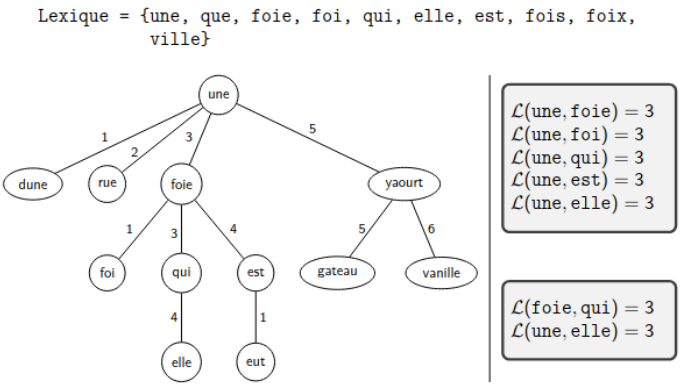
Ilhan ACHERCHOUR / Hakim AOUDIA
24/04/2022

Semestre 4 - 2ème année de Licence

Algorithmique des arbres

Rapport de l'étape 3 - Final



Exemple d'un arbre BK. (cf. le cours)

Sommaire :

Introduction

Pour cette troisième étape nous avons réalisé l'implémentation de l'arbre BK. Un arbre BK permet de stocker des mots en fonction de leur distance. Leur distance est calculée grâce à la fonction de Levenshtein.

Cette représentation des arbres nous permet une plus grande rapidité qu'un arbre ternaire de recherche. Cet arbre BK permet comme un ATR de rechercher des mots similaires ce qui va nous permettre de proposer des corrections à des mots mal orthographiés.

Compilation

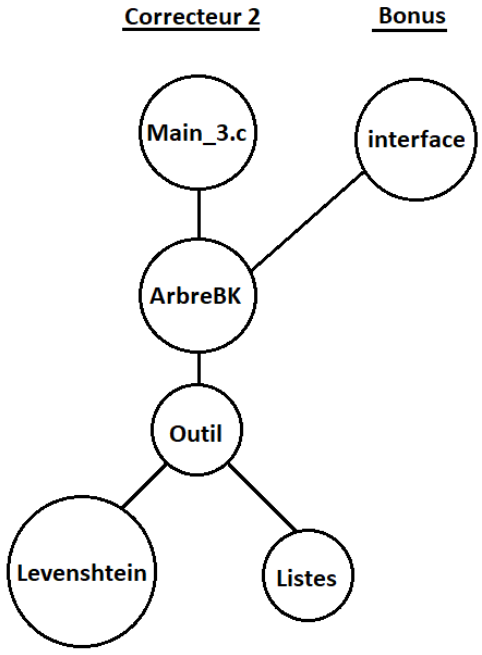
Nous avons amélioré le makefile pour permettre de compiler cette étape plus facilement.

La commande pour compiler cette 3e étape du projet est la suivante:

```
makefile correcteur_2
```

Pour lancer le programme on écrit :

```
./correcteur_2 FichierACorriger Dictionnaire
```



Voici un schéma permettant de voir les modules de notre projet :

Difficultés

Arbre BK

Nous n'avons pas rencontré beaucoup de difficultés à l'implémentation mais surtout sur la compréhension de la structure des arbres BK. Savoir que les frères droit avait les même parent que le fils gauche et comprendre que même si la structure contient deux pointeurs cela n'en fait pas un arbre binaire mais bien un arbre n-aire.

Nous avons donc commencé cette phase par une étape de compréhension d'un arbre BK avant de réellement programmer. Cela s'applique notamment pour la fonction ***rechercher_dans_ArbreBK()*** car nous avons attentivement regardé l'exemple du cours pour ensuite mieux comprendre l'algorithme.

Retour sur la phase 2

Nous n'avions pas réussi à faire des propositions de corrections en parcourant l'ATR lors de la phase précédente, c'est pourquoi nous avons opté pour le parcours du fichier directement. Ce qui nous donnait de bon résultats mais ne respectait pas toujours la consigne.

