Rapport de Projet

Rendu 3

Algorithmique des Arbres

Gaël Paquette
Hervé Nguyen
22 avril 2022
L2 Informatique

Table de matières

Introduction

Ce dernier rendu consiste à développer une implémentation des Arbres BK. Cette structure de donnée étant plus adaptée dans le cadre de la recherche de corrections. Un Arbre BK sera alors représenté sous la forme d'un arbre binaire avec des chaînages « Fils Gauche » et « Frère Droit » pour simuler un Arbre n-aire.

Pour ce second rendu , nous avons repris notre rendu 2 et puis nous avons ajouté un nouveau module ArbreBK.

Ce module ArbreBK permettra d'employer une correction orthographique en se passant de la méthode brute force en recherchant (via Levenshtein) avec les ATR.

Module ArbreBK

Ce module a une dépendance avec le module Levenshtein pour les calculs de valeurs de noeuds et aussi une dépendance avec le module Listes pour la fonction de recherche dans un Arbre BK qui renvoie une liste.

Comme pour les ATR, la subtilité ici est autour de la recherche et l'insertion. L'insertion consiste à calculer au fur et à mesure la distance de Levenshtein

entre le mot à insérer et le mot du noeud père, après un calcul on cherche à insérer au fils gauche du père actuel.

Soit le fils gauche n'existe pas donc on crée un noeud sur cette position sinon, on se déplace suivant le fils gauche si d(Mot à ajouter, Mot du père) == d(Mot du noeud actuel, Mot du père).

Dans le cas contraire on se déplace suivant le frère droit si d(Mot à ajouter, Mot du père) != d(Mot du noeud actuel, Mot du père).

On répète ces instructions jusqu'à que l'on se retrouve à un noeud vide ou bien si le mot à insérer est déjà présent.

L'affichage sur la console consiste simplement à bien reconnaître les profondeurs de chaque noeud et espacer correctement les noeud. Un frère droit a la même profondeur que le noeud courant, tandis que son fils gauche est à une profondeur « + 1 » à celui-ci.

Un affichage de l'Arbre BK sur pdf avec dot est aussi disponible si le système où le programme tourne a le paquet graphviz installé.

Recherche

La recherche (de correction) est dans ce rendu très important car c'est un des points qui permettent au programme d'avoir une complexité bien moindre que le parcours naïf avec des ATR.

On s'est servi d'une propriété du cours :

Propriété :

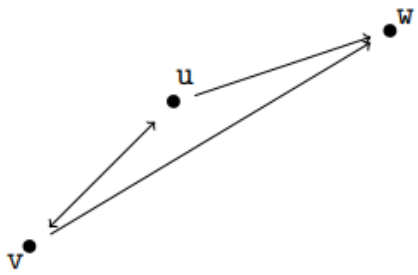
Soit v_i la valuation du sous-arbre $\mathtt{A}_i$ vérifiant $|d - v_i| > S_0$.
Alors, le sous-arbre $\mathtt{A}_i$ ne contient pas de mots au plus à distance S_0 de $\mathtt{m}$.

Qui provient de cette inégalité triangulaire :

Propriété : *Seconde inégalité triangulaire*

Soit u, v et w trois chaines de caractères.
Alors, on a :

$$\left|\mathcal{L}(u,v) - \mathcal{L}(u,w)\right| \leq \mathcal{L}(v,w)\ . \qquad (2)$$



Cela nous permet de ne pas parcourir des sous-arbres avec des mots qui ont peu d'intérêt (Distance de Levenshtein trop important avec le mot recherché).

Nous avons pris peu de liberté et nous avons implémenté la recherche décrite par le cours :

Principe détaillé :

recherche(ArbreBK A, char * m, Liste * corrections, int * seuil) :

- Si l'arbre est vide, on ne fait rien
- Sinon, soit u la racine de l'arbre
  - $d = \mathcal{L}(u, m)$
  - Si $d = *seuil$, ajouter r en tête de la liste chainée *corrections
  - Si $d < *seuil$:
    vider *corrections et y ajouter ensuite r
    modifier *seuil en le fixant à d
  - Pour chaque enfant e de r, relié par une arête de valuation v_i :
    Si $|v_i - d| \leq$ *seuil :
    recherche(e, m, corrections, *seuil)

Jusque là nous avons décris un algorithme de recherche de correction, mais nous avons également écris une fonction de recherche qui renvoie 0 ou 1 selon la présence d'un mot dans l'Arbre BK. Cette fonction de recherche reprends le même parcours de l'insertion.

Évaluation des performances

Nous avons temporairement modifié nos programmes du précédent rendu et de celui-ci pour évaluer les performances entre l'Arbre BK et l'ATR.

Nous avons mesuré le temps d'exécution des deux programmes avec gettimeofday() de <sys/time.h> pour calculer l'intervalle au début et à la fin des exécutions. Il y a eu 2 test, le premier avec un petit texte puis le deuxième avec un texte plus volumineux.

Pour le premier test, nous avons utilisé le texte a_corriger_1.txt avec le dictionnaire dico_3.dico :

- Le programme correcteur_1 a en moyenne 110ms en temps d'exécution.
- Le programme correcteur_2 a en moyenne 180ms en temps d'exécution.

Le programme utilisant les ATR s'est exécuté 70ms plus rapidement que celui avec les Arbre BK, cependant ce text était assez court donc on ne peut pas encore juger si les ATR sont supérieur au niveau de la complexité.

Pour le second test, nous avons utilisé le texte a_corriger_2.txt avec le dictionnaire dico_3.dico, ce texte est un extrait du Chapitre VI « Jean Valjean » Des Misérables de Victor Hugo (accessible sur ce lien) :

- Le programme correcteur_1 a en moyenne 2100ms en temps d'exécution.
- Le programme correcteur_2 a en moyenne 900ms en temps d'exécution.

