Projet Tries - ALGAV
Master STL

Clément Chouteau

22 décembre 2017

1 Structure du projet

Le but du projet est d'implémenter une bibliothèque logicielle permettant d'utiliser une structure de donnée de type dictionnaire, et d'effectuer principalement des opérations d'insertion, de rechercher et de suppression de mots.
J'ai choisi le langage C++, et le projet comporte 1500 lignes de code. Les structures de données sont génériques et supportent tout type de caractère (`char`), de plus elles peuvent être paramétrées par des classes d'équilibrage ou de gestion de noeuds.
L'intégralité du sujet est traité : les tries Hybride et Patricia sont implémentés, la fusion de Patricia, les conversions, la visualisation des graphes.
Les deux types de tries (Hybride et Patricia) implémentent l'interface `Trie` définie dans "Trie.h".

```
template<typename T>
class Trie {
public:
        virtual ~Trie() {}

        virtual bool Recherche(const std::vector<T>&) const = 0;
        virtual long ComptageMots() const = 0;
        virtual std::vector< std::vector<T> > ListeMots() const = 0;
        virtual long Prefixe(const std::vector<T>&) const = 0;

        virtual void Suppression(const std::vector<T>&) = 0;
        virtual void AjoutMot(const std::vector<T>&) = 0;
};
```

Les tests sont effectués à l'aide des fonctions définies dans le fichier `"TrieTest.h"`, les classes de Tries développés sont essentiellement testées l'une par rapport à l'autre, les fonctions de fusion et de conversions sont testées en vérifiant qu'une instance de taille importante est transformée comme il faut.

```
template<typename GivenTrie>
void assertTrie();

template<typename GivenTrie1, typename GivenTrie2>
void assertClassesTrieEgaux(const std::vector< std::vector<char> >&);

template<typename T>
void assertInstancesTrieEgaux(Trie<T>& t1, Trie<T>& t2);
```

Les tests de performance sont effectués à l'aide de l'ensemble des mots de l'oeuvre de Shakespeare (avec doublons) pour un total de 23086 mots uniques. Nous considérerons dans ce rapport que le but est d'améliorer les performances par rapport à ce jeu de données (donc alphabet inférieur à 256 caractères, lettres plus fréquentes que d'autres).

2 Structures de données

Sans prendre en compte la partie équilibrage, le `TrieHybride` s'implémente :

```
template <typename T>
class TrieHybrideNoeud {
public:
        T l;
        bool finmot;
        TrieHybrideNoeud<T>* g;
        TrieHybrideNoeud<T>* c;
        TrieHybrideNoeud<T>* d;
};
```

Le `TriePatricia` se compose de noeuds contenant des cases, qui s'implémentent :

```
template <typename T, template< typename, typename > class Map>
class Case {
public:
        std::vector<T> mot;
        TriePatriciaNoeud<T, Map>* lien;
};

template <typename T, template< typename, typename > class Map>
class TriePatriciaNoeud {
public:
        bool finmot;
        Map<T, Case<T, Map>> noeuds;
};
```

J'ai choisi de ne pas imposer le choix d'un de caractère de fin de mot pour les TriePatricia contrairement à ce qui est suggéré, pour cela je stocke un (`bool`) nommé `finmot` dans chaque noeud. Et je simplifie le cas où il n'y a qu'un seul mot sur un lien (c'est-à-dire du type menant vers noeud contenant seulment `finmot = true`), je considère que s'il y a un mot non vide sans lien alors le mot est dans le dictionnaire.

2.1 Conteneurs de noeuds de Patricia

Les tries Patricia sont paramétrables par une structure associative $Map : T \longrightarrow Case < T, Map >$, plusieurs choix sont envisageables :

Tableau de taille $|A|$: tableau permettant de contenir toutes les lettres : c'est-à-dire un R-trie, c'est trop lourd en mémoire, de plus cela impose que les lettres de l'alphabet soient consécutives (ou bien imposer une traduction).

Liste : la recherche et l'insertion prennent du temps et de nombreuses indirections, le parcours complet du Trie est très efficace, cela revient à faire un arbre de la Briandais autorisant des mots dans les noeuds.

ABR sur les lettres : cela revient à faire un trie Hybride équilibré à chaque niveau et permettant de mettre des mots en plus de lettres.

Table de hachage : permet une recherche en temps constant et une insertion constante amortie, la suppression est moins efficace, l'énumération dans l'ordre des éléments de l'arbre est ralentie par le tri des tables de hachage.