Nous sommes donc revenu sur notre travail et avons découvert dans la fonction **insere_en_tete(Liste * L, char * mot),** qui s'occupe d'insérer une cellule en début de liste chaîné, que nous ne prenions pas le temps d'allouer de l'espace pour le champs mot de la cellule.

Pour remplir le champs mots nous faisions simplement une affectation d'adresse ce qui faisait que ce champs devenait une copie profonde du mot à insérer.

```
Cellule *allouer_Cellule(char * mot){
    Cellule *lst = (Cellule *) malloc(sizeof(Cellule));
    if(lst != NULL){
        lst->mot = mot;
        lst->suivant = NULL;
    }
    return lst;
}
```

Il suffisait d'allouer de l'espace supplémentaire pour ne plus avoir de problèmes.

```c
Cellule *allouer_Cellule(char * mot){
    Cellule *lst = (Cellule *) malloc(sizeof(Cellule));
    if(lst != NULL){
        lst->mot = (char *) malloc(sizeof(char) * 256);
        strcpy(lst->mot, mot);
        lst->suivant = NULL;
    }
    return lst;
}
```

Cette erreur d'inattention est surtout la conséquence d'avoir voulu aller trop vite.

Documentation technique

L'algorithme suit celui donné dans l'énoncé c'est-à-dire, construire une liste chaînée erreur contenant les mots mal orthographiés puis par une recherche dans l'arbre BK, propose des corrections adaptées.

On commence par initialiser un arbre BK que nous remplissons à l'aide d'un dictionnaire donné en paramètre. Nous avons une fonction d'insertion qui va nous aider à ajouter des mots dans un arbre BK. Nous pourrons alors rechercher chaque mots du texte à corriger dans notre arbre. Si les mots ne sont pas dans notre arbre c'est donc qu'ils sont mal orthographiés alors nous les ajoutons dans une liste d'erreurs.

Ensuite pour chaque mots dans la liste d'erreurs nous cherchons les mots similaires à l'aide la distance de levenshtein dans notre arbre pour proposer une correction.

Documentation utilisateur

L'utilisateur va comme pour l'étape deux lancer son programme à l'aide d'une commande.

Comme vu lors du chapitre **Compilation** il faut qu'il donne un **texte à corriger** avec un **dictionnaire**.

Dans notre projet nous avons ajouté un dictionnaire comportant tous les mots de la langue française.

Nous pouvons l'utiliser comme ceci :

```
./correcteur_2 a_corriger_1.txt dico_3.dico
```

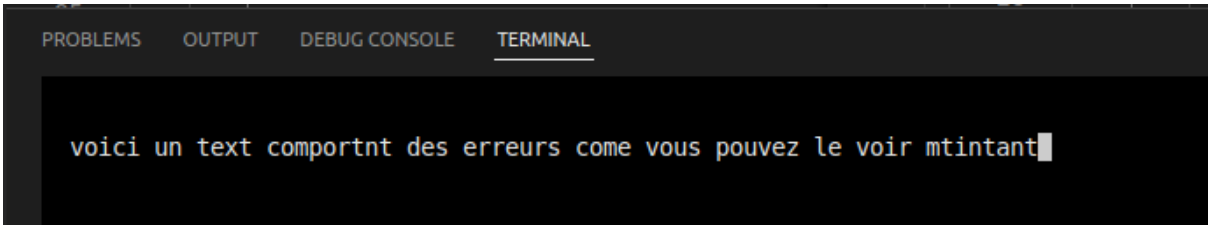
```
hakim@hakim-ThinkPad-X250:~/Documents/L2/S4/ALGO DES ARBRES/DM - Arbre ternaire de recherche/3$ ./correcteur_2 a_corriger_1.txt dico_3.dico
Mot mal orthographié : c
Propositions : y, x, v, s, i, ca, ca, ca, ce, ci, a
Mot mal orthographié : faix
Propositions : paix, fait, faim, faux, fais
Mot mal orthographié : foie
Propositions : oie, foi, voie, soie, noie, joie, fois, folie, foire
Mot mal orthographié : foit
Propositions : fit, fait, foi, fout, voit, toit, soit, font, doit, fois, fort, boit

Après parcours d'un ArbreBK dans le fichier 'dico_3.dico'
Temps écoulé : 0.008661
```