On remarque alors que les ArbreBK commencent à avoir un avantage vis-à-vis des ATR. Avec le programme avec les ATR qui s'exécute 2 fois plus lentement que les ArbreBK.

Nous avons triplé la taille de ce même texte (copier-coller) pour confirmer l'avantage en complexité de correcteur_2 et nous avons eu :

- correcteur_1 avec 6000ms en temps d'exécution.
- correcteur_2 avec 2000ms en temps d'exécution.

On peut alors conjecturer avec un confiance relativement haute que les ArbreBK sont beaucoup plus adapté que les ATR en terme de complexité pour les correction orthographiques, sur ce dernier essai correcteur_2 s'est exécuté avec 1/3 du temps d'exécution de correcteur_1. L'écart semble bien se creuser selon la taille du texte.

Structuration du rendu

Le rendu contient les fichiers en-têtes des modules dans le sous répertoire « ../include »

Les fichiers sources sont dans le sous répertoire « ../src ».

Le makefile, les fichiers régénérables, le rapport se trouvent au répertoire courant du rendu.

Voici la liste des modules :

- ATR (Permet d'avoir une implémentation d'un dictionnaire)
- Listes (Implémentation de listes chainées simples)
- Levenshtein (Permet de calculer des distances de Levenshtein)
- ArbreBK (Permet de manipuler des Arbres BK)

Voici les fichiers « main » :

- correcteur_0.c (Fichier main du rendu 1 pour le programme correcteur_0)
- correcteur_1.c (Fichier main du rendu 2 pour le programme correcteur_1, affiche des suggestions de correction en plus du comportement habituel de correcteur_0)
- correcteur_2.c (Fichier main du rendu 3 pour le programme correcteur_2)

Deux fichiers textes sont fournis :

- a_corriger_0.txt
- a_corriger_1.txt
- a_corriger_2.txt

Trois fichiers dictionnaires sont fournis :

- dico_1.dico
- dico_2.dico
- dico_3.dico

Organisation du travail

Les communications et échanges de fichiers ont étés sur Discord.

Comment compiler les programmes ?

Nous avons fourni un makefile qui permet de compiler les deux programmes de ce rendu.

Il suffit de faire :

```
make
```

Pour supprimer les fichiers objets :

```
make clean
```

Pour supprimer la totalité des fichiers régénérables :

```
make mrproper
```

Comment utiliser les programmes ?

Le premier programme est correcteur_0.

Celui-ci permet d'afficher les erreurs contenu dans le fichier texte passé en argument selon le dictionnaire aussi passé en argument.

Pour le lancer, il suffit de faire :

./correcteur_0 [fichier_texte] [fichier_dictionnaire]

Le second affiche les erreurs et aussi des suggestions de correction pour chaque mot erronés.

Pour le lancer il suffit de faire :

./correcteur_1 [fichier_texte] [fichier_dictionnaire]

Le troisième affiche les erreurs et aussi des suggestions de correction pour chaque mot erronés mais en utilisant des ArbresBK pour faire des recherches approximatives.

Pour lancer la correction il suffit de faire :

./correcteur_2 [fichier_texte] [fichier_dictionnaire]

Il est possible d'afficher l'arbre BK avec deux modes,

Pour afficher en pdf avec evince + dot (graphviz) :

./correcteur_2 -g [fichier_dictionnaire]

Pour afficher sur le terminal :

./correcteur_2 -a [fichier_dictionnaire]

Problèmes rencontrés

Dans ce rendu, nous avons rencontrés de nombreux problème comme par exemple des erreurs de segmentations, la plupart sont dû à des mauvaises manipulation de pointeurs. Nous avons en général, utilisé lldb / gdb pour retrouver les lignes des erreurs de segmentations.

Un autre problème rencontré, était que le programme pouvait avoir un segmentation fault sur MacOS ARM alors que le programme fonctionnait sur WSL2 (Ubuntu).

Le problème était visiblement lié au fait que l'on faisait deux fopen sur un même fichier sur deux pointeurs différents, nous avons estimé que le comportement sur les manipulations de pointeurs de fichiers entre les deux systèmes sont différents. Pour corriger cela, nous avons simplement gardé un seul pointeur et utilisé rewind() pour remettre à 0 la tête de lecture sur le fichier.

Addendum

Nous avons renommé les fichiers Main_0.c / Main_1.c / Main_2.c vers correcteur_0.c / correcteur_1.c / correcteur_2.c pour mieux suivre le sujet contrairement aux deux derniers rendus.

Dans le sujet, la détection de mots erronés pour ce 3ème programme aurait pu se faire avec un ATR ou bien un Arbre BK. Étant donné que c'était assez ambiguë nous avons décidé que ce 3ème programme (correcteur_2) utilisera exclusivement des Arbres BK pour l'implémentation de dictionnaires.

Nous avons choisi le standard C99 seulement parce-que nous avons voulu utiliser des tableaux statique de taille défini par variable. Sinon, hormis cela le programme n'a aucun warning en compilant avec -ansi (C89).

Conclusion

Comparé aux autres rendu, celui-ci n'était pas nécessairement plus difficile. Il était nécéssaire de d'implémenter correctement les algorithmes décris en cours et de ne pas se perdre dans les utilisations de pointeurs.

Il y a eu un peu d'ambiguïtés dans le sujet pour cette 3 étape mais ce n'était pas très gênant.

L'intérêt des structures de données « complexe » dans les programme était très apparent ici. L'objectif étant de limiter un maximum la complexité des algorithmes utilisés.