J'ai retenu le choix de la table de hachage pour la rapidité d'insertion, de recherche et l'efficacité mémoire.

2.2 Noeuds laissés lors d'une suppression "naïve"

Hybride noeuds de type *feuilles* : pas de fin de mot, ni de liens.
noeuds de type *branchements* : noeuds qui ne contiennent rien au centre `x->c`, ils peuvent êtres supprimés mais nécessitent d'insérer les sous arbres `x->g` et `x->d` dans le père de `x`.

Patricia les *feuilles* : le noeud courant ne contient pas de fin de mot, et aucun mot.

3 Equilibrage des Hybride

Remarquons que tenir en compte les liens `x->c` mots a pour effet de réduire le pire cas, ce qui peut être utile si le critère principal est la *réactivité* de l'application.
On suppose maintenant que l'on prend en compte seulement les liens gauche ou droite dans l'arbre. On peut équilibrer selon :

— on peut équilibrer selon un seul niveau (mais cela ne regarde pas le déséquilibre dans les sous arbres une fois que l'on continue au centre), la complexité est alors en $O(lg(|A|)|w|)$ dans le pire des cas, mais comme l'alphabet A est petit a priori, il y a peu de différence entre $|A|$ et $lg(|A|)$, en pratique le surcoût engendré par l'équilibrage n'est pas compensé par le gain en $O()$.
— on peut tenter de minimiser les indirections (gauche, droite) nécessaire pour trouver un mot, en équilibrant l'arbre selon la hauteur totale en nombre d'indirections de la racine a une feuille, mais l'arbre n'est plus un AVL (car le critère : différence de hauteur ≤ 1 ne peut pas toujours être respecté).

4 Conversions Hybride, Patricia

Patricia ⇒ Hybride Conversion peu efficace (115 ms) à cause du coût du tri des tables de hachage présentes dans les noeuds des Patricia. Notons $n_1, ..., n_k$ les tailles des noeuds (en nombre de cases) des TriePatriciaNoeud On effectue pour chaque TriePatriciaNoeud, le tri des cases $O(n_i lg(n_i))$, puis pour chaque case c on effectue des opérations en $O(n_i)$, puis on traite le lien associé à la case c. Le coût global (en temps) est donc en $O()$ de la somme des $n_i lg(n_i)$. Ce n'est pas optimal (linéaire en la taille du TriePatricia donné).

Hybride ⇒ Patricia Conversion très efficace (10 ms). L'algorithme s'exécute en temps linéaire en la taille du TrieHybride donné en entrée.

On peut obtenir la complexité théorique optimale *pour les conversions* en utilisant des listes chaînées ordonnées pour stocker les cases des TriePatriciaNoeuds, au lieu des tables de hachage.

5 Fusions de Patricia

La fusion des TriePatricia est assez efficace : (65ms) de création de chaque PatriciaTrie, puis (10ms) de fusion, donc environ (75ms) pour une création parallèle de tries (contre (130ms) pour la création en une seule passe).
L'algorithme se décompose en 3 fonctions mutuellement récursives dont le but est de permettre d'insérer des morceaux d'un TriePatricia dans un nouveau TriePatricia

FusionNoeud crée une copie d'un des TriePatriciaNoeud puis insère l'autre dedans.

InsererNoeud insère un TriePatriciaNoeud dans un autre TriePatriciaNoeud et retourne le résultat.

InsererCase insère une Case dans un TriePatriciaNoeud et retourne le résultat.

L'algorithme de fusion que j'ai implémenté a une complexité linéaire en la somme des tailles des TriePatricia donnés en entrée. L'essentiel étant de voir que pour créer un TriePatricia, l'ordre des Cases dans les noeuds n'importe pas donc on utilise seulement l'itération non ordonnée et l'accès par indice (en lecture et écriture), tout cela est en temps constant amorti.
La fusion de deux TriePatricia est très bien parallélisable, puisqu'elle se décompose en la fusion des cases de même lettre.

6 Complexité des fonctions de Trie

Le nombre d'accès mémoire (déréférencement de pointeur) me semble être le bon critère de complexité, on considère le pire cas pour le TrieHybride et la moyenne pour le TriePatricia. Une étoile $*$ dans un $O()$ indique une complexité amortie (utilisé dans le cas du doublement de la table de hachage). On notera w le mot pris en entrée de certaines fonctions qui demandent un mot on notera A l'alphabet, on utilisera la notation $|\cdot|$ pour indiquer la taille ou le nombre d'éléments. Le nombre n désigne le nombre de noeuds utiles dans le TrieHybride, et f, b désignent respectivement le nombre de noeuds inutiles (feuilles, branchements). Dans notre implémentation nous supprimons les feuilles, donc $f = 0$. Un prime $'$ sur un nombre indique « dans le sous arbre seulement ».

