Cet énoncé comporte 10 pages

Algorithmique Avancée 2 - Projet

Licence Informatique - 2^{me} année

année 2017-2018

L'objectif de ce projet est d'écrire une application permettant de mesurer une distance entre des textes, en vue de déterminer si ces textes ont été écrits par le même auteur. Cette application permettra également de tester de manière pratique les diverses structures de données vues en cours et en TD (liste chaînées, arbres binaires de recherche et tables de hachage), ainsi que les bonnes pratiques de documentation de fichiers sources.

Le projet peut être réalisé par binôme (2 étudiants maximum) ou individuellement. Il donnera lieu à la rédaction d'un petit rapport et d'un contrôle individuel sur machine le jeudi 17 mai 2018. Ce contrôle consistera essentiellement à rajouter des fonctionnalités aux classes qui auront été écrites. La note finale du projet sera calculée comme suit : une note sur 7 points sur la partie correspondant à ce sujet et une note sur 13 points pour le contrôle individuel.

Les sources du projet et le rapport devront se trouver sur le compte ULCO de **chaque étudiant** pour le mardi 15 mai 2018 à 19h, dans votre dossier `Documents/AAC2/Projet`. Le nom des étudiants ayant participé à la réalisation du projet devront se trouver dans le rapport et dans les sources, sous forme de commentaire de type *doxygen*[1].

On précise que bien que vous puissiez travailler sur vos machines pour développer le projet, les sources que vous déposerez sur vos comptes ULCO doivent compiler et fonctionner sur les machines des salles TP, même si l'ensemble des fonctionnalités n'a pas été développé. Il est donc conseillé de prévoir un peu de temps en fin de projet pour vérifier ces propriétés.

Préliminaires

Organisation du projet

Le projet se décompose en deux parties, qu'il est nécessaire de réaliser dans l'ordre :

— la première partie consiste à développer les classes nécessaires à la représentation en mémoire des mots présents dans un texte au sein de diverses structures de données et à évaluer l'efficacité de chacune de ces structures ;

— la seconde partie consiste à utiliser ces structures de données, pour évaluer une distance entre deux textes, dont la valeur permettra d'envisager si les deux textes sont du même auteur ou pas. La notion de distance entre texte qui devra être développée dans cette partie sera présentée en TD.

Pour chacune des parties, ce sujet vous propose une série d'étapes à réaliser, afin de vous faciliter le développement. Il est rappelé qu'il est nécessaire que chaque étape soit testée avant de passer à la suivante.

Données fournies

L'archive associée à cet énoncé comporte un dossier `Projet` contenant :

— un dossier `Sources` qui contient plusieurs fichiers sources, détaillés dans ce qui suit. Le fichier principal, qui permettra de lancer l'application à ses différents stades de développement, se nomme `mesurer.cpp`. Tous les fichiers sources que vous développerez devront se trouver dans ce dossier ;

— Un dossier `Obj`, vide, qui permettra de centraliser tous les fichiers `.o` qui resulteront de vos compilations ;

1. voir annexe

— une version initiale du fichier `Makefile`, qui est utilisable directement et qui illustre la manière de créer les lignes de compilation pour assurer que les fichiers sources soient bien recherchés dans le dossier `Sources` et les fichiers objets soient bien installés dans le dossier `Obj` ;
— un dossier `Tests` contenant plusieurs petits fichiers texte, que vous pourrez utiliser durant les phases de mise au point de vos algorithmes ;
— un dossier `Data` qui contient des fichiers texte de taille importante, représentant des oeuvres tombées dans le domaine public. Du fait de leur taille, ces fichiers ne devront être utilisés que lorsque cela sera précisé dans l'enoncé ;
— un dossier `html` qui contient une documentation de type *doxygen* des fichiers sources fournis. Cette documentation est accessible à partir du fichier `index.html` qui se trouve dans ce dossier ;
— un dossier `Rapport`, qui contient la trame du rapport que vous aurez à rédiger et à rendre avec le projet ;
— un fichier de configuration pour l'utilitaire *doxygen* est également présent, que vous ne devez *a priori* pas modifier, et qui vous servira lors de la génération de la documentation des classes que vous développerez.

Partie 1 : Construction de structures de données pour le stockage des mots en mémoire.

