Projet 2018 - (Co)occurences de mots

David Braun & Jean-Michel Dischler

1 Introduction

L'objectif de ce projet est de trouver des (co)occurrences de mots au sein de plusieurs phrases d'un texte donné. De manière pratique, pour un ensemble de mots donné en entrée, on souhaite obtenir la liste des phrases (par ordre de lecture) où ces derniers apparaissent. L'ordre dans lequel les mots sont trouvés au sein d'une phrase n'a aucune importance tant qu'ils apparaissent effectivement dans la même phrase.

2 Implémentation

Pour résoudre ce problème, nous allons implémenter un dictionnaire de mots sous la forme d'un arbre binaire de recherche. Cet arbre binaire de recherche contient l'intégralité des mots du texte parcouru (chaque mot représente une clé de l'arbre). De plus, nous associons à chacun des noeuds de l'arbre un ensemble de positions des occurrences du mot dans le texte. C'est à dire que nous sauvegardons les indices de phrases dans l'ordre de lecture où le mot apparait. Par convention, si un même mot apparait plusieurs fois au sein d'une même phrase, il est sauvegardé qu'une fois dans l'ensemble des positions. La recherche des (co)occurrences de plusieurs mots consiste à faire l'intersection des ensembles de positions associés à chacun des mots dans l'arbre binaire de recherche. Ces deux structures de données sont crées en parcourant séquentiellement le texte, phrase par phrase et mot par mot.

Votre projet se décomposera en **4 parties** avec des **jeux de tests** et une **analyse théorique** :

1. Implémentation de l'ensemble de position
2. Implémentation de l'arbre binaire de recherche (ABR)
3. Construire automatiquement l'ABR en parcourant le texte
4. Mise à jour de l'ABR pour obtenir un arbre binaire de recherche équilibré (AVL)

Ci-dessous un exemple de texte représenté sous forme d'arbre binaire de recherche avec ses ensembles de positions :

foo bar baz grault.
qux foo bar corge.
corge waldo grault foo.

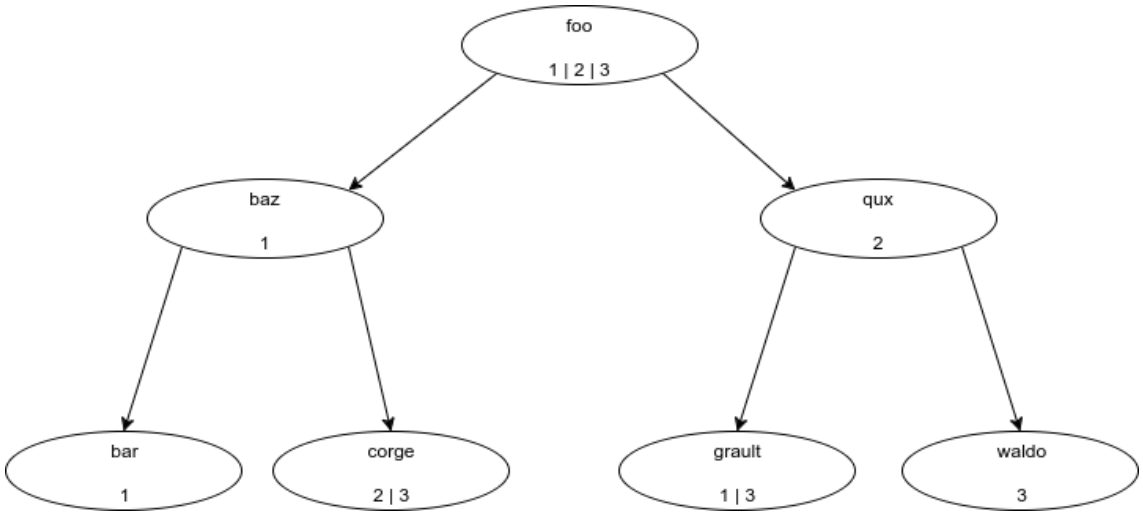

FIGURE 1 – Arbre binaire de recherche correspondant au texte ci-dessus. Le mot foo apparait dans les trois phrases tandis que le mot *waldo* n'apparait que dans la troisième.

