TP n°7

Programmation à objets avancé : langage C++

La librairie standard C++ : *STL*

L'objectif de ce TP est de découvrir quelques aspects de la librairie standard de C++ : les conteneurs, les algorithmes et la gestion des chaînes de caractère. Ce TP se déroulera sur plusieurs séances.

La documentation officielle du langage et de la librairie standard du C++ peut être consultée à cette adresse : `http://en.cppreference.com/`.

Préambule

Afin de mettre en pratique les éléments de la STL, nous allons réaliser dans ce TP un indexeur de document textuel permettant une recherche de documents par mot-clé.

Le principe de l'indexation repose sur l'extraction des mots d'un ensemble de document et de l'association de chaque mot unique du corpus à l'ensemble des documents qui le contienne. Cette structure, appelée index inversé, sera construite petit à petit tout au long du TP.

Pour tester vos algorithmes, vous pourrez utiliser le jeu d'essai fourni sur *madoc* comprenant 6675 textes issus des comptes rendu des points presse du porte-parole du Quai d'Orsay de 2008 à 2012. Chaque fichier représente un point presse donné. La première ligne de chaque fichier correspond au titre donné à ce point presse, vient ensuite une ligne vide puis le texte retranscrit du point presse. La source de ces documents est accessible sur l'OpenData français `http://data.gouv.fr`.

1 Construction d'une *stop words list*

Afin d'extraire les mots des documents à votre disposition pour réaliser l'indexation, il est indispensable de filtrer les mots non informatifs des documents et plus particulièrement les mots de liaison (par exemple *le*, *la*, *les* en français).

Cette étape est réalisée en utilisant des listes de mots pré-calculées : les *stop words lists*. Ces listes sont évidemment dépendantes de la langue utilisée et sont généralement constituées par des linguistes. Il est cependant possible de construire cette liste automatiquement sur des bases statistiques, à partir du moment ou la quantité de texte à notre disposition pour une langue donnée est suffisante. C'est ce que nous nous proposons de réaliser dans ce premier exercice.

L'analyse statistique de nombreux textes a montré que la fréquence d'apparition d'un mot dans un corpus en fonction de son rang dans le document - fonction décroissante du nombre d'occurrences du mot - forme une courbe similaire à une hyperbole. La courbe de fréquence de la figure 1 illustre cette situation. Cette courbe suit en réalité *la loi de Zipf* qui établit que le produit de la fréquence d'un terme par son rang est approximativement constant. Cette loi a été vérifiée par George Kingsley Zipf au début du XX^{e} siècle sur la base de journaux américains en anglais mais s'applique sur tout corpus linguistique.

Or dans un texte, la valeur informative d'un mot peut s'exprimer sous la forme d'une gaussienne en fonction du rang des termes d'un document comme montré dans le figure 1. En effet, les mots les plus fréquents d'un texte représentent généralement des mots de liaison ou d'usage courant et ne

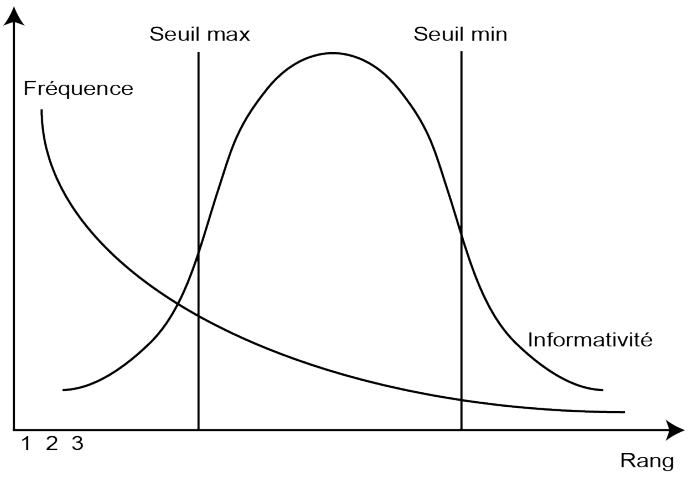

FIGURE 1 – Relation entre le rang et la fréquence d'apparition d'un terme dans un document.

véhiculent pas ou peu d'information sur la signification d'un document, et les mots les moins fréquents n'apportent que rarement un sens primordial à la compréhension d'un texte.