Avant de pouvoir évaluer la distance entre deux textes, il est nécessaire de charger ceux ci-en mémoire, dans une structure de données permettant leur représentation et leur manipulation. Cette partie vise à développer 3 structures vues en cours et en TD et à comparer leur efficacité.

Lecture d'un fichier texte

Compléter le module `mesurer.cpp` de manière à ce qu'il relise le contenu des deux fichiers passés en paramètre. Pour tester votre code, il peut être utile, pour cette étape, d'afficher ce qui est relu à l'écran. Vous pourrez tester votre code avec les petits fichiers texte se trouvant dans le dossier `Tests`.

Stockage dans une liste chaînée

Contexte

Une classe `Liste` vous est fournie (fichiers `Liste.hpp` et `Liste.cpp`). Elle permet de représenter une liste chaînée de mots, ces derniers étant stockés dans un maillon de la liste (classe `Maillon`, fichiers `Maillon.hpp` et `Maillon.cpp`). Chaque maillon contient un mot et son nombre d'occurrence (le nombre de fois qu'il apparaît dans un texte, un même mot ne pouvant apparaître qu'une seule fois dans la liste). Afin de faciliter son utilisation (en particulier en cas de recherche d'un mot), la liste doit être triée par ordre alphabétique croissant.

Exemple : la figure 1 illustre cette structure de liste chaînée ; chaque mailllon contient un mot du texte lu en entrée, son compteur d'occurrences ayant une valeur qui correspond au nombre de fois que le mot est présent dans le texte (par exemple, le mot `peu` n'est présent qu'une seule fois, contrairement aux mots `petit` et `un` qui sont présents deux fois). Notez également que la liste est triée par ordre alphabétique des mots présents.

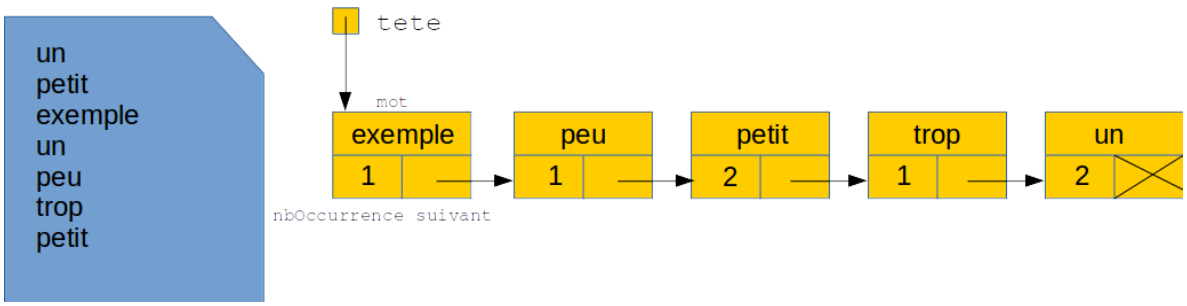

FIGURE 1 – Représentation de la liste chaînée obtenue (partie droite) à partir du fichier texte figurant en partie gauche.

La classe `Liste`

Les fichiers fournis contiennent l'ensemble des méthodes nécessaires à ce stade pour pouvoir manipuler une liste chaînée de mots. La classe est définie dans le fichier `Liste.hpp` (attributs et méthodes), chaque élément de celle-ci étant commenté au format *doxygen*. Vous retrouvez tous les éléments d'information dans la description `html` de la classe. Le fichier `Liste.cpp` contient l'implémentation de certaines des méthodes de la classe, le tout étant entièrement commenté pour une meilleure compréhension. Notez que vous disposez également d'une implémentation complète de la classe `Maillon` (définition, implémentation, documentation), dont les attributs sont ici publics afin de vous faciliter son utilisation.

Travail à réaliser : À ce stade, il vous est demandé deux choses :

- lors de la lecture des deux fichiers de la question précédente, de stocker leurs mots dans deux listes chaînées (une par fichier), puis d'afficher le contenu de chacune des listes. Vous aurez ainsi à compléter la méthode `afficher` de votre classe `Liste`, en suivant les spécifications présentes dans la documentation de la classe et dans les commentaires. Vous pourrez utiliser les petits fichiers texte présents dans le dossier `Tests` pour vérifier le fonctionnement de votre application ;
- compléter la méthode `rechercher` de la classe `Liste`, en vous conformant également aux spécifications de la documentation et aux commentaires présents dans les fichiers. Vous pourrez tester votre implantation en recherchant certains mots dans chaque liste et en vérifiant leur présence et leur nombre d'occurrence.