	Complexité (pire cas)		
	Hybride	HybrideAVL	Patricia
Recherche(w)	$O(\|A\| \cdot \|w\|)$	$O(lg(\|A\|) \cdot \|w\|)$	$O(\|w\|)$
ComptageMots()	$O(n + f + b)$	$O(n + f + b)$	$O(\Sigma\|w\|)$
Prefixe(w)	$O(\|A\| \cdot \|w\| + \|w\| \cdot \Sigma\|w'\| + f' + b')$	$O(lg(\|A\|) \cdot \|w\| + n' \cdot \|w\| + f' + b')$	$O(\|w\| * \Sigma\|w'\|)$
ListeMots()	$O(\Sigma\|w\| + f + b)$	$O(\Sigma\|w\| + f + b)$	$O(\Sigma\|w\| + \Sigma n_i lg(n_i))$
Suppression(w)	$O(\|A\| \cdot \|w\|)$	NON IMPLÉMENTÉ	$O(\|w\|^*)$
AjoutMot(w)	$O(\|A\| \cdot \|w\|)$	$O(lg(\|A\|) \cdot \|w\|)$	$O(\|w\|^*)$

7 Performance de l'implémentation

Les performances des Hybride et Patricia sont testées en insérant l'ensemble des mots de l'oeuvre de shakespeare (pour un total de 23086 mots). Le Hybride est très efficace en temps d'exécution, le Patricia est potentiellement plus efficace en mémoire. La fusion est potentiellement rentable en parallèle, et l'une des conversions est particulièrement efficace.

```
TrieHybride<char>
Ajout des mots: 73.26 ms
Recherche des mots: 54.788 ms
Comptage des mots: 23086 mots, 1.266 ms
Suppression des mots: 84.913 ms

TrieHybride<char, AVL>
Ajout des mots: 130.727 ms
Recherche des mots: 63.543 ms
Comptage des mots: 23086 mots, 1.307 ms
Suppression des mots: 91.712 ms

TriePatricia<char>
Ajout des mots: 133.534 ms
Recherche des mots: 153.54 ms
Comptage des mots: 23086 mots, 1.543 ms
Suppression des mots: 270.086 ms

TrieStdSet<char>
Ajout des mots: 209.012 ms
Recherche des mots: 205.157 ms
Comptage des mots: 23086 mots, 0.001 ms
Suppression des mots: 167.882 ms

TriePatricia<char> (1/2) AjoutMot: 17262 62.623 ms
TriePatricia<char> (2/2) AjoutMot: 16929 60.85 ms
Fusion TriePatricia<char> (1/2) (2/2): 23086 8.121 ms

Conversion TrieHybride<char> => TriePatricia<char>: 23086 12.28 ms
Conversion TriePatricia<char> => TrieHybride<char>: 23086 119.376 ms
```

Il est difficile de mesurer l'utilisation de la mémoire, cela serait une bonne métrique de performance des structures de données, car par exemple le TrieHybride utilise 3 pointeurs 64 bits (24 octets au total) pour stocker 1 octet.

8 Visualisation des structures de données

Les graphes ci dessous sont une représentation graphique des structures de données, ils sont tous construits à partir de l'ensemble des mots de la phrase suivante :

> "A quel genial professeur de dactylographie sommes-nous redevables de la superbe phrase ci dessous, un modele du genre, que toute dactylo connait par coeur puisque elle fait appel a chacune des touches du clavier de la machine a ecrire ?"

Ces visualisations permettent de remarquer plusieurs choses :

- Les TrieHybride comportent énormément de noeuds, un allocateur de bloc de taille fixe permettrait sûrement d'améliorer les performances.
- Le conteneurs de noeuds de type *Tableau* pour TriePatricia (avec $|A|$ noeuds) vont contenir de nombreuses cases vides, particulièrement pour les noeuds proches des feuilles.