3 Ensemble ordonné

Il s'agit d'implémenter l'ensemble des positions de manière séquentiel : un ensemble d'entiers ordonné. Une même valeur ne peut être présente qu'une seule fois dans la structure. Lors d'une opération d'insertion, penser à conserver l'ordre.

Ci-dessous une liste minimale d'opérations à implémenter qui est associée à cette structure :

- `initOrderedSet` qui crée un nouvel ensemble ordonné.
- `freeOrderedSet` qui libère la mémoire allouée pour l'ensemble ordonné.
- `getNumberElt` qui renvoie le nombre d'éléments contenus dans l'ensemble ordonné.
- `insertValue` qui insère un nouvel élément dans l'ensemble ordonné.
- `contains` qui teste si un élément appartient à l'ensemble ordonné.
- `printOrderedSet` qui affiche un ensemble ordonné.
- `intersect` qui réalise l'intersection de plusieurs ensembles ordonnés.

Etant donné l'application visée, il est nécessaire d'optimiser en priorité la complexité en temps de l'opération d'insertion et d'intersection, ainsi que l'espace mémoire occupé par cette structure.

4 Arbre binaire de recherche

Il s'agit d'implémenter un dictionnaire sous la forme d'un arbre binaire de recherche contenant des couples de valeurs : un mot (chaîne de caractères) et un ensemble de positions associé à ce mot. Le mot est utilisé pour trier et effectuer des recherches dans l'arbre binaire.

Ci-dessous une liste minimale d'opérations à implémenter qui est associée à cette structure :

- `initBinarySearchTree` qui crée un nouvel arbre binaire de recherche vide.
- `freeBinarySearchTree` qui libère la mémoire allouéev pour l'arbre.
- `getNumberString` qui renvoie le nombre de mots différents contenus dans l'arbre.
- `getTotalNumberString` qui renvoie le nombre de paires mots-phrases différentes contenues dans l'arbre.
- `insert` qui permet d'ajouter une nouvelle paire mot-position dans l'arbre :
  1. Si le mot n'existe pas, on crée un nouveau noeud contenant le mot et un nouvel ensemble de positions pour y stocker l'index de la phrase.
  2. Si le mot existe, on rajoute à l'index de la phrase au noeud correspondant.
  3. Rapel : Si un mot apparait plusieurs fois, on stocke une unique fois l'index de la phrase.
- `find` qui permet de retrouver tous les indices d'occurrences d'un mot.
- `findCooccurrences` qui permet de retrouver tous les indices de coocurrences de plusieurs mots.
- `printBinarySearchTree` qui affiche un arbre binaire de recherche.

A noter qu'il n'est pas nécessaire d'implémenter une opération de suppression de clés pour ce problème.

5 Construction automatique de l'arbre

L'étape suivante consiste à programmer des fonctions permettant de lire un texte à partir d'un fichier afin de construire l'arbre binaire de recherche correspondant et de remplir les ensembles de positions. Dans un premier temps, il est important de considérer une phrase par ligne sans ponctuation et sans accent où le retour chariot constitue le délimiteur de fin de phrase.

La boucle principale de parcours du fichier ressemble à :

1. Ouverture du fichier
2. Initialisation de l'arbre binaire de recherche
3. Boucle de lecture ligne par ligne tant que le fichier n'est pas fini
   (a) Boucle de lecture mot par mot tant que la phrase n'est pas fini
   (b) Insertion de chaque paire mot-position dans l'arbre
4. Fermeture du fichier
5. Renvoie de l'arbre binaire de recherche complété