Stockage dans un arbre binaire de recherche

Contexte

La seconde structure à implémenter pour le stockage des mots d'un fichier est un arbre binaire de recherche. Deux classes vous sont fournies pour ce faire : la classe `Noeud`, qui permet de représenter un noeud d'un arbre binaire et la classe `Abr` qui permet de représenter l'arbre binaire lui-même. Contrairement à la classe `Liste`, ici seuls vous sont donnés les fichiers de type `.hpp`, qui contiennent néanmoins tous les commentaires nécessaires pour la compréhension des méthodes et la génération de la documentation *doxygen* correspondante.

On précise que, comme pour les listes précédentes, un mot présent plusieurs fois dans un texte ne poura être présent qu'une seule fois dans un arbre binaire, un compteur d'occurrences présent dans la classe `Noeud` permettant à nouveau de comptabiliser le nombre d'apparitions d'un mot dans le texte.

Exemple : la figure 2 illustre cette structure d'arbre binaire de recherche, appliquée à un petit fichier de test. Chaque noeud contient un mot du texte lu en entrée, son compteur d'occurrences ayant une valeur qui correspond au nombre de fois que le mot est présent dans le texte. Notez que la propriété de tri des arbres binaires de recherche s'applique bien sur les mots (et non pas les compteurs d'occurrences), qui seront donc considérés commes les clés dans les opérations sur les arbres.

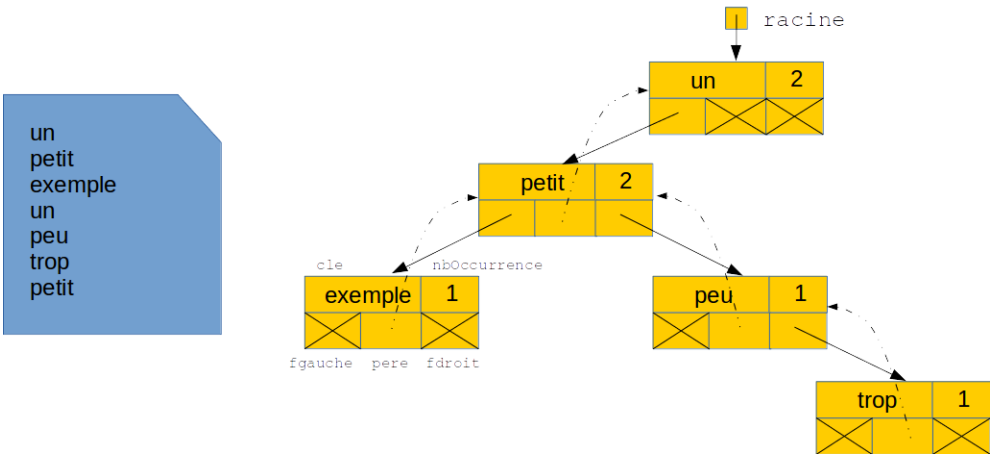

FIGURE 2 – Représentation de l'arbre binaire de recherche (partie droite) obtenu à partir du fichier texte figurant en partie gauche.

Travail à réaliser

— implémenter les classes `Noeud` et `Abr` en fonction de leurs spécifications ;
— tester leurs fonctionnalités dans l'application de mesure, de manière similaire à ce qui a été fait pour les listes, mais en chargeant les fichiers texte dans des arbres binaires de recherche plutôt que dans des listes chaînées. Vous utiliserez les petits fichiers d'exemple fournis dans le dossier `Tests`.

Stockage dans une table de hachage

Contexte

La troisième et dernière structure à implémenter est une table de hachage à adressage ouvert. Vous sont fournies deux classes :
— la classe `Htable`, qui permet de représenter une telle table ;
— la classe `Mot`, qui permet de représenter le contenu d'une alvéole de la table.
Vous disposez des deux fichiers de type `.hpp` correspondant, mais sans aucune documentation de type *doxygen*. Quelques explications concernant ces classes sont données ci-après.