Hans Peter Luhn, un pionnier de l'indexation textuelle a utilisé cette constatation comme hypothèse de base pour spécifier deux seuils de coupure (seuil max et seuil min sur la figure) déterminant les mots apportant le plus d'information pour caractériser le sens d'un document et éliminant les mots sans signification. Les mots au-delà du seuil maximum sont considérés comme trop communs et ceux en deçà du seuil minimum comme trop rares. La probabilité d'obtenir un mot discriminant le contenu d'un document est maximale entre les deux seuils et tend à diminuer dans toutes les directions atteignant des valeurs proches de zéro aux points de coupure. Il n'y a pas de solution idéale pour établir ces seuils et la meilleure méthode consiste à procéder par essais successifs par rapport à un corpus donné. Cependant, dans le cas général, l'intervalle $\left[\frac{|\mathcal{C}|}{100}, \frac{|\mathcal{C}|}{5}\right]$ où $|\mathcal{C}|$ est le nombre de mots dans le corpus, est considéré comme adéquat pour fournir des termes avec un bon pouvoir discriminant.

Sur l'ensemble du TP nous auront à extraire des mots de différents textes. Nous commencerons par créer une fonction générique d'extraction de mots d'un document puis nous construirons un index permettant de calculer les fréquences d'apparition des mots dans le corpus traité. Enfin, nous trierons ces mots par leur fréquence afin de déterminer les mots devant figurer dans la *stop words list*.

Afin de pouvoir traiter l'ensemble des fichiers, une librairie multi-plateforme de parcours du contenu d'un répertoire vous est fourni dans les fichiers `FileHandling.hpp` et `FileHandling.cpp`. Cette librairie prends la forme d'une classe `FileHandling` et d'une fonction `IterateOnFileDir`. Cette dernière fonction simplifie l'utilisation de cette librairie en permettant d'itérer sur chaque nom de fichier contenu dans un répertoire par un appel à un foncteur prenant un paramètre : la chaîne de caractère contenant le nom d'un fichier.

QUESTION 1. Créez une fonction `main` (à partir du template fourni sur *madoc*) dans laquelle vous utiliserez la fonction `IterateOnFileDir` afin d'afficher à l'écran le contenu d'un répertoire. Le premier paramètre de cette fonction corresponds au répertoire à examiner et le deuxième au foncteur à utiliser. Le premier paramètre template permet de produire un affichage par palier indiquant le nombre de fichiers traités : une valeur de 0 n'affichera rien ; une valeur de 5 affichera un message tous les 5 fichiers. Le deuxième paramètre template permet de limiter le nombre de nom de fichier passé au foncteur : une valeur de 10 ne traitera que les 10 premiers noms de fichiers contenu dans le répertoire.

QUESTION 2. Créez une fonction `ExtractWords` dont le but est d'extraire les mots d'un fichier. Cette

fonction *template* prendra en paramètre un foncteur à appeler à chaque découverte d'un mot, et un nom de fichier à traiter. Dans un premier temps, limitez le code de cette fonction à lire le contenu d'un fichier ligne par ligne (fonction `std::getline`) et testez votre appel au foncteur en lui donnant par exemple dans un premier temps en paramètre chaque ligne extraite, le foncteur se limitant à afficher à l'écran de ces lignes.

QUESTION 3. Modifiez le contenu de la fonction `ExtractWords` afin de pouvoir extraire tous les mots de chaque ligne du fichier. Cette extraction sera réalisée à l'aide d'une expression régulière (classes `std::regex`, `std::sregex_iterator`). Afin d'extraire un mot dans la langue française, vous utiliserez le pattern fourni dans le listing 1 : un mot commence par une lettre, un chiffre ou un caractère `_` et contient par la suite des lettres, des chiffres, des caractères `_` ou des caractères `-`.

Listing 1 – Pattern d'extraction de mots.

```
std::regex pattern(R"#(\w[\w\-]*)#");
```

QUESTION 4. Modifiez l'appel au foncteur pour lui envoyer en paramètre une référence constante vers la ligne courante, le numéro de la ligne, le numéro du mot extrait et le mot extrait. Ce foncteur retournera un booléen dont la valeur *faux* ordonnera de passer immédiatement à la ligne suivante du fichier.

QUESTION 5. Modifiez votre fonction afin que chaque mot soit converti en minuscule. Pour celà, utilisez l'algorithme `std::transform` et la méthode `std::tolower`.

QUESTION 6. Créez une classe `CWordStat` dont le but sera de compter les occurrences des mots dans l'ensemble des fichiers. Ce comptage sera réalisé à l'aide de la classe `std::map`. Ajoutez un foncteur à cette classe compatible avec celui utilisé dans la fonction `ExtractWords`. Appliquez le comptage sur les fichiers du corpus (utilisez un sous-ensemble pour le débuggage).

QUESTION 7. Ajoutez une méthode à la classe `CWordStat` permettant de trier les mots en fonction de la fréquence d'apparition de ces mots dans les documents. Le tri sera réalisé dans un `std::vector` par l'algorithme `std::sort` par ordre décroissant de la fréquence des mots.

