Programmation Multi-cœurs — TP 3
Recherche récursive d'expressions régulières sur le Web

Le TP peut être fait en Java ou en C++. Cependant, un fichier d'outils est déjà fourni aux programmeurs Java pour les lectures de pages Web. Ces outils utilisent la bibliothèque JSoup, qui n'ont pas d'équivalent aussi simple et portable en C++. Il est donc ***très fortement*** *recommandé de travailler en Java. Le code obtenu n'est pas à rendre à la suite du TP, mais servira de brique de base pour une partie du projet.*

1 Indexation du Web

Pour pouvoir répondre efficacement aux requêtes des utilisateurs, Google Search maintient un index de toutes les pages du Web, accompagné de la liste des mots qu'elles contiennent. Cet index est établi grâce à des *robots d'indexation* (en anglais *web crawlers*) qui explorent les pages Web en suivant leurs hyperliens.

Le but de ce projet est d'implémenter notre propre robot d'indexation pour rechercher des expressions régulières sur le Web. Par exemple, la commande suivante doit rechercher toutes les pages contenant le mot « Nantes » accessibles depuis la page Wikipédia de la ville.

```
$ java WebGrep Nantes https://fr.wikipedia.org/wiki/Nantes
https://fr.wikipedia.org/wiki/Nantes
https://fr.wikiquote.org/wiki/Nantes
https://fr.wiktionary.org/wiki/Nantes
https://fr.wikivoyage.org/wiki/Nantes
...
```

Algorithme à implémenter. Pour obtenir les résultats, on commencera par chercher une occurrence de l'expression dans la page donnée en argument. Si la page contient l'expression, on continuera la recherche en suivant tous les liens de la page, récursivement. Le temps de recherche étant largement dominé par les accès réseaux de chargement de pages, on veut accélérer la recherche en parallélisant le programme. Le but du TP est de paralléliser l'application.

Une classe `Tools` est mise à votre disposition en Java pour effectuer les tâches séquentielles. Elle contient principalement les méthodes statiques suivantes :

`Tools.initialize(args)` parse les arguments du programme à la recherche de l'expression rationelle, des URL de départ et des options d'affichage. La liste d'options peut être affichée par la commande `Tools.initialize(``--help'')`. Après initialisation, la liste des URL de départ peut être obtenue par la fonction `Tools.startingURL()` et le nombre de threads à instancier peut être obtenu par la fonction `Tools.numberThreads()`.

`Tools.parsePage(address)` récupère la page HTML à l'adresse indiquée, analyse son code et retourne un objet `page` de type `ParsedPage` donnant accès à la liste des liens hypertextes (`page.hrefs()`) et la liste des éléments HTML contenant l'expression rationelle recherchée (`page.matches()`) dans la page.

`Tools.print(page)` produit un affichage textuel représentant la `ParsedPage` passée en argument, en respectant les options d'affichage passées lors de l'initialisation de `Tools`.

Architecture du programme. L'architecture distribuée du programme est dictée par les trois contraintes suivantes

1. La recherche doit être parallélisée entre n threads. Il faudra donc créer un thread pool de n threads, auquel on passera une tâche pour chaque page à parcourir.
2. Il ne faut pas explorer deux fois la même URL. On gardera en mémoire une structure de données encodant l'ensemble des pages explorées. La vérification qu'une page n'est pas présente dans l'ensemble et son insertion doivent être dans un même bloc atomique (pourquoi ?). Il est conseillé de se renseigner sur la classe `ConcurrentSkipListSet`, et en particulier sa méthode `add` pour régler ce point sensible.
3. Il ne faut pas d'entrelacement entre les sorties pour deux pages. La méthode statique `Tools.print` fournie n'est pas thread safe. On suggère de dédier un thread aux sorties textuelles, selon le modèle des producteurs et des consommateurs : les producteurs sont les threads du thread pool qui produisent des objets de type `ParsedPage`, et le consommateur est le thread qui gère la sortie.

2 Dictionnaire de chaînes de caractères

La mémoire nécessaire au stockage des adresses est le facteur limitant dans l'algorithme précédent. On souhaite développer une structure de données qui stocke les chaînes de caractères sous forme compressée, en mutualisant l'espace de stockage pour les préfixes communs des chaînes. Par exemple, si l'on veut stocker les chaînes "chameau" et "chat", on peut stocker le préfixe commun "cha", puis les suffixes "t" et "meau". Un dictionnaire est une structure de donnée qui encode un ensemble de chaînes de caractères sous forme d'arbre. L'implémentation séquentielle (pas thread-safe) de gestion d'un dictionnaire vous est donné dans le fichier `Dictionary.java`. L'exemple ci-dessous illustre l'encodage de l'ensemble {"chameau", "chameaux", "chamelle", "chamelles", "chamelon", "chamelons", "chat", "chaton", "chatons", "chats", "chatte", "chattes"}. Les nœuds sont entourés deux fois si leur champ `present` vaut **true**, et les $\perp$ gris signifient que les pointeurs `next` (vers le bas) ou `suffix` (vers la droite) valent **null**. Vous devez transformer cette implémentation pour la rendre linéarisable et wait-free, et l'utiliser dans votre Web crawler. On pourrait compresser encore plus les données en stockant des chaînes de caractères dans les nœuds au lieu de caractères. Quels problèmes se poseraient alors ? Comment pourrait-on gérer les suppressions ?