La classe `Mot`

Une instance de la classe `Mot` aura pour attributs une chaîne de caractères représentant le mot (attribut `valeur`) et un entier représentant le nombre de fois que le mot a été rencontré dans le texte (attribut `nbOccurrence`).

Deux constructeurs sont proposés : le constructeur par défaut qui initialise le mot à « vide » et un constructeur prenant en paramètre le mot devant se trouver dans l'instance de l'objet construit.

La classe dispose également de plusieurs méthodes, qui sont *a priori* les seules à devoir être présentes (mais rien ne vous interdit de rajouter des méthodes si vous le jugez nécessaire) :
— `bool egal(Mot m)` : teste si le mot passé en paramètre est le même que le mot de l'instance appelante ;
— `afficher()` : permet d'afficher le mot de l'instance courante et son nombre d'occurrences ;
— `string getValeur()` : accesseur permettant de récupérer le mot stocké dans l'instance appelante ;
— `int getOccurrence()` : accesseur permettant de récupérer le nombre d'occurrences du mot stocké dans l'instance appelante ;
— `void addOccurrence()` : incrémente le nombre d'occurrences du mot socké dans l'instance appelante ;
— `int getCle(int base, int size)` : fournit la clé d'entrée dans la table de hachage de l'instance appelante. Cette méthode implémente la formule de calcul d'une clé entière associée à une chaîne de caractères vue en cours, avec `base` l'entier multiplicateur du code des caractères et `size` la taille de la table de hachage, afin d'appliquer un modulo. Pour mémoire, cette formule s'écrit sous la forme :

$$cle(s) = (\sum_{i=1}^{|s|} code(s_i) \times base^{i-1}) \bmod size$$

avec s la chaîne dont il faut calculer la clé. La somme présente dans cette formule devra être calculée dans un entier de type `long` afin de disposer d'une précision importante pour les chaînes comportant de nombreux caractères.

La classe `Htable`

Une table de hachage disposera des attributs suivants :
— Un pointeur vers une zone mémoire représentant toutes les entrées possibles dans la table (attribut `alveole`). Cette zone devra être allouée dynamiquement en fonction de la valeur de la taille de la table de hachage (une taille par défaut est définie dans la constante `DEF_SIZE`) ;
— Un pointeur vers une zone mémoire représentant l'état d'affectation de chaque alvéole durant l'utilisation de la table (attribut `etat`). Cette zone devra également être allouée dynamiquement en fonction de la valeur de la taille de la table de hachage. Chaque entrée de la zone ne pourra prendre que l'une des trois valeurs définies dans l'énumération `Etat`, à savoir `LIBRE` si l'alvéole correspondante n'est pas utilisée, `OCCUPE` dans le cas contraire et `EFFACE` si elle a été occupée mais que le mot stocké dans l'alvéole correspondante a été effacé ;
— un entier contenant le nombre d'entrées total dans la table.

Dans la version qui vous est fournie, seules 3 méthodes sont prévues : le constructeur par défaut, qui crée et initialise une table de `DEF_SIZE` entrées ; un second constructeur, auquel on fournit en paramètre la taille de la table souhaitée et un destructeur. Il vous appartiendra de déterminer quelles sont les autres méthodes nécessaires.

Exemple : La figure 3 illustre la structure de table de hachage proposée, à partir d'un petit fichier d'exemple. Chaque alvéole contient un mot et son compteur d'occurrences, et le tableau `etat` indique si une alvéole est occupée, libre ou effacée (cette dernière possibilité n'apparaissant pas sur cet exemple).