QUESTION 8. Écrivez dans un fichier les mots les plus fréquents (du premier au total/100) et les moins fréquents (du total/5 au dernier). Ce fichier sera la *stop words list* à utiliser dans le reste du TP.

2 Construction d'un index inversé

Nous allons maintenant réaliser l'indexation des documents en utilisant un index inversé. Le terme *inversé* vient de l'utilisation d'une table reliant les mots des documents aux documents ; la représentation des corpus sous la forme de textes étant une table reliant les documents aux mots contenus dans les documents.

Pour améliorer la pertinence, nous trierons également les documents liés à un mot en utilisant un critère lié à la fréquence d'apparition des mots dans le document.

QUESTION 1. Créez une classe `CIndex` munie d'un constructeur à un paramètre : le nom d'un fichier contenant une *stop words list*. Ce constructeur chargera cette liste de mot dans un conteneur `std::set`.

QUESTION 2. Sur le modèle de la classe `CWordStat`, créez dans la classe `CIndex` un foncteur compatible avec celui utilisé dans la fonction `ExtractWords`. Ce foncteur sera chargé de compter les fréquences d'apparition des mots dans chaque document (voir questions suivantes).

QUESTION 3. Créez une structure `SDoc` interne à la classe `CIndex` et non accessible à un code externe à la classe. Le but de cette structure est de stocker les informations propres à chaque document : le nom du fichier correspondant, son titre ainsi que la structure contenant les fréquences d'apparition de chaque mot du document.

Question 4. Complétez le foncteur créé dans la question 2 selon le processus suivant. Pour chaque nouveau fichier, créez une instance de `SDoc` stockée dans un conteneur `std::vector`. Pour éviter les duplications de structures `SDoc` (qui vont être relativement lourdes), ce `vector` stockera des pointeurs vers des instances allouées par un `new` (faites en sorte de désallouer proprement ces éléments dans le destructeur de `CIndex`). Pour chacun de ces documents, stockez le nom du fichier, le titre correspondant et complétez la structure de comptage de fréquence des mots dans les documents des structures en utilisant la fonction `ExtractWords` créée précédemment. Attention à ne pas comptabiliser les fréquences des mots contenus dans la *stop words list*, et attention à la casse des mots !

Question 5. Une fois le comptage de fréquence terminé, nous allons construire l'index inversé reliant une liste de document à chaque mot contenu dans le corpus. Pour cela, ajoutez un conteneur de type `std::unordered_map` adapté à une recherche rapide de texte et capable de stocker cet index inversé. À chaque mot du texte, nous associerons un `std::vector` contenant des pointeurs vers les `SDoc` correspondant aux documents. Complétez le foncteur créé dans la question 2 en mettant à jour cet index pour chaque nouveau mot détecté dans un document.

Question 6. Ajoutez une méthode `Calculate` à la classe `CIndex` dont le but sera d'effectuer le tri des listes de documents associés à chaque mot de l'index inversé, selon un ordre basé sur une fonction de pertinence. Cette fonction de pertinence met en œuvre une mesure de l'importance de chaque mot dans un document : pour un mot donné, on désire afficher les documents qui le contiennent dans l'ordre de l'importance du mot dans ce document.

À ces fins, nous allons utiliser une mesure classique en traitement de texte : la mesure *tfidf*. Elle consiste simplement à calculer pour chaque mot unique d'un document un score égal à la fréquence relative du mot dans le texte, pondéré par le nombre de documents contenant ce mot : $tfidf = \frac{tf}{df}$ avec tf la fréquence d'un mot dans un document ($tf = \frac{\#mot}{\#mots}$) et df le nombre de documents contenant ce mot. Un terme est donc d'autant plus important qu'il apparait fréquemment dans un document et dans très peu de documents.

Pour chaque mot de chaque document, calculez la valeur $tfidf$ que vous stockerez conjointement au nombre d'occurence de chaque mot de la structure `SDoc`. Triez ensuite tous les vecteurs de documents associés à chaque mot de l'index inversé en fonction du score $tfidf$ du mot de l'index et des documents concernés (fonction `std::sort`).

3 Interrogation de l'index

Maintenant que l'index inversé est créé, l'interrogation de notre corpus par des mots clés peut être réalisé rapidement. Pour simplifier le traitement, nous n'autoriserons l'interrogation du corpus que par un unique mot clé.

Question 1. Proposez une méthode `PrintDocs` à la classe `CIndex` prenant en paramètre un mot (chaîne de caractères) et affichant à l'écran la liste des documents contenant ce mot, trié par l'importance relative du mot dans le document.