En Bonus : Gestion des accents et de la ponctuation

6 Arbre binaire de recherche équilibré

La dernière étape de ce projet consiste à améliorer la structure d'arbre binaire utilisé pour faciliter les opérations d'insertion et de recherche d'un élément. Pour cela, il faut implémenter les opérations d'équilibrages d'arbres binaires **vues en cours** afin de rééquilibrer un arbre après chaque insertion d'un nouveau mot dans l'arbre. Pour rappel, le rééquilibrage d'un noeud dans un tel arbre a lieu lorsque la différence de hauteur entre son sous-arbre gauche et son sous-arbre droit n'est pas égal à 1,0 ou -1. Il faut donc ajouter et maintenir à jour sur chaque noeud de l'arbre un facteur d'équilibrage correspondant à cette différence.

Ci-dessous une liste d'opérations minimale à rajouter à la structure d'arbre binaire :

- `isBalanced` qui détermine si l'arbre est équilibré ou non. Un arbre est équilibré si pour chacun de ces noeuds, la hauteur du sous-arbre droit de ce noeud ne diffère pas de la hauteur du sous-arbre gauche de plus d'une unité.
- `getHeight` qui renvoie la hauteur de l'arbre.
- `getAverageDepth` qui renvoie la profondeur moyenne de chaque noeud de l'arbre.
- `rotateLeft` qui effectue une rotation simple gauche.
- `rotateRight` qui effectue une rotation simple droite.
- `rotateRight` qui effectue une rotation double droite-gauche.
- `rotateRight` qui effectue une rotation double gauche-droite.

Pour plus de détails, voici deux liens complémentaires sur les opérations de rotations :

— Cours I : https://cours.etsmtl.ca/SEG/FHenri/inf145/Suppl%C3%A9ments/arbres%20AVL.htm
— Cours II : https://www.labri.fr/perso/maylis/ASDF/supports/notes/AVL.html

7 Analyse théorique

Il vous est demandé de fournir un rapport de quelques pages où vous répondrez aux questions suivantes :

1. Ensemble ordonné
   (a) Décrire et justifier l'implémentation de l'ensemble ordonné.
   (b) Expliquer le fonctionnement de la fonction intersection.
   (c) Identifier le pire cas et la complexité pour les opérations.
      i. Insertion en fonction de n, le nombre d'éléments de l'ensemble.
      ii. Test de l'appartenance d'un élément en fonction de n, le nombre d'éléments de l'ensemble.
      iii. Intersection de deux ensembles (en fonction de m, le nombre total d'éléments dans les deux ensembles).

2. Arbre binaire de recherche
   (a) Choix d'implémentations pour l'arbre binaire de recherche.
   (b) Expliquer le fonctionnement des fonctions `getAverageDepth` et `isBalanced`.
   (c) Etude de la complexité de `FindCooccurrences`.
      i. Etudier la hauteur moyenne d'un noeud entre un arbre non équilibré et équilibré contenant le même nombre de noeuds.
      ii. Caractériser la complexité dans le pire cas en supposant qu'on recherche deux mots dans un arbre binaire de recherche non équilibré contenant n mots et qu'un mot apparait au plus dans k phrases différentes.
      iii. Que devient cette complexité si l'arbre est équilibré ?

8 Deadline et soumission

Le projet est à réaliser en binôme avant le 7 mai 2018 à 23h59. Il est à rendre via la plateforme moodle sous la forme d'une archive tar.gz contenant :

- L'analyse théorique et les choix d'implémentations au format PDF (4 pages maximum).
- Le code source commenté du projet ainsi que le makefile.
- Jeux de données utilisés.

Notation : Tout code qui **ne compile pas** ou **non rendu à temps** recevra une note nulle. La notation prendra tout particulièrement en compte les choix d'implémentations des structures de données. Il est important de documenter le fonctionnement de votre programme et en particulier son lancement sur différents jeux de données. Profiter de la décomposition incrémentale de votre projet pour tester chacunes des fonctionnalités grâce à différents jeux de tests. Finalement ne pas oublier de libérer la mémoire.