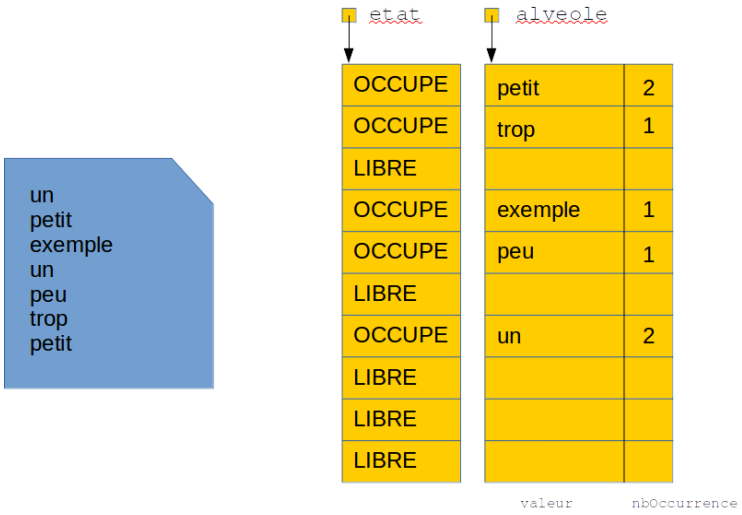

FIGURE 3 – Représentation d'une table de hachage (partie droite) obtenue à partir du fichier texte figurant en partie gauche.

Travail à réaliser

- implémenter la classe `Mot`, en fonction de ses spécifications ;
- implémenter la classe `Htable`, en vous appuyant sur les fonctionnalités disponibles dans les classes `Liste` et `Abr` ;
- compléter la documentation de type *doxygen* de ces deux classes et générer la documentation `html` correspondante ;
- tester leurs fonctionnalités dans l'application de mesure, de manière similaire à ce qui a été fait pour les listes et les arbres binaires, en chargeant les fichiers texte dans des tables de hachage.

Comparaison des différentes structures

On souhaite à présent pouvoir comparer l'efficacité de chacune de ces trois structures sur des fichiers de plus grande taille et montrer expérimentalement que le coût de chacune d'entre-elle correspond bien au coût théorique qui a été vu en cours.

Travail à réaliser

- ajouter au fichier `mesurer.cpp` les trois fonctions suivantes et modifier votre fonction `main` pour tester chacune d'entre-elles lorsqu'elle est terminée :
  - `int chargerListe(char *nomFichier, Liste *l)` : cette fonction a pour objectif de relire le fichier dont le nom est passé en paramètre, mot à mot, et d'insérer chacun des mots lus dans la liste également passée en paramètre. La liste doit être passée par adresse, car son contenu sera modifié. La fonction retourne 0 si tout s'est bien passé, une valeur négative sinon ;
  - `int chargerArbre(char *nomFichier, Abr *a)` : cette fonction a pour objectif de relire le fichier dont le nom est passé en paramètre, mot à mot, et d'insérer chacun des mots lus dans l'arbre binaire de recherche également passé en paramètre. L'arbre doit être passé par adresse, car son contenu sera modifié. La fonction retourne 0 si tout s'est bien passé, une valeur négative sinon ;

— `int chargerTable(char *nomFichier, Htable *h)` : cette fonction a pour objectif de relire le fichier dont le nom est passé en paramètre, mot à mot, et d'insérer chacun des mots lus dans la table de hachage également passée en paramètre. La table doit être passée par adresse, car son contenu sera modifié. La fonction retourne 0 si tout s'est bien passé, une valeur négative sinon.

— Modifier votre fonction `main` de telle sorte qu'elle prenne en compte le second paramètre de la ligne de commande (`-l`, `-a` ou `-h`), afin de charger les fichiers dans le type de structure souhaité. À titre d'exemple, la commande suivante :

```
./mesurer -a Tests/court.txt Tests/court2.txt
```

chargera les mots contenus dans les deux fichiers `court.txt` et `court2.txt` dans deux arbres binaires de recherche ;

— Modifier enfin le code de votre fonction `main` de telle sorte que vous puissiez mesurer le temps (en seconde) de chargement des fichiers, selon la structure utilisée. Vous pourrez vous référer au premier TP du S3 pour ce faire. Lorsque votre code sera opérationnel sur les petits fichiers fournis dans le dossier `Tests`, vous testerez celui-ci sur les exemples de plus grande taille fournis dans le dossier `Data` et vous compléterez la première partie de la trame du rapport fourni (tableau de mesure, réponse aux questions).

