Quel bazar!

Thomas Minier Benjamin Sientzoff

5 décembre 2014

Table des matières

1 Structures utilisées **3**
1.1 Arbre AVL 3
1.2 Classe-Union 3
1.3 Paire 5

2 Résolution du problème **6**
2.1 Notre approche 6
2.2 Résolution 6

Introduction

Dans le cadre du module d'algorithmique et structure de données 3, il nous a été demandé de proposer un algorithme pour résoudre un problème précis. Il s'agit, en partant d'un ensemble de pages provenant d'un livre, de les regrouper en chapitres. Pour cela, nous utiliserons un dictionnaire de mots et le critère suivant : si deux pages ont en commun au moins les k mêmes mots du dictionnaire, alors elles appartiennent au même chapitre.
Nous devons également identifier, pour chaque chapitre, les mots propre à ce dernier.

Pour résoudre ce problème, nous avons mis en place un programme écrit en Java articulé autour d'un algorithme principal utilisant plusieurs structures de données, présentées dans une première partie. Ces dernières ont été choisies pour rendre la résolution du problème aussi efficace que possible. Nous avons donc cherché à minimiser la complexité de l'algorithme principal par l'utilisation de structures efficaces. Notre démarche pour répondre au problème est détaillé dans une seconde partie.

Pour compiler et utiliser notre programme, procédez comme suit. En supposant que vous utiliser un système UNIX avec les programmes *java*, *javac* et *make* d'installés.

```
$ cd AVLTree
$ make
$ java -jar Bazar.jar <k> <dictionnaire> <page_1> <page_2> ... <page_n>
```

```
fonction contient
    entrée : elt element,
             noeud AVL
    sortie : Vrai si elt appartient à l'AVL
             Faux sinon
debut
    si (non estFeuille(noeud)) alors
        si (noeud.etq = elt alors
            retourner Vrai;
        sinon
            retourner noeud.fils_gauche.contains(elt)
                      OU noeud.fils_droit.contains(elt);
    sinon si (noeud.etq = elt) alors
        retourner Vrai;
    sinon
        retourner Faux;
fin
```

FIGURE 1 – Algorithme de la fonction *contains* de AVL

1 Structures utilisées

1.1 Arbre AVL

Il s'agit d'une structure d'arbre binaire avec équilibrage mise au point par Adelson-Velsky et Landis. Cet AVL est là pour stocker les mots du dictionnaire et, par page, les mots du dictionnaire présent dans la page. Nous avons implémenté la structure vue en cours, avec des méthodes d'ajout et d'équilibrage par rotations. Nous avons aussi ajouté des méthodes non vues en cours, qui sont détaillées çi dessous :

- `contient( elt T ) : booleen` est une fonction qui renvoie Vrai si le l'élément de type T passé en paramètre est présent dans l'arbre. L'algorithme de cette fonction est à la figure 1.
- `toString() : chaîne` est une fonction qui renvoie une représentation sous forme de chaîne de caractères de l'arbre. L'algorithme de cette fonction est à la figure 2.
- `nbCommuns( arbre : AVL ) : entier` est une fonction qui renvoie le nombre d'éléments en communs entre l'arbre courant et l'arbre passé en paramètre. L'algorithme de cette fonction est à la figure 3.

1.2 Classe-Union

Une classe-union est une structure de données permettant stocker des ensembles disjoints d'éléments. ces sous-ensembles sont appelés classes. Chaque classe a un représentant. Dans notre cas, les classes sont des arbres, leur représentant est alors la racine de cet arbre. L'avantage d'utiliser une telle structure est de pouvoir faire l'union des sous-ensembles simplement. Dans notre programme, les arbres des classes sont implémentées dans un tableau.

```
fonction toString
    entrée : noeud AVL
    sortie : une chaîne de caractères
debut
    si (estFeuille(noeud)) alors
        retourner noeud.etq;
    sinon
        retourner noeud.fils_gauche.toString()
                    + noeud.etq
                    + noeud.fils_droit.toString();
    finsi
fin
```

FIGURE 2 – Algorithme de la fonction *toString* de AVL

```
fonction nbCommuns
    entrée : arbre AVL
    sortie : entier
    variables : i entier
debut
    si (arbre.contains(noeud.etq)) alors
        i := 1
    sinon
        i := 0
    finsi
    retourner  i + noeud.fils_gauche.nbCommuns(arbre)
                 + noeud.fils_droit.nbCommuns(arbre)
fin
```