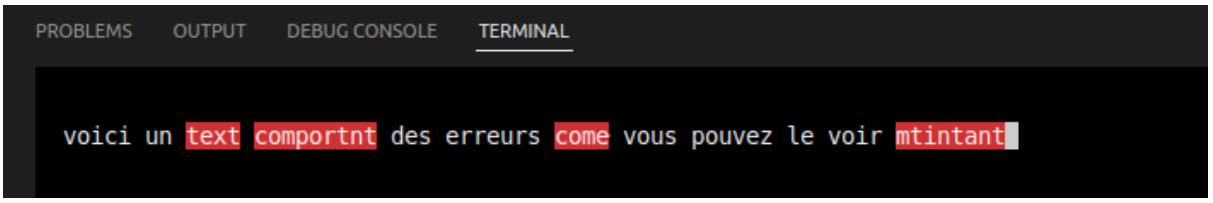
Bonus

Nous avons voulu réaliser une amélioration bonus après avoir fini la 3ème phase.

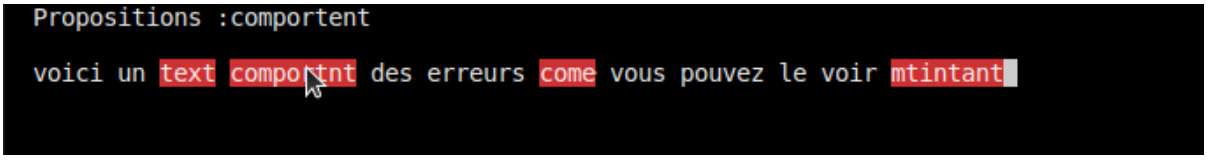
Notre bonus consiste à laisser l'utilisateur écrire un texte.



Il clique alors sur **Entrée** pour finaliser. Le programme va **détecter les mots mal orthographiés** selon le dictionnaire donné en paramètre puis afficher ces mêmes mots sur fond rouge pour les identifier plus facilement.



L'utilisateur peut ensuite cliquer sur ces mêmes mots pour afficher une proposition de correction.



Difficulté lors de la phase bonus

On lance le programme bonus à l'aide la commande

```
./application liste_francais.txt
```

Le programme fonctionne parfaitement jusqu'à la phase de détection des erreurs. Ce qui nous a posé problème est la partie **"gestion des cliques"**.

En effet lorsque nous cliquons sur un mot afficher en rouge il y a une proposition de mot qui s'affiche cependant cela ne **marche qu'une seule fois**. Lors du deuxième clique nous avons une Erreur segmentation. Nous avons cherché sans trouver d'où provient l'erreur. Nous avons donc dû arrêter cette phase bonus qui pourtant nous trouvions très utile et agréable à réaliser.

Comparaison des recherches

Pour réaliser cette étape nous avons mesuré le temps de recherche d'un arbre BK et d'un ATR à l'aide d'un grand dictionnaire comme celui de la langue française. Nous avons utilsié les mêmes textes à corriger avec le même dictionnaire.

Voici le résultat :

Arbre BK

./correcteur 2 a corriger 0.txt liste_francais.txt

```
Après parcours d'un ArbreBK dans le fichier 'liste_francais.txt'
Temps écoulé : 0.072523
```

ATR

./correcteur_2 a_corriger_0.txt liste_francais.txt

```
Après parcours d'un ATR dans le fichier 'liste_francais.txt'
Temps écoulé : 0.163711
```

On voit que l'arbre BK est **deux fois plus rapide** que l'ATR pour un même fichier à corriger à l'aide du même dictionnaire.

Conclusion

Nous avons plutôt apprécié d'implémenter cet arbre BK car cela ne nous a pas posé énormément de problème. La répartition a toujours été fluide étant donné que nous sommes appelés sur discord pour réaliser le projet. Cette phase nous montre qu'il y a toujours des moyens d'améliorer la rapidité de parcours d'un arbre permettant alors une meilleure utilisation pour un utilisateur en condition réel (comme lors d'une proposition orthographique sur téléphone).