Partie 2 : Évaluation des distances intertextuelles

La seconde partie du projet concerne l'utilisation de l'application développée en première partie pour mesurer une distance intertextuelle entre deux textes et permettre (éventuellement) de donner une indication sur le fait que deux textes sont potentiellement du même auteur. Pour ce faire, vous utiliserez la formule de calcul 1 qui suit et qui a été détaillée en TD :

$$d(A,B) = \frac{\sum_{V_A} |N_{A,i} - N_{B,i}| + \sum_{V_B} |N_{B,j} - N_{A,j}|}{T_A + T_B} \quad (1)$$

avec :

- A et B les deux textes à considérer ;
- $d(A,B)$ la distance intertextuelle en ces deux textes ;
- V_A et V_B le vocable de chacun des deux textes, c'est à dire l'ensemble des mots différents présents respectivement dans A et dans B ;
- $N_{A,i}$ le nombre d'occurrences du i^{eme} mot du vocable du texte A et $N_{B,i}$ le nombre d'occurrences de ce même mot dans le vocable du texte B ;
- $N_{B,j}$ le nombre d'occurrences du j^{eme} mot du vocable du texte B et $N_{A,j}$ le nombre d'occurrences de ce même mot dans le vocable du texte A ;
- T_A et T_B le nombre total de mots présents respectivement dans le texte A et dans le texte B.

Compter le nombre de mots présents

Dans la formule de calcul 1 rappelée ci-dessus, apparaissent le nombre de mots présents dans chacun des deux textes. Il est donc nécessaire de pouvoir retrouver ce nombre pour chacune des trois structures développées précédemment.

Travail à réaliser

Ajouter à chacune des structures `Liste`, `Abr` et `Htable` une méthode de prototype :

```
int nbMots()
```

qui permet de calculer combien de mots (au total) sont stockés dans la structure. Cette méthode doit donc prendre en compte le nombre d'occurrences de chaque mot ...

Calculer la différence entre deux textes

Le numérateur de la formule 1 effectue la différence (en valeur absolue) entre le premier texte et le second, puis entre le second et le premier. Dans les deux cas, il s'agit du même calcul, mais qui sera appliqué d'abord à partir du premier texte, puis à partir du second.

D'un point de vue pratique, il s'agit de parcourir le premier texte et, pour chacun de ses mots, sommer la différence entre le nombre de fois où il apparaît dans le premier texte et celui où il apparaît dans le second. On effectue ensuite la même opération en sens inverse, à partir du second texte.

Travail à réaliser

Ajouter à chacune des structures `Liste`, `Abr` et `Htable` une méthode qui permet de calculer la différence entre deux textes (c'est à dire l'un des deux termes du numérateur de la formule 1). Ces méthodes seront déclarées dans une partie privée à chaque classe et elles auront les prototypes suivants :

- `int Liste::difference(Liste *l)` pour la classe `Liste` ;
- `int Abr::difference(Abr *a)` ou `int Abr::difference(Noeud *n, Abr *a)` pour la classe `Abr`, selon que vous écriviez une version non récursive ou récursive de la méthode. Dans ce dernier cas, le paramètre `n` représente la racine du sous-arbre pour lequel on recherche les mots dans le second arbre, passé dans le paramètre `a` ;
- `int Htable::difference(Htable *t)` pour la classe `Htable`.

Calculer la distance entre deux textes

À partir des deux méthodes écrites précédemment, il est désormais possible de calculer la distance finale entre deux textes, en appliquant la formule 1.

Travail à réaliser

Ajouter à chacune des trois structures de données une méthode permettant de calculer la distance entre deux textes, l'un contenu dans une instance appelante de la structure concernée, l'autre contenu dans une structure passée en paramètre. Le prototype de ces fonctions sera le suivant :

— `float Liste::distance(Liste *l)` pour la classe `Liste` ;
— `float Abr::distance(Abr *a)` pour la classe `Abr` ;
— `float Htable::distance(Htable *t)` pour la classe `Htable`.

Lorsque ces fonctions seront au point (à tester sur les petits fichiers fournis dans le dossier `Tests`), vous pourrez compléter la seconde partie du rapport et répondre aux questions qui y sont posées.

On précise que, quelle que soit la structure utilisée, la distance obtenue entre deux textes doit être la même ...