Question 2. Déterminez la *stop words list* sur l'ensemble des documents du corpus (compilez et exécutez votre programme en *Release* pour éviter de ne perdre trop de temps).

Question 3. Calculez l'index inversé sur l'ensemble des documents du corpus fourni (compilez et exécutez votre programme en *Release* pour éviter de ne perdre trop de temps).

Question 4. Interrogez votre index inversé pour quelques mots du corpus ; par exemple *croissante*, *licorne*, *albanie*, *corruption*, *déclarations*, *centrafrique*. Attention à rendre vos différentes méthodes insensibles à la casse dans toutes vos fonctions !

4 Mesure de performance et parralélisme

Certains calculs nécessaires à la réalisation de l'index inversé sont couteux. Une des solutions à notre disposition pour améliorer le temps de calcul est d'utiliser le parallélisme. Chaque étape de l'indexation est parallélisable... Avec plus ou moins de difficulté.

Dans l'exercice qui suit, nous nous contenterons de paralléliser le tri des listes de documents associés à chaque mot de l'index inversé. Ce n'est clairement pas le goulet d'étranglement le plus important du processus (au contraire des entrées/sorties), mais c'est certainement le plus facile à réaliser sur les machines des salles de TP : il ne pose aucun problème de concurrence et les processeurs des machines sont multicœurs.

Dans la première étape nous allons mettre en place une mesure simple de performance basée sur le temps. Nous établirons ensuite plusieurs mesures en fonction du nombre de threads utilisés pour le calcul afin de déterminer le meilleur paramétrage pour la machine utilisée.

QUESTION 1. Créez une nouvelle méthode `ThreadCalc` à la classe `CIndex`. Recopiez dans cette méthode le calcul de tri sur l'index inversé.

QUESTION 2. Ajouter une mesure du temps écoulée autour de l'appel à la fonction `ThreadCalc`. Pour cela vous utiliserez la classe `std::chrono::high_resolution_clock` afin d'obtenir une mesure du temps la plus précise possible.

QUESTION 3. Faites en sorte de répéter la mesure de temps sur plusieurs appels à `ThreadCalc` et d'effectuer une moyenne pour obtenir des résultats plus précis et moins sujets aux activités réalisées en parallèle sur votre machine. Il faut cependant veiller à ce que les tests se déroulent dans les mêmes conditions, et donc que l'index soit dans le même état avant appel à la fonction de tri. Pour cela, créez simplement une recopie de l'objet stockant l'index inversé avant d'effectuer le calcul et écrasez l'index avec cette copie avant chaque calcul.

QUESTION 4. Créez une méthode `LaunchThreadCalc` dans la classe `CIndex` prenant en paramètre un entier `n` permettant de contrôler le nombre de thread à lancer. Pour lancer un thread, utiliser un objet `std::thread`.

La fonction à lancer dans le thread devra être une lambda qui devra exécuter la fonction `ThreadCalc` de l'instance courante de `CIndex`. Pour ce faire, capturez le pointeur `this` (vous aurez probablement à capturer d'autres paramètres dans la suite des exercices). Vérifiez que votre code fonctionne avec une valeur `n` fixée à 1 pour ne lancer qu'un seul thread ; les résultats obtenus doivent être similaires au code existant avant l'ajout du thread.

Une fois l'ensemble des threads lancés, attendez dans la méthode `LaunchThreadCalc` la fin du calcul de chaque thread en utilisant la méthode `thread::join` sur chaque objet `thread` créé.

QUESTION 5. Pour lancer plusieurs threads de calcul, il est nécessaire d'indiquer à chaque thread de ne traiter qu'une partie de l'index tout en gardant à l'esprit qu'aucun thread ne doit traiter les mêmes données. Cependant, votre index est une table de hachage pour laquelle il n'existe pas d'itérateur à accès direct (vous pouvez faire avancer un itérateur que d'un élément à la fois). Une des stratégies possible consiste alors, par exemple dans le cas de 3 threads, de demander au premier threads de ne traiter que les listes 0, 3, 6, ... ; au dexième les lsites 1, 4, 7, ... ; et au troisième les listes 2, 5, 8, ...

Vous allez pour cela modifier le prototype de la méthode `ThreadCalc` en lui passant 2 entiers en paramètre : le nombre de threads et le numéro du thread utilisé. Modifier ensuite le code de cette méthode afin de ne traiter que les listes correspondant au numéro de thread passé en argument de la fonction.

QUESTION 6. Testez pour plusieurs valeurs de nombre de thread les performances obtenus par notre méthode de parallélisation.