FIGURE 3 – Algorithme de la fonction *nbCommuns* de AVL

```
fonction trouver
    entrées : x element
    sorties : element
debut
    si ( x.parent != x ) alors
        x.parent := trouver( x )
    fin pour
    retourner x.parent
fin
```

FIGURE 4 – Algorithme de la fonction trouver de classe-union

```
procédure unir
    entrées : x, y element
    variables : p_x, p_y element
debut
    p_x := trouver( x )
    p_y := trouver( y )
    p_y.parent := p_x;
fin
```

FIGURE 5 – Algorithme de la fonction unir de classe-union

— `trouver( el T ) : T` est la première fonction pertinente d'une classe-union. Cette fonction permet de trouver le représentant d'une classe. Elle prend en paramètre un élément t et retourne le représentant de la classe à laquelle appartient *el*. L'algorithme de cette fonction est à la figure 4. Pour des questions de d'optimisation, cette fonction compresse les chemins de l'arbre, on obtient alors une complexité amortie en $\Theta(1)$.
— `union( el1 : T, el2 : T)` permet de faire l'union de deux classes. Son algorithme est disponible à la figure 5. Une telle fonction prend en paramètre deux éléments et va unir leurs deux classes. Sa complexité est de l'ordre de $\Theta(1)$.

1.3 Paire

Cette structure représente une paire d'éléments, appelés respectivement partie gauche et partie droite. Elle dispose de quelques fonctions et procédures, que nous détaillons ci-dessous.

— Des *getters*, `getLeft(): T` et `getRight(): T`, et des *setters*, `setLeft(T elt)` et `setRight(T elt)`, pour pouvoir accéder et modifier les éléments de la paire.
— `toString() : chaîne` renvoie une représentation sous forme d'une chaîne de caractères de la paire. Elle concatène le partie gauche et la partie droite de la paire et retourne la chaîne ainsi formée.

2 Résolution du problème

Quel bazar Le problème que nous devons résoudre est le suivant : reformer les chapitres d'un livre en regroupant les pages de ce dernier s'ils ont au moins k même mots du dictionnaire en commun. La difficulté ici est d'identifier les mots qui appartiennent au dictionnaire en lisant les pages puis de regrouper ces pages d'après les k mots en commun.

Concernant l'identification des mots uniques aux pages, nous ne nous en sommes pas préoccupés, par manque de temps.

2.1 Notre approche

Évidemment, une approche basique du problème consisterais à produire un algorithme (glouton) avec une complexité exorbitante. Mais soyons plus intelligent et penchons nous sur des structures efficaces, comme celles présentées dans la première partie de ce rapport. Tout d'abord pour les recherches des mots, en utilisant un arbre et non une structure de données linéaire, on passe d'une complexité en $\Theta(n)$ à un complexité en $\Theta(\log(n))$, n étant le nombre de mots dans le dictionnaire.

Ensuite, pour les pages, au lieu de garder en mémoire l'intégralité de ces dernières, stockons uniquement les mots qu'elles ont en commun avec le dictionnaire. La même question revient ensuite, structure linéaire ou arbre ? Arbre ! Potentiellement, comme pour le dictionnaire, on va effectuer des recherches sur les mots des pages. Et on peut estimer le nombre de telles recherches de l'ordre de $k \times n^2$.
En effet, on va obligatoirement chercher les k mots dans le dictionnaire. Ensuite pour ces k mots, on va comparer les pages obligatoirement deux à deux. Donc pour n pages, on obtient n^2 comparaisons.

2.2 Résolution

L'algorithme à la figure 6 correspond à l'algorithme qui regroupe les pages en chapitre, d'après les critères énoncés précédemment. On en retiendra les grandes étapes suivantes :

1. Lecture du dictionnaire, stockage dans un AVL.
2. Pour chaque page, parcourir la page et identifier les mots qu'elle a en commun avec le dictionnaire, puis placer ses mots dans un AVL.
3. On place toutes les pages dans une classe-union.
4. Pour chaque page, on test combien de mots elle a en commun avec les autres. Si elle en a au moins k, on réuni les deux pages (union de classes).
5. Enfin, on affiche la solution.