Suppression des doublons dans l'intersection

La formule de calcul 1 de la distance entre deux textes pose le problème de compter deux fois la différence du nombre d'occurrences des mots communs aux deux textes. On souhaite vérifier que ce problème ne génère pas d'erreurs dans l'interprétation du fait que des textes peuvent être considérés ou pas comme écrits par le même auteur. Pour ce faire, on propose une petit modification dans la formule de calcul de la distance :

$$d(A,B) = \frac{\sum_{V_A} |N_{A,i} - N_{B,i}| + \sum_{V_B - (V_A \cap V_B)} |N_{B,j} - N_{A,j}|}{T_A + T_B} \quad (2)$$

L'idée, pour éviter de compter deux fois les mots communs dans les sommes du numérateur, est de les compter dans la première somme de celui-ci, mais de n'appliquer la seconde somme que sur les mots de B qui ne sont pas communs avec A, soit l'ensemble $V_B - (V_A \cap V_B)$. Une manière simple d'implanter cette nouvelle formule consiste alors, durant l'évaluation de la première somme, à supprimer de B tous les mots communs avec A. Lorsque la seconde somme sera évaluée, il ne restera donc dans B que les mots qui ne se trouvent que dans cet ensemble.

Travail à réaliser

— Ajouter à chacune des 3 structures de données développées une méthode permettant de supprimer un mot passé en paramètre (le type du paramètre et la valeur de retour dépendront de la structure) ;
— Modifier les méthodes `difference` et `distance` de vos trois structures, de telle sorte qu'elles prennent toutes un paramètre booléen supplémentaire, permettant de préciser si durant leur exécution, on souhaite conserver les doublons (formule 1) ou pas (formule 2) ;
— vous testerez votre nouvelle formule de calcul sur les textes fournis et vous compléterez la dernière partie du rapport.

Annexe : Introduction à *doxygen*

L'objectif de cette annex est de vous fournir une petite introduction à un outil de documentation de code dont le nom est `doxygen`. Des informations plus complètes peuvent être trouvées sur le site officiel `www.doxygen.org` ou via une recherche sur le web.
)

Doxygen

Introduction

Doxygen est un système de documentation de code pour divers langages de programmation, associant des commentaires insérés dans le code source, via une syntaxe spécifique, à un utilitaire analysant ces commentaires et générant une documentation dans différents formats (html, latex, rtf, xml, pdf, etc.). *Doxygen* est multi-plateforme, des versions existant pour Linux, Mac OS et Windows. Dans ce qui suit, nous ne décrirons que le mode de fonctionnement sous Linux. Si *doxygen* n'est pas installé sur votre système, reportez vous à une page expliquant la procédure d'installation (par exemple `https://doc.ubuntu-fr.org/doxygen` pour *Ubuntu*).

Fonctionnement

Le principe de base de *doxygen* est d'insérer le texte de la documentation dans les fichiers source (généralement dans les fichiers `.hpp` pour documenter les classes en `C++`), en utilisant une forme particulière de commentaires. Différentes formes sont disponibles, mais ne n'en décrirons ici qu'une seule, utilisée dans les fichiers fournis avec le projet. Cette approche permet d'une part, d'avoir une cohérence entre la documentation externe disponible pour un projet et les commentaires insérés par les développeurs dans le code source. Elle permet d'autre part d'obliger ces mêmes développeurs à commenter leur source s'ils veulent pouvoir mettre à disposition une documentation de leur travail.

Les commentaires dans les fichiers source Un bloc de documentation *doxygen* prendra ici la forme d'un commentaire ayant la forme suivante :

```
/**
 * texte de la documentation
 * suite de la documentation
 * etc.
 */
```

Le style de commentaire illustré ci-dessus s'apparente aux commentaires du langage C, à la différence que le début d'un commentaire est introduit par la suitecaractères `/**`.

Un tel bloc de documentation s'appliquera sur l'élément de code qui suit, sauf cas particulier de syntaxe (voir par exemple la documentation des attributs de la classe `Liste` dans le fichier `Liste.hpp` : la syntaxe `/*< documentation */` permet de préciser que la documentation porte sur le code qui se situe avant le commentaire.). Des *tags* particuliers, qui seront vus ci-après, permetten d'enrichir un bloc de documentation avec des informations spécifiques.

Génération de la documentation Lorsque tout ou partie de vos fichiers source ont été commentés avec la syntaxe *doxygen*, vous pouvez lancer la commande `doxygen` dans le dossier où se trouvent ces fichiers, afin qu'ils puissent être analysés et transformés en une documentation facilement exploitable.