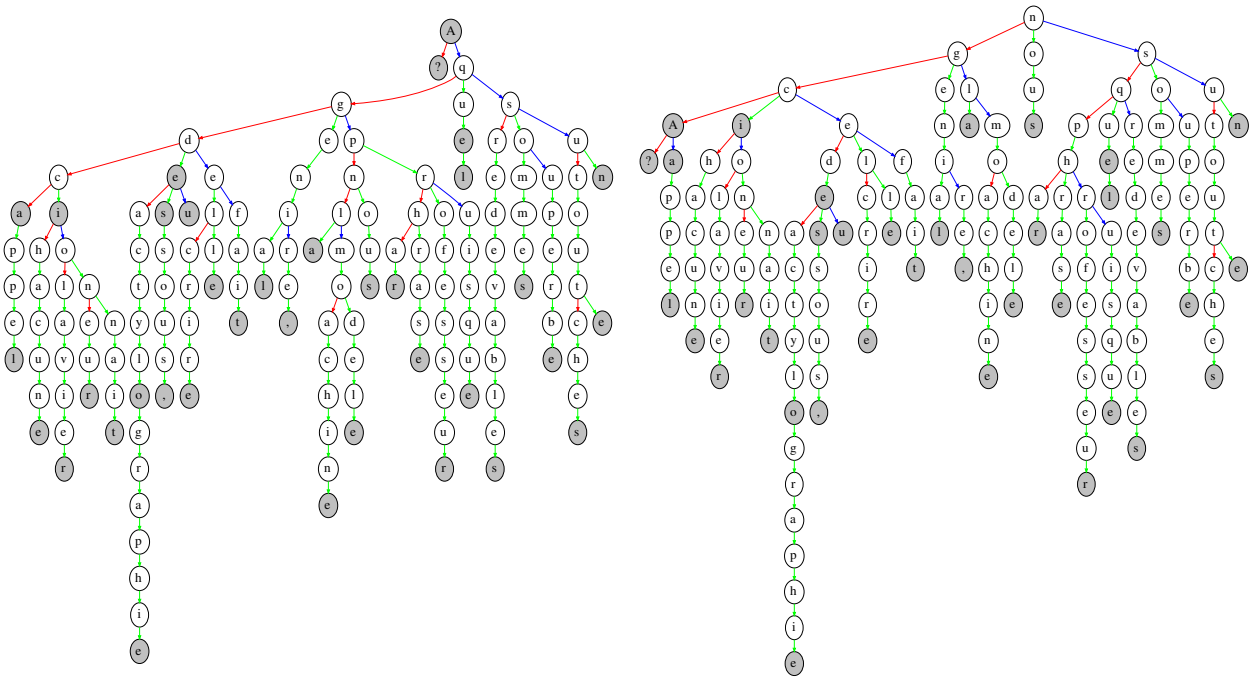

FIGURE 1 – TrieHybride sans équilibrage (à gauche) et avec équilibrage (à droite). Les flêches bleues, vertes, rouges représentent respectivement des liens gauche, centre, droite.

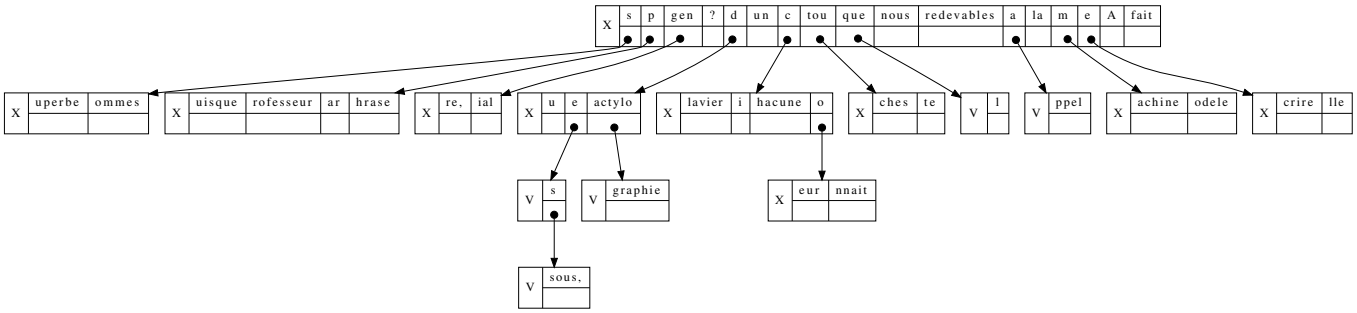

FIGURE 2 – La structure de donnée TriePatricia peut être plus compacte (ici seulement 15 noeuds).