Complexité Considérons que le dictionnaire est composé de k mots. Il y a n pages composées de q_i mots, i étant l'identifiant de la page.
La lecture des fichiers en entrée du programme ce fait en un temps linéaire, proportionnel aux nombres des arguments et à la taille du dictionnaire et des

pages. La construction de l'arbre contenant le dictionnaire est alors en $\Theta(k)$. Ensuite, la construction des arbres des pages se fait en $\Theta(n \times q_i \times \log(k))$. $\log(k)$ car on va chercher les mots dans le dictionnaire. L'union et la recherche des pères dans la classe-union sont en $\Theta(1)$ grâce à la compression des chemins notamment. Là où la complexité de notre algorithme est importante, c'est lorsqu'on compare les pages pour reconstituer les chapitres. Des petites optimisation permettent de gagner du temps. Par exemple, on ne va pas comparer la page avec elle-même et on ne comparera pas deux pages qui sont déjà dans le même chapitre.

La comparaison des pages deux à deux est donc en $\Theta(n^2)$, ensuite l'accès à leur dictionnaire est au plus en $k \log(k)$. On obtient alors $n^2(\sum_1^n k \log(k))$ opérations. Étant donné qu'on cherche une majoration grossière, retenons que la complexité finale de notre programme est en $\Theta(n^2)$.

Pour comparer, si nous avions utiliser des structures linéaires, nous aurions plutôt obtenu une complexité de l'ordre de $\Theta(k \times n^2)$. On a donc réussi à gagner d'un facteur k en complexité.

```
Algorithme principal
    entrées : k entier,
              dico ensemble de mots,
              pages ensemble d'ensembles de mots
    variables : k entier,
                dictionnaire AVL de chaînes,
                collection Vector de Pair(chaînes, AVL de chaîne),
                classeUnion ClasseUnion de Pair(chaînes, AVL de chaînes),
                pairA, pairB Pair(chaîne, AVL de chaînes)
debut
    dictionnaire := AVL vide;
    collection := Vector de Pair(chaîne, AVL de chaîne) vide;

    pour chaque mot dans page faire
        dictionnaire.ajouter(mot);
    finpour

    pour chaque page dans pages faire
        pair := Pair(chaîne, AVL de chaîne) vide;
        pair.element_gauche := premier_mot(page);
        pair.element_droit := AVL vide;

        pour chaque mot dans page faire
            si (dictionnaire.contient(mot)) faire
                pair.element_doit.ajouter(mot);
            finsi
        finpour

        collection.ajouter(pair);
    finpour

    classeUnion := ClasseUnion(collection);

    pour i de 0 à taille(collection) - 1 faire
      pairA := collection[i]

      pour j de 0 à taille(collection) - 1 faire
        pairB := collection[j]
        si ( (i != j)
           ET (classeUnion.trouver(pairA) != classeUnion.trouver(pairB))
           ET (pairA.element_droit.nbCommuns(pairB.element_droit) >= k))
            classeUnion.union(pairA, pairB);
        finsi
      finpour
    finpour
    afficher(classeUnion);
fin
```

FIGURE 6 – Algorithme principal

Conclusion

Pour conclure, notre algorithme permet de fournir une solution plutôt efficace, tout de même de l'ordre de $\Theta(n^2)$. Ce qui n'est pas révolutionnaire, donc pas de publication pour le moment. D'autre part, nous n'avons pas répondu à la seconde question qui était de trouver les mots uniquement présent dans un chapitre. Mais nous pouvons donner un début de réponse.

Les mots propres aux pages Dans notre programme, pour chaque page stockons également dans une arbre AVL tous les mots de la page (et qui ne sont pas dans le dictionnaire, car ils sont forcément dans les autres chapitres, à priori). En suite, lorsqu'on compare les pages deux à deux, supprimons des mots de la pages les mots de l'autre page. Ainsi, après avoir comparé toutes nos pages, les AVLs qui contenaient tous les mots de la pages ne contiennent plus que les mots uniquement présent dans cette dernière et non dans les autres. Un tel algorithme aurait une complexité de l'ordre de $\Theta(n^2 \times \log(q_i))$.

En conclusion, ce projet aura été un bon moyen de nous frotter à des problèmes difficiles qui peuvent se résoudre plus aisément grâce aux structures de données vues en cours. De plus, les implémenter nous a permis de nous familiariser avec, en particulier les arbres AVL et les classes-union.