y a inséré les mots du dictionnaire précédent par colonne. L'affichage de cet arbre par la fonction `afficher_ArbreBK` pourra être sous la forme obtenu à la figure 10, ou bien être obtenu en utilisant `GraphViz` et le langage `.dot`.

```
Il etais une foit,          Dans la vil de Faix.        Et la derniere fois,
Une marchande de doigts,    Elle se dit ma foi,         Que je vends du foie,
Qui vendait du foie,        C est la premiere fois      Dans la ville de Faix.
```

FIGURE 7 – Le contenu du fichier texte `texte_1.txt`

Cherchons à corriger le texte suivant :

En utilisant le fichier texte donné à la figure 7 avec le dictionnaire de la figure 6, l'exécutable `correcteur_1` aura pour sortie la figure 8

```
bouillot $ ./correcteur_1 texte_1.txt dico_2.dico
Mot mal orthographié : etais
Proposition(s) : etait
Mot mal orthographié : foit
Proposition(s) : foix, foie, foi, fois
Mot mal orthographié : doigt
Proposition(s) : dit
Mot mal orthographié : vil
Proposition(s) : il
Mot mal orthographié : faix
Proposition(s) : foix
Mot mal orthographié : c
Proposition(s) : et, je, se, il, ma, la, de, du
Mot mal orthographié : faix
Proposition(s) : foix
```

FIGURE 8 – Un exemple d'utilisation et de rendu de l'exécutable `correcteur_0`

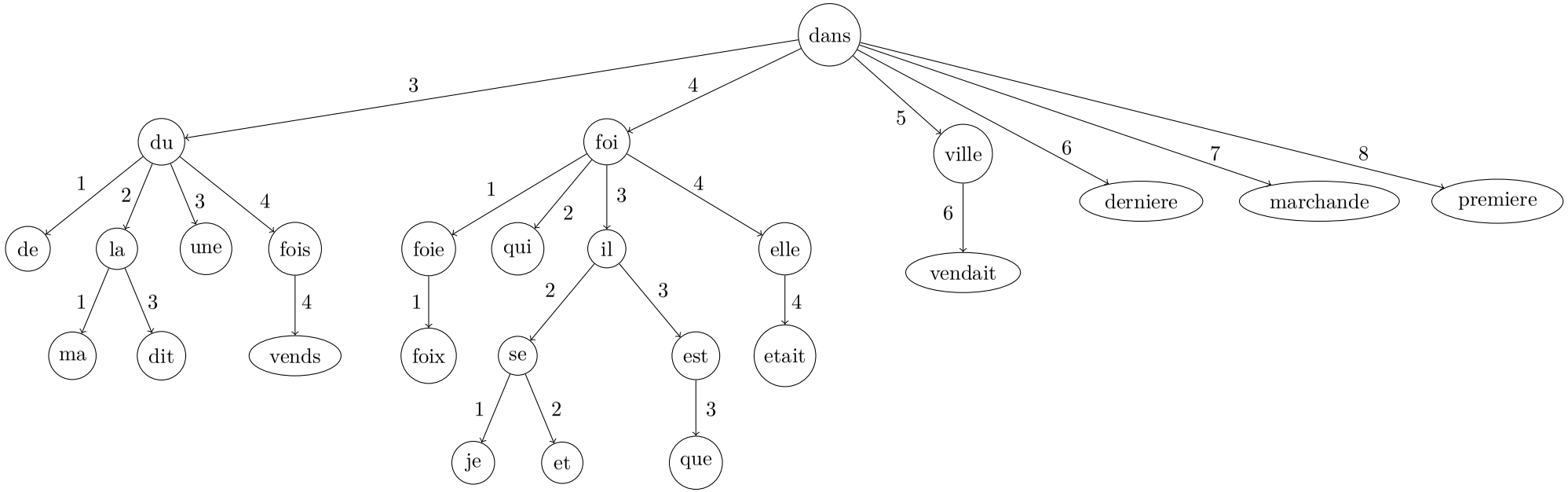

FIGURE 9 – L'arbre BK associé au dictionnaire précédent

```
dans
|--3--> du
        |--1--> de
        |--2--> la
                |--1--> ma
                |--3--> dit
        |--3--> une
        |--4--> fois
                |--4--> vends
|--4--> foi
        |--1--> foie
                |--1--> foix
        |--2--> qui
        |--3--> il
                |--2--> se
                        |--1--> je
                        |--2--> et
                |--3--> est
                        |--3--> que
        |--4--> elle
                |--4--> etait
|--5--> ville
        |--6--> vendait
|--6--> derniere
|--7--> marchande
|--8--> premiere
```

FIGURE 10 – Affichage textuel de l'arbre BK de la figure 9