Le comportement de la commande peut être paramétré via une fichier de configuration nommé `doxyfile`, qui permet de préciser différentes options liées à la manière dont le documentation va être générée : format de sortie, dossier de sortie, génération de diagrammes, etc. L'utilitaire `doxywizard` est une interface graphique permettant de spécifier de manière plus interactive ces différents paramètres, sachant que le fichier `doxyfile` est facilement éditable et compréhensible, ce qui permet de se passer éventuellement de cette interface.

Dans le cadre de ce projet, le fichier `doxyfile` a été généré et vous est fourni. Il est paramétré pour analyser tous les fichiers source qui se trouvent dans le même dossier que celui où il est placé et générer une documentation html, qui sera placée dans le sous-dossier `html`. Vous n'avez *a priori* pas à le modifier.

Les *tags* spécialisés

De nombre *tags* (ou mots-clé) spécialisés peuvent être intégrés dans un bloc de documentation, afin d'introduire une information particulière. Ils sont introduits par le caractère \, suivi du mot-clé. L'exemple ci-dessous, issu du fichier `mesurer.cpp`, montre l'utilisation de 3 tags :

— le tag `\file`, qui permet de spécifier le nom du fichier concerné par le commentaire ;
— le tag `\brief`, qui permet de donner un résumé de la documentation figurant dans ce bloc ;
— le tag `\author`, qui permet d'indiquer l'identité de l'auteur du code.

```
/**
 * \file mesurer.cpp
 *
 * \brief Fichier principal d'une application permettant de calculer la distance
 * intertextuelle entre deux textes.
 *
 * Cette application correspond à la réalisation du projet 2017/2018 du module
 * d'Algorithmique Avancée de seconde année de Licence Informatique
 * à l'Université du Littoral Côte d'Opale. Elle a pour but de mesurer
 * une distance intertextuelle entre deux textes, afin de déterminer si ceux-ci
 * ont été écrits par le même auteur ou pas.
 *
 * \author  ----- A COMPLETER -----
 */
```

La figure 4 ci-dessous illustre le résultat obtenu sur une partie de la page de documentation html du fichier `mesure.cpp`.

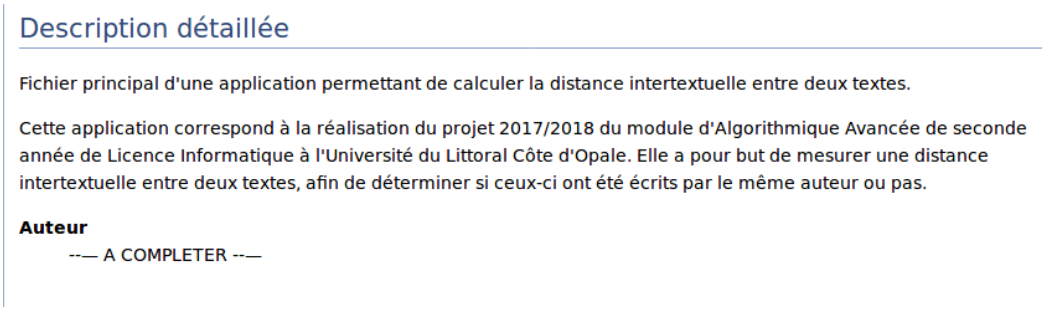

FIGURE 4 – Aperçu du bloc de documentation générale du fichier `mesurer.cpp`.

Quelques tags classiques sont donnés ci-après, sachant que vous trouverez la liste complète des tags et leur syntaxe dans la documentation *doxygen* et des exemples d'utilisation de la plupart d'entre-eux dans les sources commentés qui vous ont été fournis :

— `\author` pour identifier l'auteur de la partie à documenter ;
— `\brief` pour fournir une description courte d'une partie à documenter, afin d'avoir un aperçu rapide de celle-ci ;
— `\class` pour forunir toutes les informations concernant une classe ;
— `\def` pour documenter une définition `#define` ;
— `\enum` pour documenter un type énuméré ;
— `\fn` pour documenter une fonction ;
— `\file` pour expliquer le contenu d'un fichier ;
— `\param` pour donner des informations sur un paramètre d'une fonction ou d'une méthode ;
— `\return` pour donner des informations sur la valeur de retour d'une fonction ou d'une méthode ;
— `\see` pour indiquer au lecteur de la documentation qu'une autre partie de celle-ci donne des indications complémentaires.